Intel® Xeon Phi™ Processor High Performance Programming

Knights Landing Edition

Intel Xeon Phi Processor High Performance Programming

Knights Landing Edition

By

Jim Jeffers

James Reinders

Avinash Sodani

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann is an imprint of Elsevier

Morgan Kaufmann is an imprint of Elsevier
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, USA

Cover art: Knights Landing Die Photo, courtesy of Intel Corporation. Superimposed over the die is an ENZO cosmology simulation of the intergalactic "cosmic web" of gravitational filaments that link galaxies, define the structure of the universe, and indicate dark matter. Simulation by Brian O'Shea of Michigan State University. Visualization generated on Knights Landing processor using VisIt open source application by Jefferson Amstutz, Intel Corporation, with Anne Bowen and Dave Semeraro from the Texas Advanced Computing Center.

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

Disclaimer

Results in this book are based on preproduction Knights Landing processors, which may differ from results with the actual Knights Landing products.

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN: 978-0-12-809194-4

Publisher: Todd Green
Editorial Project Manager: Lindsay Lawrence
Production Project Manager: Mohana Natarajan
Cover Designer: Matthew Limbert

Typeset by SPi Global, India

Acknowledgments

Stone Soup is an ancient folk story in which hungry travelers visit a town and ultimately convince wary townsfolk into contributing food for their soup making endeavor. The travelers set up in the center of the town, confidently boiling a pot of water that initially contained only a local stone. They commented to curious onlookers from the town that it would be delicious, and everyone could partake, but it would be a little better if they had a few additional ingredients. It was the additional food added, that the townsfolk had initially hid from the travelers, which ultimately made the soup a treat for all.

James has frequently described projects as "Stone Soup." This book became another of his "Stone Soup" exercises. It started with Jim and James anticipating updating their original book for programming Knights Corner, the original Intel® Xeon Phi™ product, in time for Knights Landing. They were thrilled that the head architect for Knights Landing, Avinash, agreed to work on the book as a coauthor. The terms of being a coauthor were explicit: each coauthor must contribute several chapters and must review *every* chapter in the book. Avinash has been an invaluable addition to the team.

What we had not anticipated was how many people would volunteer to help. Their help transformed a project from an update to a complete rewrite with incredible content born from early experiences with Knights Landing. This book is much more comprehensive and useful than would have been possible with just Jim, James, and Avinash.

In the Contributors section, we acknowledge 56 people (42 from Intel, 14 from outside Intel), who wrote and submitted specific substantial portions of this book. The line between those 56 and the additional people we thank here is not always distinct because of the amount of time and effort so many people beyond the 56 contributed. All the contributors are grateful for the steady source of feedback and review given by James, Jim, and Avinash, as well as their support, patience, and gentle encouragement. We will not repeat their thanks chapter by chapter! We also try to not mention the 56 contributors again in this section—we thank them by listing them in the Contributors section of this book.

James thanks Mike Julier for puzzling out how to best update the "worth a thousand words" graph in Chapter 1.

Lukasz Anaczkowski, Chris Cantalupo, Jaroslaw Kogut, Krzysztof Kulakowski, Ruchira Sasanka, Piotr Uminski, and Stan Whitlock were very helpful in the creation of Chapter 3.

Karthik Raman supplied key passages for the descriptions of Knights Corner versus Knight Landing coding that we used in Chapter 6. Karthik was also very helpful with his feedback.

We are thankful for Jim Cownie's feedback especially on the indirect branching description and advice found in Chapter 6.

We are thankful to John Pennycook for sharing his evolution/revolution storytelling and review of Chapter 7 to introduce key concepts for parallel programming.

We are thankful to the critical editing that greatly improved Chapters 8, 9, and 18 by Susan Meredith.

We thank Jan Zielinski and Piotr Uminski for feedback and suggestions for Chapter 18 (Offload to Knights Landing). More than three grosze worth indeed.

Zakhar thanks Nancee Moster and Stephen Blair-Chappel for their outstandingly detailed reviews, and Asaf Hargil, Martyn Corden, Ilyas Gazizov, Serguei Preis, and Nikita Astafiev for their insightful technical feedback, for Chapter 10 on the Vectorization Advisor.

Mark Charney and Shuo Li were helpful with inputs and suggestions while preparing Chapter 12 on AVX-512 intrinsics.

Ramesh and Larry thank Mike Chynoweth, Sumedh Naik, and Dmitry Prohorov for help and review on Chapter 14 for Tuning and Timing.

Heinrich thanks Klaus-Dieter Oertel, Dmitry Sivkov, and James Tullos for help and review on Chapter 15 on MPI.

We thank Sayan Ghosh and Erik Schnetter for carefully reviewing the text of the PGAS chapter, Chapter 16, and the associated code examples.

For Chapter 17, Jim, Jeff, and co-contributors would like to thank Attila Afra for his work and contributions to the Embree section including the shading coherence and ray streaming implementation, Knights Landing code runs, and results images. Also to Sven Woop, Johannes Günther, and Greg P. Johnson for their inputs and work on Embree and OSPRay. Special mention goes to Carsten Benthin whose knowledge and effort to perform early ports of Embree to Knights Landing were critical to making the chapter a reality.

For his work on Chapter 19, about Power, Claude is grateful for review by Federico Ardanaz, Alex Gutkin, and Sushmith Hiremath. Claude also appreciates direction, input, and essential feedback from Micah Barany.

For Chapter 23, the N-Body Chapter, Andrey and Ryo are grateful to John Pennycook for his feedback, which has made the presentation in the chapter clear and helped us reported facts correctly.

For Chapter 25, Trinity mini-app workload performance results, analysis, and source code optimizations on preproduction Knights Landing were truly a team effort. We would like to thank Gregg Skinner for his guidance on MiniFE and on affinitization schemes, Indraneil Gokhale for his guidance on SNAP, Karthik Raman for his guidance on MiniGhost, Sudha Udanapalli Thiagarajan for her guidance on AMG, Ruchira Sasanka for his guidance on MiniDFT, and Ashish Jha for his guidance on MILC.

We thank Balint Joo for his help with introduction and detailed reviews for Chapter 26 on Quantum Chromodynamics.

We thank the entire Knights Landing architecture, design, and validation teams past and present. The Knights Landing processor has so many innovations that it is practically a design from scratch. In Intel's now-famous terminology of "tick" and "tock," where a "tick" processor has small architectural changes and mainly rides

the process-node change, while a "tock" has significant architectural changes, Knights Landing is probably the grandest "tock" to date. Knights Landing would not have been possible without the tireless, self-less, and single-minded dedication of countless people over many years. Big teams worked day and night, making many personal sacrifices, to get this processor done. While it is hard to name every major contributor, some people come to mind as we look back into the history of Knights Landing. Roger Gramunt, Ramon Matas, Ram Padmanabhan, Jonathan Hall, and Alexey Suprun were some of first to join the project and played a key role in defining the new core for Knights Landing along with Julio Gago, Santi Galan, Roger Espasa, Chung-Lun Chan, and Faisal Azeem. Jesus Corbal, Dennis Bradford, and Tom Fletcher who were responsible for bestowing Knights Landing with its vector performance. Anurag Sah oversaw the design and validation of the new core from start to finish. Toni Juan, Yen Chen-Liu, Pau Cabre, Krishna Ganapathy, Bahaa Fahim, and others helped bring the 2D-mesh on-die interconnect to life. Rajat Agarwal, Steven Hutsell, Sundaram Chinthamani, Darrell Mcginnis, and Derek Thompson developed the high-bandwidth memory architecture in Knights Landing. Krishna Vinod and Zainul Aurangabadwala were responsible for formulating the tile architecture and the coherence protocol that went with it; Giri Venkata helped put 38 of these tiles on the Knights Landing die. Many others, such as Ho-seop Kim, Rob Kyanko, Alan Kyker, Micah Barany, Federico Ardanaz, Yogesh Bansal, Ben Chaffin, and Sundaram Chinthamani, helped shoulder many important responsibilities such as microarchitecture, power management, firmware, and performance. The design and validation teams worked tirelessly over many years. It is hard to single out everyone here, but their contributions were immense. Aakash Tyagi was the design manager for the longest time and was one of the pillars of the project from the very beginning. Amit Goel and Ravi Kuppuswamy helped see the project through to a successful completion under their watchful eyes. We cannot be more indebted to these and many other individuals whom we could not name due to space constraint for the Knights Landing processor.

For multiple aspects of the book, we are thankful for input, discussions, review, and support from Jerry Baugh, Russ Beutler, Martin Corden, Kirti Devi, Eric Gardner, Craig Garland, Milind Girkar, Nicole Grieve, Lars Jonsson, Dick Kaiser, Jeongnim Kim, Steve Lionel, Hongjiu Lu, David Mackay, Paresh Pattani, Mike Pearce, Paul Petersen, Dave Poulsen, Deepika Punyamurtula, Arch Robison, Elizabeth Schneider, Tatiana Shpeisman, Lisa Smith, Philippe Thierry, Martin Watt, and Georg Zitzlsberger. For contracts review and other legal help, we thank Sonja Alpert. For help on artwork, we thank Mark Grabow, Jeremy Gates, Curt McKay, and Fiona Solliday.

We were blessed with the support and encouragement of some special managers in Intel including Atiq Bajwa, Joe Curley, Barry Davis, Diane Bryant, Bob Burroughs, Mike Greenfield, Raj Hazra, Herb Hinstorff, Sailesh Kottapalli, Nathan Schultz, Thor Sewell, Sanjiv Shah, and Charlie Wuischpard.

James Reinders thanks his wife Susan Meredith for her love, support, review, and invaluable advice. James also thanks his daughter Katie and son Andrew for their

wonderful and positive support. Many thanks to my coauthors and friends, Jim Jeffers and Avinash Sodani. We could not have had a better team for this book. Incredibly busy and super knowledgeable people who found time for our Stone Soup in a big way.

Jim Jeffers thanks his wife Laura for showing him what it really means to be strong and for her boundless support and encouragement. Jim also thanks his children (including spouses), Tim, Patrick, Colleen, Sarah, Jon, and his very special granddaughter Hannah for keeping him grounded and always proud. Thanks to my incredibly talented Intel Visualization teammates and collaborators already mentioned, plus Carson Brownlee and George Kyriazis, as well as all his industry friends and partners including Hank Childs, Jian Huang, Berk Geveci, Dave DeMarle, Ken Martin, Paul Navratil, Dan Stanzione, Kelly Gaither, Aaron Knoll, Chris Sewell, Ken Morrison, Paul Shellard, Juha Jäykkä, and the Cambridge "COSMOS" team who enable Jim to have one of the best jobs in the world. Many thanks to Avinash Sodani for his great insight and commitment to delivering a ground breaking Knights Landing product and to a similarly worthy book. Jim thanks James Reinders for once again rising to the occasion to (sometimes not so) gently encourage (cajole?) his coauthors and our plethora of contributors to go the extra mile to meet tight publishing deadlines while delivering the highest quality information possible. This book simply would not have been completed without James' organization and over-the-top effort. And finally, a special message for James: this really is my last Stone Soup.

Avinash Sodani is truly grateful to his family—his wife, Shilpa, his children, Abhigya and Ayush, his parents, and his brothers in India—for their boundless love and support throughout his career, including during the many years spent in developing the Knights Landing processor. Without their steadfast support, it would not have been possible to complete this long and eventful journey. Avinash will also like to thank Jim and James for involving him in writing of this book. This was an incredible (first time) experience; he learnt many important lessons from both of them on good, clear writing.

We appreciate the hard work by the entire Morgan Kaufmann team including the three people we worked with directly: Todd Green, Lindsay Lawrence, and Mohana Natarajan.

Numerous colleagues offered information, advice, and vision. We are sure that there are more than a few people who we have failed to mention who have positively impacted this book project. We thank all those who helped by slipping in their ingredients into our Stone Soup. We apologize for all who helped us and were not mentioned here.

Thank you for help in the kitchen with our Stone Soup,

Jim Jeffers, James Reinders, and Avinash Sodani

Foreword

In their first *Intel® Xeon Phi™ Coprocessor High-Performance Programming* book, Jeffers and Reinders utilize a sports car analogy to introduce high performance computing. I like this analogy because I see many parallels between the automotive and computing worlds. There are drivers who see vehicles solely as appliances that get them from point A to point B. Perhaps these are people who look forward to self-driving cars. Similarly, there are computer users that are not concerned with performance—they are willing to wait however long is needed for their applications to complete. But if you are reading this book, you are likely very interested in computer performance. For you, a sports car analogy is appropriate for how you drive your computers. It is good to see that the authors' sports car introductory tutorial is available to an even wider audience at their open website, http://www.lotsofcores.com/sportscar.

EXTENDING THE SPORTS CAR ANALOGY TO HIGHER PERFORMANCE

I work at Sandia National Laboratories, and as with many of my colleagues at the U.S. Department of Energy national laboratories, we are interested in the highest possible performance. To carry forward the automotive analogy, we could say that as extreme performance supercomputer users, we are drivers of race cars. We want not only to reach our modeling and simulation goals as fast as possible but also to look under the hood, understand what's there, and modify the race car to go even faster. It is also accurate to note that there is a larger population of national lab users that would be considered sports car drivers—they don't develop or modify the applications they use. They are interested in performance but don't have the time to tune and optimize their applications for the highest possible performance. Instead, these HPC users are focused on their scientific research or engineering analysis, so for them, supercomputers with their relevant modeling and simulation applications are tools to support their R&D.

I must also confess that I am an automotive enthusiast. I recall the thrill of receiving my driver's license, prefer driving a stick shift, and enjoy taking long road trips. I won't be a customer for the self-driving car. As an undergraduate mechanical engineer at the University of Illinois, I took every automotive engineering course that was offered. My vehicle dynamics class was taught by Professor Robert A. White, who also consulted for the Porsche Research Center. This explains why I read with great interest, *The Unfair Advantage*,[1] a book by Mark Donohue about his career driving

[1]Mark Donohue with Paul Van Valkenburgh, *The Unfair Advantage*, Robert Bentley, Inc., Cambridge, MA, USA, second edition, 2000.

and developing race cars. I especially enjoyed Chapter 25, which describes the collaboration that Donohue and the Penske racing team established with Porsche on the development of the Type 917 race car.

WHAT EXACTLY IS *THE UNFAIR ADVANTAGE*?

Mark Donohue was not just a race car driver. He was also an automotive engineer who was able to translate what he felt through the throttle, brake pedal, steering wheel, and racing seat into improvements to his race car. These could be simply the tuning adjustments that mechanics would make. Or these could be more drastic changes that were needed in the design of sub-systems that an engineer would make, for example, changes to suspension geometry, or chassis bodywork and airflow. I was impressed by Donohue's following passage:

> *We knew a lot about the engineering they were doing, and we had already come to some of the same conclusions. He [Helmut Flegl, the Porsche 917 chief engineer] had been engineering race cars for some years, and as he began to realize that we could relate, I could almost see a spark come to his eyes. There are certain things that are of interest only to racing engineers–like lateral acceleration in "g's," aerodynamic downforce, centers of mass–and we spoke the common language. We began to convince him that we were not like any other race team he had ever worked with.*[2]

The Penske racing team entered into an agreement with Porsche to collaborate directly on the development of the turbocharged version of the 917 race car when it was still a preproduction prototype. As a mechanical engineer, Donohue was able to communicate with the 917 race car designers in engineering terms. As a driver, Donohue had a first-hand understanding of the user requirements to refine and complete the final development of the 917. This was Donohue's *Unfair Advantage*.

PEAK PERFORMANCE VERSUS DRIVABLE/USABLE PERFORMANCE

In 1971, the Porsche engineers originally designed their 917 race car engine to produce maximum horsepower. Donohue and the Penske racing team were the first people outside of Porsche to receive this turbocharged engine. Unfortunately, according to Donohue, the 917 simply would not idle or run at part throttle. No amount of tuning by their mechanics could make the engine work. In the end, the Penske team had

[2]Ibid., page 282.

to stop testing the turbocharged engine in the 917 race car, and they went back to the Porsche engine test stands and dynos. I pick up Donohue's description:

> *I looked at their dyno output curves. They had all the necessary data–torque, rpm, boost pressure, and so on–except that the curves started at 5,000 rpm. I said, "Why are there no curves up to that point?" They said, "The motor does not run there." I thought "Christ! That's what I've been trying to tell you for a month!" I couldn't believe it was that simple. I couldn't believe that they had simply calibrated the fuel injection for wide open throttle with full boost, and totally ignored any part-throttle operation. Flegl and I sat down and designed a dyno program to get the information we needed for proper calibration.*[3]

In short order, the turbocharged engine was properly calibrated to operate at all engine speeds and turbo boost pressure levels. This is the driver's perspective on *The Unfair Advantage*. Engineers can be misled by simple benchmarks that do not reflect how high performance systems are actually used. As a driver, Donohue needed the engine to operate well at all engine speeds from idle on up to redline. The Porsche engineers were not to blame for this oversight, and they delivered what was asked for—an engine designed for maximum horsepower. They did not understand the user perspective because they were not drivers.

Let me segue back to the analogy between supercomputers and race cars. In the supercomputing community, our baseline metric is High Performance LINPACK (HPL) and the units of measure are floating point operations per second. This benchmark is used by the Top500 supercomputing sites list,[4] and has been useful for providing a simple measure that can be used to characterize system performance. However, HPC users also understand that their real applications are usually not represented by how the HPL benchmark tests supercomputer capabilities. In recent years, a singular focus on HPL has led computer and system engineers to design supercomputers that can generate peak HPL benchmark measurements but have not been very usable for real-world HPC applications. The concepts of useable and drivable are synonymous for high-performance systems whether they are supercomputers or race cars.

In the end, Donohue and the Penske racing team helped Porsche "complete" the engineering and development of their 917 race car, with key collaborative improvements in a variety of sub-systems from suspension geometry and engine tuning to vehicle aerodynamics. The bottom line is that as a driver, that is, user of the race car system, Donohue had insights regarding how best to design and tune the system for optimal performance. But as an engineer, he was able to communicate how to improve the Porsche 917 design.

---

[3] Ibid., page 289.
[4] See http://top500.org/.

HOW DOES *THE UNFAIR ADVANTAGE* RELATE TO THIS BOOK?

Section I is for readers that consider themselves to be race car "engineers/drivers." They are interested in extracting the highest performance potential of the underlying hardware. Within the national labs and in some university and commercial settings, these users are likely to be developers of architecture-centric software capabilities. For example, they may need to develop highly tuned and optimized math libraries that extract performance from all the architectural capabilities that were designed into Intel® Xeon Phi™ Knights Landing by Avinash Sodani and his processor architecture team. The audience for Section I may include users that are interested in the development of new system software to support the mapping of many HPC applications to Xeon Phi processor architectures. Finally, the audience for Section I may also be interested in how codesign can influence the design of future hardware.

Section II is for the race car and enthusiast sports car driver of Xeon Phi supercomputers. The chapters in this section describe how to program the Intel® Xeon Phi™ for those users that are interested in realizing the performance potential of the new Knight Landing architectural capabilities. These chapters are for readers that need to understand how to develop HPC applications to leverage the new architectural capabilities that are provided by the Xeon Phi Knights Landing. This audience may also be interested in how codesign principles are used to understand the tuning and application modifications needed to exploit the various advanced architecture features provided in the Knights Landing processor.

Section III is for the sports car driver. This section of the book provides application examples with Knights Landing results for quickly coming up to speed for a diverse portfolio of HPC applications. It may be possible to directly apply the lessons and patterns in these application examples to the reader's own applications, or the reader may already be a user of these applications in their own HPC workloads. Section III is also for readers that are interested in seeing examples for how the parallel programming concepts of Section II are implemented in real applications.

CLOSING COMMENTS

Sandia National Laboratories is the DOE/National Nuclear Security Administration's (NNSA) engineering lab. I have often discussed with my colleagues that as an engineering lab, we are in a position to foster opportunities to collaborate with industry to help develop our needed supercomputers and supporting computing technologies. Recently, I helped write the NNSA Advanced Simulation and Computing (ASC) program's Codesign Strategy.[5] This document describes different levels of codesign including *transformative* codesign as a way to influence future hardware designs.

[5] *ASC Co-design Strategy*, NNSA's Advanced Simulation and Computing Program, NA-114, February 2016, http://nnsa.energy.gov/sites/default/files/nnsa/inlinefiles/ASC_Co-design.pdf.

While not every supercomputer user wants to help develop supercomputer technology, we have many scientists and engineers at Sandia's Center for Computing Research that are not afraid of driving first-of-a-kind, preproduction prototype computers. Yes, there is a risk of crashing, but the opportunity to drive leading edge computer systems is often one of the main attractions we can offer to our technical staff. The point is not just to have early access to hardware. These early testbeds foster direct collaborations between our adventurous "engineer-drivers" and our computer engineering/computer science counterparts in industry. These collaborative discussions are the payoff for being the first to drive new HPC technology. This is the "spark in the eye" described in Donohue's early conversation with Flegl when they realized they had a common engineering understanding.

I am grateful that my team at the Center for Computing Research has had similar conversations with Jim Jeffers and his Intel colleagues through our early access to three generations of preproduction versions of Intel® Xeon Phi™ products. I would also like to acknowledge the long-term collaboration that Sandia has established with support from many Intel executives including Raj Hazra, Charlie Wuischpard, Joe Curley, Thor Sewell, Mike Julier, Ranna Prajapati, Thomas Metzger, and Rajesh Agny to establish our NNSA/ASC-funded series of first-of-a-kind, Intel® Xeon Phi™ products codenamed Knights Ferry, Knights Corner, and Knights Landing rack-scale testbeds. My Sandia team includes James Laros III, Simon Hammond, Sue Kelly, Jim Brandt, and Ann Gentile, with strong management and programmatic support from Bruce Hendrickson, Ken Alvin, Rob Hoekstra, Tom Klitsner, and John Noe. These collaborations with Intel have helped us jointly develop lessons learned on our preproduction testbeds that we can go forward to apply on our DOE production Xeon Phi and Xeon-based supercomputers.

I believe you will find this book is an invaluable reference to help develop your own *Unfair Advantage*.

James A. Ang, Ph.D.
Manager, Exascale Computing Program,
Center for Computing Research,
Sandia National Laboratories,
Albuquerque, New Mexico, USA
March 2016

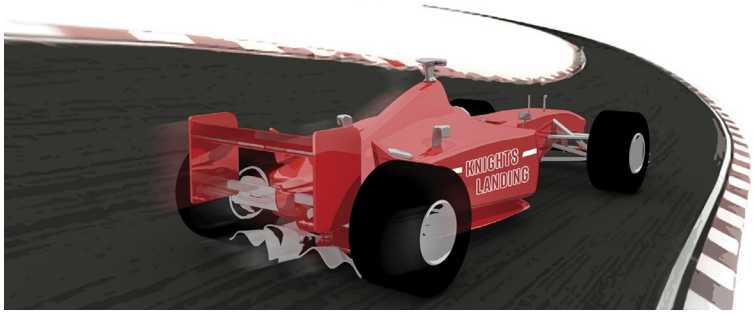
Preface

When we published the original *Intel Xeon Phi Coprocessor High-Performance Programming* in early 2013, we said that we hoped to publish a second book with examples. We actually published *two* volumes that have become popularly known as "Pearls 1" and "Pearls 2." Officially they are titled *High-Performance Parallelism Pearls*, volumes *One* and *Two*. For inclusion in these Pearls books, we were blessed with high quality work from around the world illustrating parallel programming. Ultimately, 132 people contributed to those two volumes on parallel programming for many-core and multicore alike.

With this new book, we revisit teaching the essentials of Intel Xeon Phi processor programming in order to replace the original *Intel Xeon Phi Coprocessor High-Performance Programming* book. We are also building on the Pearls books which remain highly relevant for Knights Landing. We are excited to have Avinash Sodani join this effort. Avinash is the chief architect of second generation of Intel® Xeon Phi™ products (code named *Knights Landing*). While the three of us pulled together this book, we were also gratified by numerous contributors who joined the effort by writing sections and chapters. We also had many great reviewers who helped with this book a great deal.

This completely new book focuses on parallel programming methods to take best advantage of the many enhancements included in Knights Landing. Much has happened since our first book introduced the first generation of Intel® Xeon Phi™ products (code named *Knights Corner*). We are excited about the second generation of Intel Xeon Phi products now available combined with years of experience using Knights Corner. These experiences have consistently highlighted the "dual-tuning" value of the Intel® Many Integrated Core (MIC) architecture at the heart of all Intel Xeon Phi products. This "dual-tuning" characteristic for programming Intel Xeon Phi products has caused software developers to share their excitement with comments such as "We had no idea how wonderful *Xeon* was, until *Xeon Phi* came along" and "Tuning for *Xeon Phi* always gives us substantial gains on *Xeon* as well." That is because many-core programming has significantly motivated code updating to utilize parallelism well. Such work is rewarded with performance gains for many-core and multicore processors using the same code. Hence the term "dual-tuning." This updating has been called "code modernization" #ModernCode.

We started off this book project with a relatively modest goal of "just updating" our original book. The key to programming remains the same—it is "just" parallel programming, albeit at the very high level possible with many-core. However, we wanted to add more real world examples that were tuned for the high degree of parallelism available with Intel Xeon Phi processors—something that was not available to us in 2012. Also, we wanted to cover a number of topics in more depth (eg, vectorization) and introduce some new material to address continuing progress for OpenMP 4.x, MPI 3, and PGAS models.



SPORTS CAR TUTORIAL: INTRODUCTION FOR MANY-CORE IS ONLINE

We made a decision to make space available for newer material in this book by taking out our popular "sports car" introduction to programming the Intel Xeon Phi processors. This allows this book to dedicate more pages to advanced programming concerns, but it removes three chapters that have been praised by instructors for their learning value. Because of their great value as introductory material, we updated those three chapters and have placed them free online at http://lotsofcores.com/sportscar.

If you are completely new to programming Intel Xeon Phi products, we recommend reading the introduction to programming (http://lotsofcores.com/sportscar). In the three chapters that are online, we walk through steps to realize the full performance potential of Knights Landing.

PARALLELISM PEARLS: INSPIRED BY MANY CORES

In the past few years, we shared many examples of parallel programming in the form of books titled *High-Performance Parallelism Pearls*, volumes one and two. When we prepared the first volume, we were pleased to repeatedly see speed-ups on significant real world applications that many people thought had previously been well optimized for parallelism. In one early example that came to our attention, a team happily showed us optimizations that gave them a very real 10× performance difference on Intel Xeon Phi coprocessors. What was most surprising however was the very real 5× speed-up they reported on Intel Xeon processors. While we had expected "dual-tuning" benefits, we had not anticipated how wide spread the lack of prior optimization would be. Of course, we asked the developers why they had not done these optimization years before. The answer was "4 or 8 cores is not that exciting, but 61 cores… that's exciting!"

Therefore, while we offer some tips in this book specific to getting the most from Knight Landing, you will find most of the programming techniques we cover to be the keys to general parallel programming. These keys are useful for any general-purpose parallel computer. The challenges of parallel computing are simple to enumerate: expose lots of parallelism and minimize communication (said another way: maximize locality). Every programmer we have talked with who has optimized for Knights Corner or Knights Landing has told us that their code runs faster on Intel Xeon processors as well as parallel systems with non-Intel processors.

This also means that *High-Performance Parallelism Pearls*, volumes one and two, will remain outstanding illustrations of effective parallel programming for years to come and will be valuable to anyone reading this book. The MCDRAM available in Knights Landing offers some new opportunities for applications, including some applications that were previously optimized in a Pearls book. Section III of this book, aptly titled *Pearls*, includes chapters that have such optimizations.

ORGANIZATION

We do assume some familiarity with many-core programming. If you are entirely new to many-core programming, we recommend you take a look at three introductory chapters, available freely on the web at http://lotsofcores.com/sportscar. The "sports car" introductory text is organized to get started programming quickly while touching on many key concepts.

After an introductory Chapter 1, this book if organized into three distinct sections:

Section I: Knights Landing. This section focuses on Knights Landing itself, diving into the architecture, the high-bandwidth memory, the cluster modes, and the integrated fabric. Like any computer, really understanding it gives us the best application tuning opportunities to utilize it well. This section builds that deep understanding. However, for first time, many-core programmers, Chapters 2–6, may be an excessive amount of information and could be skipped after reading Chapter 1 by proceeding directly to read Sections II and III of this book. Ultimately, Section I contains the Knights Landing details that any expert programmer desiring optimal performance will want to understand.

Chapter 1 describes the motivation for many-core and provides an overview of many-core programming. Chapter 2 gives an introductory overview of the Knights Landing architecture. Chapter 3 discusses application usage of the high-bandwidth memory (MCDRAM) and cluster modes. Chapter 4 is an in-depth look at the Knight Landing tile architecture, the mesh interconnect that connects the tiles, and the related memory and cluster modes. Chapter 5 details the Intel® Omni-Path fabric architecture. This section finishes with a chapter dedicated to low-level optimization tips that are most tied to Knights Landing itself. While most of the book focuses on optimizations that are generally universally useful for parallel programming, Chapter 6 (μarch Optimization Advice) provides deeper insight into the workings

of Knights Landing microarchitecture and the "do's and don'ts" for performance that expert programmers will enjoy.

Section II: Parallel Programming. This section focuses on application programming with consideration for the scale of many-core. Chapter 7 discusses when code changes for parallelism can be an evolution versus a revolution. Chapter 8 covers tasks and threads with OpenMP and TBB in particular. Chapters 9–12 include an overview of many data parallel vectorization tools and techniques. Chapter 13 covers libraries. Chapter 14 discusses profiling tools and techniques including roofline estimation techniques and ways to pick which data structures to place in the MCDRAM. Chapter 15 covers MPI. Chapter 16 takes a look at a collection of "rising stars" that provide PGAS style programming—including OpenSHMEM, Coarray Fortran, and UPC. While these will not generally be the most efficient parallel programming techniques for Knights Landing, there is a lot of merit in exploring this space during the upcoming years starting with Knights Landing. Chapter 17 discusses the emerging benefits of parallel graphics rendering using Software Defined Visualization libraries. Chapter 18 (Offload to Knights Landing) discusses offload style programming including programming details on how to offload to a Knights Landing coprocessor, or across the fabric to a Knights Landing processor. Chapter 18 is a "must read" if you have code that used Knights Corner offload methods, or if you will use Knights Landing in offload mode. We discuss "Offload over fabric" which may be of interest for building on offload code investments and maximizing application performance in clusters with a mix of multicore and many-core processors. Chapter 19 discusses Power Analysis on Knights Landing enabling insight into the growing emphasis on energy efficiency.

Section III: Pearls. This section focuses on parallel programming in real-world applications providing examples with notes on Knights Landing specific results and optimizations. Chapter 20 gives an overview of this section and a high-level summary of each individual chapter. Chapters 21 and later build upon *High-Performance Parallelism Pearls*, volumes one and two by being among the first published examples using the high-bandwidth memory and the clustering modes in Knights Landing. We believe that we all learn a great deal when we see the work of other experts. We highly recommend benefiting from the excellent work shared in Section III of this book and both of the Pearls books. The techniques found in these many examples really bring home how to harness the power of multicore and many-core processors, including Knights Landing.

STRUCTURED PARALLEL PROGRAMMING

The topic of teaching parallel programming is very much on our minds. This book *Intel Xeon Phi Processor High-Performance Programming* combined with the volumes of *High-Performance Parallelism Pearls* are powerful learning tools for mastering many-core and multicore programming. Nevertheless, we would be remiss to suggest that they provide a complete foundation for Parallel Programming. While

there are detailed books on OpenMP, MPI, and TBB to consider for mastery, there are also foundational elements worth investing time to master for a strong intuitive ability to "Think Parallel."

If you will step back to really consider the fundamentals of parallel programming, we recommend *Structured Parallel Programming* by Michael McCool, Arch Robison, and James Reinders. It is a text formed solidly around a belief in "patterns that work" as a key to teaching parallel programming. This sentiment has gained traction with some universities seeking ways to teach fundamentals, not just techniques. James is fond of saying "if you do not know how to use Map, Reduce, Pipeline, and Stencil patterns in a parallel program, you should learn them before you jump into parallel programming." This is akin to recommending that one learn queues, stacks, lists, parsing, and other fundamentals on any journey toward mastering programming in general. You'll find that some colleges have posted materials related to using the *Structured Parallel Programming* book in university classes. There is more information, downloads, and links on http://parallelbook.com.

WHAT'S NEW?

We start every chapter with a terse summary of what key elements are new specifically due to Knights Landing in that chapter. This serves as a guide to the highlights in a chapter which would differ from a parallel programming book for Knights Corner or an Intel® Xeon® processor.

An example of a summary like those found at the beginning of each chapter. We hope that these up-front summaries will help those familiar with Knights Corner, or parallel programming for other processors, find what we believe are most important to best utilize Knights Landing's processing capabilities. We also think everyone can benefit from honing in on these important recommendations representing the latest advances in high-performance computing architectures.

What is new with Knights Landing in this chapter?
This entire chapter is about things that are new with Knights Landing.

We have also included a detailed glossary and index, at the end of the book, to help support jumping around in the book.

lotsofcores.com

Our website http://lotsofcores.com has been very popular for downloading materials associated with our books. We make available all the figures and diagrams from the books, code examples, errata, and supplemental information (such as the introductory "sports car" tutorial) at this website (http://lotsofcores.com). These online

materials help teach parallel programming suitable for highly parallel many-core processors as well as multicore processors.

We hope this book enables you to share our excitement in parallel computing that is especially inspired by the Intel MIC architecture and Intel Xeon Phi processors (Knights Landing).

Jim Jeffers, James Reinders and Avinash Sodani
Intel Corporation
May 2016

SECTION I

Knights Landing

This section focuses on Knights Landing itself, diving into the architecture, the high bandwidth memory, the cluster modes, and the integrated fabric. Like any computer, really understanding it gives us the best application tuning opportunities to utilize it well. This section builds that deep understanding. However, for first-time, many-core programmers, Chapters 2–6 may be an excessive amount of information and could be skipped after reading Chapter 1 by proceeding directly to read Sections II and III of this book. Ultimately, this section contains the Knights Landing details that any expert programmer desiring optimal performance will want to understand.

Chapter 1 describes the motivation for many-core and provides an overview of many-core programming. Chapter 2 gives an introductory overview of the Knights Landing architecture. Chapter 3 discusses application usage of the high bandwidth memory (MCDRAM) and cluster modes. Chapter 4 is an in-depth look at the Knight Landing tile architecture, the mesh interconnect that connects the tiles, and the related memory and cluster modes. Chapter 5 details the Intel® Omni-Path fabric architecture. This section finishes with a chapter dedicated to low-level optimization tips that are most tied to Knights Landing itself. While most of the book focuses on

optimizations that are generally universally useful for parallel programming, Chapter 6 (μarch Optimization Advice) provides deeper insight into the workings of Knights Landing micro-architecture and the "do's and don't's" for performance that expert programmers will enjoy.

This book has three sections: I. Knights Landing, II. Parallel Programming, and III. Pearls. The book also has an extensive Glossary and Index to facilitate jumping around the book.

CHAPTER

1 Introduction

What is new with Knights Landing in this chapter?

Knights Landing is a processor, with a coprocessor option, and uses hyper-threading. Knights Landing also offers high bandwidth memory and fabric support on package. Knights Corner was only available as a coprocessor and used multithreading. Otherwise, the core message of the chapter is the same as for any processor: tune for parallelism.

In this book, we bring together the essentials to high performance programming for an Intel® Xeon Phi™ processor (or coprocessor). The architecture for the Second-Generation Intel Xeon Phi product is commonly known by the Intel code name *Knights Landing*. We will say "Knights Landing" throughout the book instead of the very long phrase "Intel Xeon Phi processor and/or coprocessor (or card)." In general, everything stated about "Knights Landing" applies to both "Intel Xeon Phi *processors*" and "Intel Xeon Phi *coprocessors*." Coprocessor and offload-specific topics are the focus of Chapter 18.

This book is organized into three sections, which cover the architecture specifics, parallel programming models, and real-world examples (pearls), respectively. Together, these sections emphasize essential knowledge to get the most out of Knights Landing.

Programming for Knights Landing is almost entirely about programming in the same way as we would for an Intel® Xeon® processor-based system, but with extra attention on exploiting lots of parallelism because of the uniquely high degree of scaling possible with Knights Landing. This extra attention pays off for processor-based systems as well due to an effect we have dubbed, the "dual-tuning" effect that comes from the compatibility that Knights Landing offers. This "dual-tuning" advantage is a key aspect of why programming for Knights Landing is particularly rewarding and helps protect investments in programming.

Knights Landing is both generally programmable and tailored to tackle highly parallel problems. As such, it is ready to accelerate very demanding parallel applications. We explain how to make sure an application is constructed to take advantage of such a highly parallel capability. As a natural side effect, these techniques generally improve performance on less parallel machines and prepare applications better for computers of the future as well. The overall concept can be thought of as "Portable High-Performance Programming."

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00001-6

INTRODUCTION TO MANY-CORE PROGRAMMING

If you are completely new to many-core programming, we encourage you to read http://lotsofcores.com/sportscar. Sports cars are not designed for a superior experience driving around on slow-moving congested highways. In our introductory tutorial for many-core programming, the similarities between Knights Landing and a sports car give us opportunities to step by step expose you to key techniques for scalable high-performance programming.

The freely downloadable http://lotsofcores.com/sportscar is an adaptation, for Knights Landing, of the first three chapters of our original book on programming for the Intel Xeon Phi coprocessor, titled *Intel Xeon Phi Coprocessor High-Performance Programming*. We had many complements on these chapters as a gentle and readable introduction to many-core programming; these chapters were used in many introductory classes. In the intervening three years, many programmers have learned many-core programming. We offer these three chapters on the web to help introduce more people to many-core programming, with the added bonus that this frees up space in this book for including more in-depth material. The sports car does achieve Knight Landing levels of performance in our updated material.

TREND: MORE PARALLELISM

To squeeze more performance out of new designs, computer designers rely on the strategy of adding more transistors to do multiple things at once. This represents a shift away from relying on higher instruction speeds, which demanded higher power consumption, to a more power-efficient parallel approach. Hardware performance derived from parallel hardware is more disruptive for software designs than speeding up the hardware because it benefits parallel applications to the exclusion of non-parallel programs.

It is interesting to look at a few graphs that quantify the factors behind this trend. Fig. 1.1 shows the end of the "era of higher processor speeds," which gives way to the "era of higher processor parallelism" shown by the trends graphed in Figs. 1.2 and 1.3. This switch is possible because, while both eras required a steady rise in the number of transistors available for a computer design, trends in transistor density continue to follow Moore's law as shown in Fig. 1.4. A continued rise in transistor density, perhaps at a reduced pace, will drive more parallelism in computer designs and result in more performance for applications using parallel programming.

Figs. 1.1–1.4 plot information regarding x86 (and x86-64) processor or coprocessor products made by Intel thus far. The trends of other processor architectures and manufacturers would reveal a similar halt to clock rate increases and a rise in parallelism. Plotting a single long running architecture from a single manufacturer helps us show these trends without other variables at play.

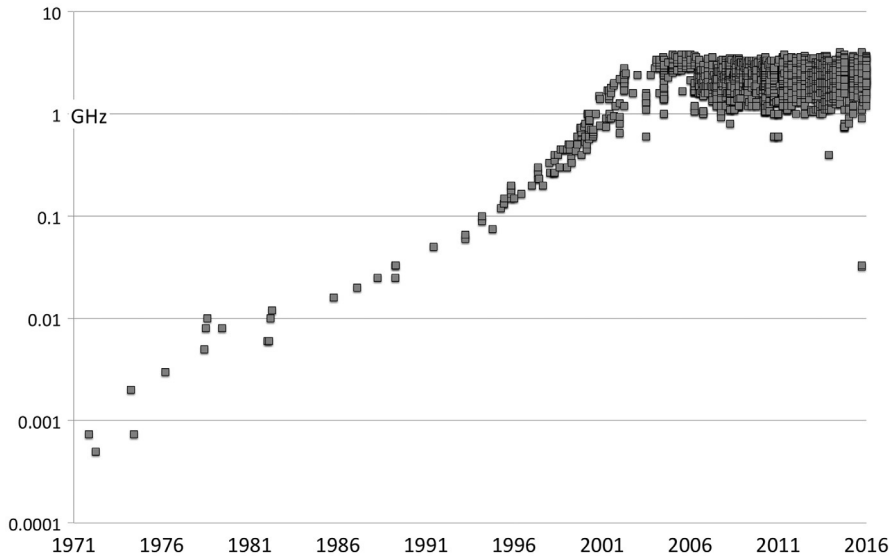


FIG. 1.1

Processor/coprocessor speed era (log scale).

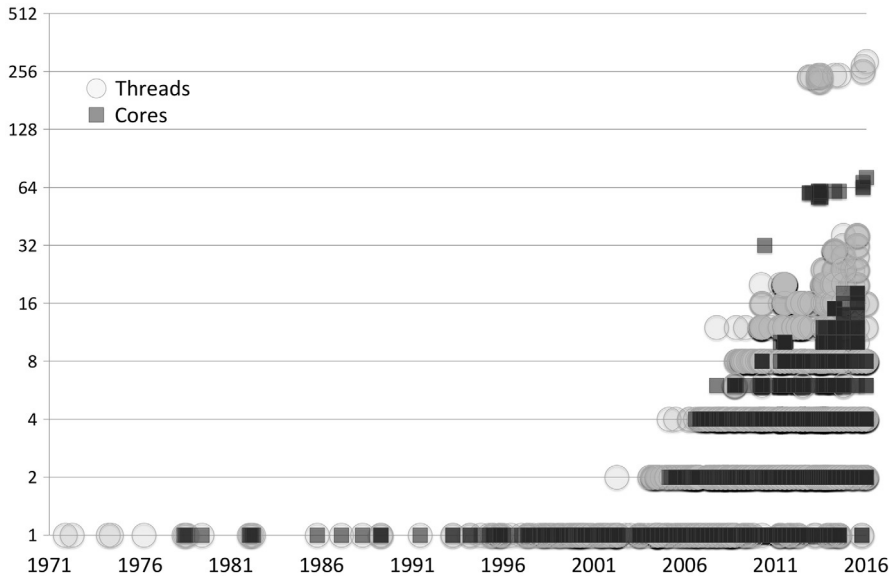


FIG. 1.2

Processor/coprocessor core/thread parallelism (log scale).

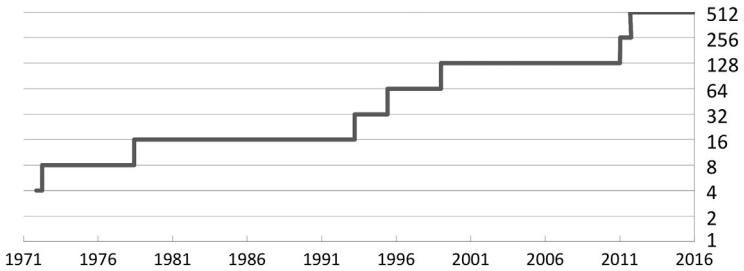


FIG. 1.3

Processor/coprocessor vector parallelism (width in bits) (log scale).

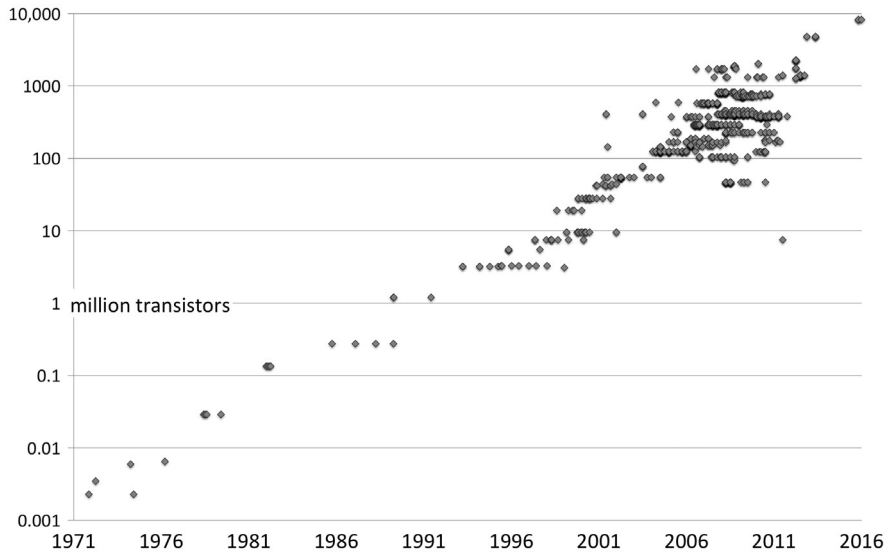


FIG. 1.4

Moore's law continues, processor/coprocessor transistor count (log scale).

WHY INTEL® XEON PHI™ PROCESSORS ARE NEEDED

Knights Landing is designed to extend the reach of applications that have demonstrated the ability to fully utilize the scaling capabilities of Intel Xeon processor-based systems and fully exploit available processor vector capabilities or memory bandwidth. For such applications, Knights Landing offers additional power-efficient scaling, vector support, and local memory bandwidth, while maintaining the programmability and support associated with Intel Xeon processors.

Many applications in the world have not been structured to fully exploit parallelism. This leaves a wealth of capabilities untapped on nearly every computer system.

Such applications can be extended in performance by a highly parallel device only when the application expresses a need for parallelism through parallel programming. This has given rise to calls for "code modernization" to move code forward to exploit the power of newer parallel processors including Knights Landing.

Advice for successful parallel programming can be summarized as "Program with lots of threads that use data organized for vectors with your preferred programming languages and parallelism models." Since most applications have not yet been structured to take advantage of the full magnitude of parallelism available in any processor, understanding how to restructure to expose more parallelism is critically important to enable the best performance for any processor including Knights Landing. This restructuring itself will generally yield benefits on most general-purpose computing systems, a bonus due to the emphasis on common programming languages, models, and tools. We refer to this bonus as the dual-tuning advantage.

It has been said that a single picture can speak a thousand words; for understanding Knights Landing (or any highly parallel device), it is Fig. 1.5 that speaks a thousand words. We used a graph like this in our programming book for Knights Corner, and in our classes to highlight that the Intel Xeon Phi products are designed to deliver unequaled performance on highly parallel code while Intel Xeon processors offer performance across a larger range of applications.

For Knights Corner, a graph like Fig. 1.5 is applied regardless of "vector intensity" (the density of floating point operations — see more in the vectorization chapters). However, assuming full utilization of the floating point math capabilities, Knights

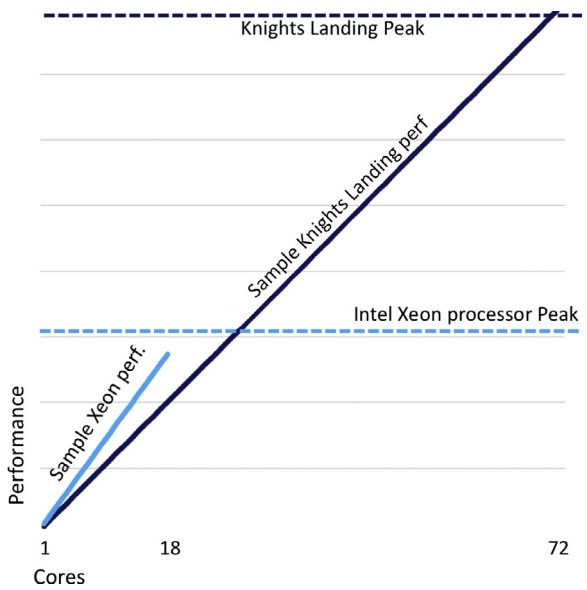


FIG. 1.5

This picture speaks a thousand words.

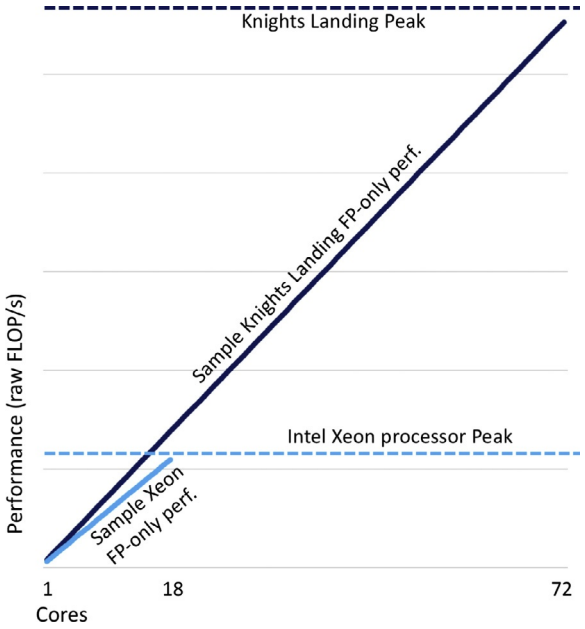


FIG. 1.6

Another thousand words based on Floating-point (FP) arithmetic performance only.

Landing yields Fig. 1.6. This is because, unlike a Knights Corner core, the Knights Landing cores have a higher FLOP/sec/core than contemporary Intel Xeon processors (Haswell architecture) on an FLOP/sec/core basis. Despite lower clock rates, Knights Landing features 512-bit wide vector instructions versus 256-bit wide vector instructions for the Haswell architecture. Knights Landing also has more cores.

Figs. 1.5 and 1.6 speak to this principle: Knights Landing offers the ability to make a system that can potentially offer exceptional performance while still being buildable and power efficient. Intel Xeon processors deliver performance for a broad range of applications but do have a lower limit on peak performance. In exchange for our investments in parallel programming, we may reach otherwise unobtainable performance while remaining portable. The "dual-tuning" double advantage of these Intel products comes from using the same parallelism models, programming languages, and familiar tools to greatly enhance preservation of programming investments.

PROCESSORS VERSUS COPROCESSOR

The original Intel Xeon Phi products, code named *Knights Corner*, were only available as coprocessors. Knights Corner was, in essence, a processor trapped in a coprocessor body. While this provided an excellent entry into the market and helped power the world's faster computer for years, limitations of the coprocessor

model had many people looking forward to an Intel Xeon Phi *processor*. A processor implementation is freed from limitations including limited memory size and PCIe transfers back and forth with a host processor.

The second-generation Intel Xeon Phi product, code named *Knights Landing*, is available as either a processor or a coprocessor. It is accurate to think of it as a processor that can also be packaged as a coprocessor. Most often in this book, we will refer to the first-generation Intel Xeon Phi products by their code name *Knights Corner*, and the second-generation Intel Xeon Phi products by their code name *Knights Landing*. We found this more readable than saying "Intel Xeon Phi (co)processors" or "Intel Xeon Phi processors and coprocessors." In Chapter 18, we address information specific to using Knights Landing as a coprocessor and in offload mode. We do refer to it as a Knight Landing *coprocessor*, in particular, when used in its coprocessor packaging. Likewise, in the few instances where we need to speak about the processor versions specifically, we will refer to it as a Knight Landing *processor*.

MEASURING READINESS FOR HIGHLY PARALLEL EXECUTION

To know if an application is utilizing parallelism well, we should examine how an application scales, uses vectors, and uses memory. We can get some impression of where we are with regard to scaling and vectorization by doing a few simple tests.

To check scaling, we can create a simple graph of performance of various runs each with various numbers of threads (from one up to the number of cores, with attention to thread affinity). This can be done with settings for OpenMP (i.e., OMP_NUM_THREADS for OpenMP) or Intel® Threading Building Blocks (TBB). If the performance graph indicates any significant trailing off of performance, we have tuning opportunities.

To check vectorization, we can compile an application with and without vectorization. For instance, using Intel compilers auto-vectorization: disable vectorization via compiler switches: `-no-vec -no-simd`, use at least `-O2 -xhost` for vectorization. Compare the performance you see. If the performance difference is insufficient, you should examine opportunities to increase vectorization. If you are using libraries, like the Intel Math Kernel Library, you should consider that math routines would remain vectorized no matter how you compile the application itself. Therefore, time spent in the math routines may be considered as vector time. Unless your application is bandwidth limited, the most effective use of Knights Landing will be when most cycles executing are vector instructions doing vector operations. While some may tell you that "most cycles" need to be over 90%, we have found this number to vary widely based on the application. If a program is bandwidth limited, the potential upside of using Knights Landing for its superior bandwidth may dominate instead of vectorization for determining the peak performance.

When using Message Passing Interface (MPI), it is desirable to see a communication versus computation ratio that is not excessively high in terms of communication. The ratio of computation to communication is a key factor in deciding between using offload

versus native model for coprocessor programming. Programs are also most effective using a strategy of overlapping communication and I/O with computation.

There are a number of tools available to assist in profiling and tuning at the system and node levels. These same tools support Knights Landing as well as other processors. Profiling and tuning are covered in detail in Chapter 14, while programming in general is the subject of Section II in this book.

WHAT ABOUT GPUS?

While graphics processing units (GPUs) cannot offer the programmability of Knight Landing, they accelerate some applications through scaling combined with vectorization or bandwidth. Applications that show positive results with GPUs should always benefit from Knights Landing because the same fundamentals of vectorization or bandwidth must be present in an application, although they may need to be expressed differently. The opposite is not true. The flexibility of Knights Landing includes support for applications that cannot run on GPUs. For instance, Knights Landing supports all the features of C, C++, Fortran, OpenMP, TBB, MPI, etc., while GPUs are restricted to special GPU-specific models (like NVidia CUDA) or subsets of standards like OpenMP. This is one reason that a system using Knights Landing will have broader applicability than a system using GPUs and offer greater portability and performance portability. Additionally, tuning for GPU is generally too different from a processor to have the "dual-tuning" benefit we see in programming for Knights Landing. This can lead to a substantial rise in investments to be portable across many machines now and into the future and restrict performance portability in a way that requires GPU-specific retuning generation to generation.

ENJOY THE LACK OF PORTING NEEDED BUT STILL TUNE!

Because Knights Landing is a compatible x86 SMP-on-a-chip, there is no Knights Landing-specific porting needed to run an application. However, the high degree of parallelism in Knights Landing is best suited to applications that are structured to use that parallelism. Almost all applications, which have not been tuned for many-core, will benefit from some tuning beyond the initial base performance to achieve maximum performance on such a highly parallel device. This can range from minor work to major restructuring to expose and exploit parallelism through multiple tasks and use of vectors. The experiences of users of Intel Xeon Phi products and the "forgiving nature" of this approach are generally promising but point out one challenge: the temptation to stop tuning before the best performance is reached. This can be a good thing if the return on investment of further tuning is insufficient and the results are good enough. It can be a bad thing if expectations were that working code would always be high performance. *There ain't no such thing as a free lunch*! The hidden bonus is the "dual-tuning," double advantage of programming investments for Intel

Xeon Phi products that generally applies directly to any general-purpose processor as well. This greatly enhances the preservation of any investment to tune working code by applying to other processors and offering more *forward scaling* to future systems.

TRANSFORMATION FOR PERFORMANCE

There are a number of possible user-level optimizations that have been found effective for ultimate performance. These advanced techniques are not essential. They are possible ways to extract additional performance for your application. The "forgiving nature" of Knights Landing makes transformations optional but should be kept in mind when looking for the highest performance. It is unlikely that peak performance will be achieved without considering some of these optimizations (Section II of this book revisits these in more depth):

- Memory access and loop transformations (e.g., cache blocking, loop unrolling, prefetching, tiling, loop interchange, alignment, affinity).
- Vectorization works best on unit-stride vectors (the data being consumed is contiguous in memory). Data structure transformations can increase the amount of data accessed with unit-strides (such as Array of Structures to Structure of Arrays transformations or recoding to use packed arrays instead of indirect accesses).
- Use of full (not partial) vectors is best, and data transformations to accomplish this should be considered.
- Vectorization is best with properly aligned data.
- Large page considerations (we recommend the widely used Linux libhugetlbfs library).
- Algorithm selection (change) to favor those that are parallelization and vectorization friendly.

HYPER-THREADING VERSUS MULTITHREADING

Knights Landing, like Intel Xeon processors, utilizes hyper-threading. In general, hyper-threading on Knights Landing will prove useful for increasing performance more often than on other prior processors. Knights Corner used a simpler multithreading. Knights Corner always required more than one thread running per core to reach maximal performance. Therefore, Knights Corner reached maximum performance with two, three, or four threads running per core. Knights Landing can reach maximum performance with one, two, or four threads per core.

Programs should parameterize the number of cores and the number of threads per core in order to easily run well on a variety of current and future processors and coprocessors. Therefore, the number of threads per core is usually set outside the application and we tune by checking the performance of using one, two, or four

threads per core on Knights Landing. While three threads per core is possible with Knights Landing, it is unlikely to match the performance of other choices. The architecture behind this is explained in more detail in Chapter 2.

PROGRAMMING MODELS

The most popular parallel programming model for a multimode system is known as MPI+X. This most often means MPI+OpenMP, and for many C++ programmers it means OpenMP+TBB. Section II of this book has chapters discussing all the key programming models in more detail. Programming using Partitioned Global Address Space (PGAS) models is on the rise, and well worth understanding. We discuss PGAS models in Chapter 16. The concept of programming with an "offload" model was popularized by GPUs and the Knights Corner coprocessor. The "offload" model is supported for Knights Landing coprocessors and Knights Landing processors as well. Programming via offloading is discussed in Chapter 18.

No popular programming language was designed for parallelism. In many ways, Fortran has done the best job adding new features, such as `DO CONCURRENT`, to address parallel programming needs. Fortran and C benefits from OpenMP. C++ users have embraced Intel TBB. C++ developers may use OpenMP as well. Section II of this book has chapters discussing the programming models in more detail; we offer some brief advice here.

Knights Landing offers the full capability to use the same tools, programming languages, and programming models as an Intel Xeon processor. However, because Knights Landing is designed for high degrees of parallelism, some models are more interesting than others.

In a way, it is quite simple: an application needs to deal with having lots of tasks, and deal with vector data efficiently (also known as vectorization). There are some recommendations we can make based on what has been working well for programming thus far. For Fortran programmers, use OpenMP, `DO CONCURRENT`, and MPI. For C++ programmers, use TBB, OpenMP, and MPI. TBB is a C++ template library that offers excellent support for task-oriented load balancing. While TBB does not offer vectorization solutions, it does not interfere with any choice of solution for vectorization. TBB is open source and available on a wide variety of platforms supporting most operating systems and processors. For C programmers, use OpenMP and MPI.

WHY WE COULD SKIP TO SECTION II NOW

Parallel programming is the key to Knights Landing. As such, until we have dealt with the critical aspects of parallel programming: scaling, vectorization, and locality; worrying about the rest of Section I in this book could be a distraction.

Of course, learning details about Knights Landing can be fun and very interesting. The rest of Section I (through Chapter 6) dives into just such fun. Chapters 2 and 3 supply an excellent overview for programmers of the keys to Knights Landing architecture and application usage of the on-package high-bandwidth memory, known throughout this book as MultiChannel Dynamic Random Access Memory (MCDRAM), and cluster modes. Chapters 4-6 dive deeply into the Knights Landing design and implications.

We would be remiss if we did not emphasize that the best starting point is to let the MCDRAM act as a cache (memory mode = cache) and leave the cluster mode set to its default (cluster mode = quadrant, in most systems). We would only encourage looking at other modes when fine-tuning an already tuned parallel application. When we have a tuned parallel application, then reading and acting on the rest of Section I will be more important and rewarding.

FOR MORE INFORMATION

We hope you read the Preface to this book. We discuss some history as well as the organization of this book in the Preface.

Some additional reading worth considering includes:

- Introduction to Many-core Programming with our Sport Car driving analogies (adopted from the original Intel Xeon Phi programming book, adapted for Knights Landing). http://lotsofcores.com/sportscar.
- Satish, N., C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. "Can traditional programming bridge the Ninja performance gap for parallel computing applications?," Computer Architecture (ISCA), 2012 39th Annual International Symposium on, vol., no., pp. 440–451, 9–13 June 2012.
- Code modernization resources: http://software.intel.com/moderncode.
- *Structured Parallel Programming: Patterns for Efficient Computation*, Michael McCool, Arch Robinson, James Reinders, Morgan Kaufmann Publishers, 2012, ISBN 9780123914439, http://parallelbook.com.

CHAPTER

Knights Landing overview 2

What is new with Knights Landing in this chapter?
This entire chapter is about things that are new with Knights Landing.

In this chapter, we describe the overall architecture of the Second-Generation Intel® Xeon Phi™ product code named *Knights Landing*. We follow with a look at how applications can best use the high-bandwidth MCDRAM and cluster modes in Chapter 3. We dive in-depth into the Knights Landing architecture in Chapter 4, and then the Knights Landing companion Omni-Path fabric architecture in Chapter 5. Programming optimizations specific to the Knights Landing micro-architecture are covered in Chapter 6. Sections II and III (Chapter 7 and beyond) focus entirely on parallel-programming tips, techniques, and examples targeting the extensive parallelism provided by Knights Landing.

OVERVIEW

Knights Landing is a many-core processor that delivers massive thread and data parallelism with high memory bandwidth. It is designed to deliver high performance on parallel workloads. Knights Landing provides many innovations and improvements over the prior-generation Intel Xeon Phi coprocessor code named Knights Corner. Knights Landing is a standard Intel Architecture standalone processor that can boot stock operating systems and connect to a network directly via prevalent interconnects such as Infiniband, Ethernet, or Intel® Omni-Path Fabric (Chapter 5). This is a significant advancement over Knights Corner, which is restricted as a PCIe-connected device and, therefore, Knights Corner could only be used when connected to a separate host processor. Knights Landing introduces a more modern, power efficient core that triples (3×) both scalar and vector performance compared with Knights Corner. Knights Landing offers over three TFLOP/s of double precision and six TFLOP/s of single precision peak floating point performance. It introduces a new memory architecture utilizing two types of memory (MCDRAM and DDR) to provide both high memory bandwidth and large memory capacity. It is binary compatible with prior Intel® processors. This means that it can run the same binary executable programs that run on

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00002-8

current x86 or x86-64 processors without requiring any modification to the binary executable programs. Knights Landing introduces the 512-bit Advanced Vector Extensions known as Intel® AVX-512. These new 512-bit vector instructions are architecturally consistent with the previously introduced 256-bit AVX and AVX2 vector instructions. The AVX-512 instructions will be supported in future Intel® Xeon® processors as well. Knights Landing introduces optional on-package integration of the Intel® Omni-Path Architecture enabling direct network connection with improved efficiency compared to an external fabric chip or card.

INSTRUCTION SET

The instruction set architecture (ISA) for Knights Landing is similar to any other x86 processor; it layers recently added new instruction groups on top of support for baseline x86/x86-64 instructions equivalent to recent Intel Xeon processors. Fig. 2.1 shows how the Knights Landing ISA compares with recent Intel Xeon processors. Knights Landing supports all legacy instructions including 128-bit SSE and SSE4.2, and 256-bit AVX and AVX2 technologies.

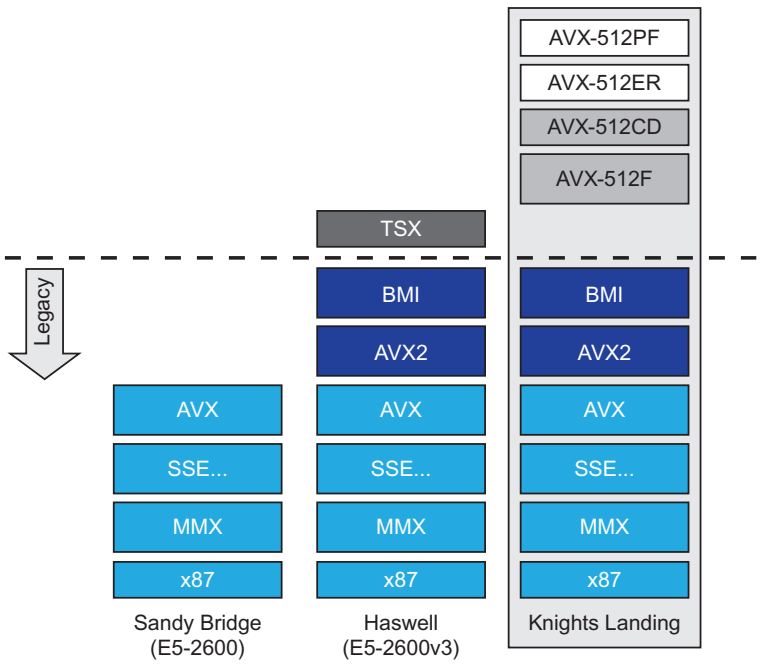


FIG. 2.1

Knights Landing ISA as compared to Intel Xeon processors.

Knights Landing introduces Intel® AVX-512 instructions. AVX-512 provides 512-bit SIMD support, 32 logical registers, 8 new mask registers for vector predication, and gather and scatter instructions to support loading and storing sparse data. Each AVX-512 instruction can perform eight double-precision multiply-add operations (16 FLOPs) or sixteen single-precision multiply-add operations (32 FLOPs), with an optional operand specifying a memory location that can either be a scalar or a vector.

Intel AVX-512 instructions consist of four categories. AVX-512 Foundation instructions (AVX-512 F) are the base 512-bit extensions as described previously. Three other categories are: Intel® AVX-512 Conflict Detection Instructions (CDI), Intel® AVX-512 Exponential and Reciprocal Instructions (ERI), and Intel® AVX-512 Prefetch Instructions (PFI). These capabilities provide efficient conflict detection for improved vectorization (e.g., for histogram updates), fast exponential and reciprocal operations for speeding up transcendental functions, and prefetch capabilities for sparse data, respectively.

AVX-512 has been defined with programmability in mind. Most AVX-512 programming occurs in high-level languages, such as C/C++ and Fortran, through vectorizing compilers and pragmas to guide the compilers or via libraries with optimized instruction sequences, which may use intrinsics.

Knights Landing does not implement Intel® Transactional Synchronization Extensions (TSX). As with other extensions, software is expected to confirm hardware support using the applicable CPUID feature bit when needed to ensure the ability to run on machines with and without these extensions.

ARCHITECTURE OVERVIEW

Fig. 2.2 shows a block diagram of the Knights Landing processor. The Knights Landing design has a concept of a tile, which is the basic unit for replication. Each tile, as shown in Fig. 2.3, consists of two cores, two vector-processing units (VPUs) per core, and a 1 M L2 cache shared between the two cores. These tiles are physically replicated 38 times on Knights Landing, however at most 36 of them are active; the extra two tiles are present to improve manufacturing chip yield and are disabled by the time the processor leaves the factory. Disabled cores consume essentially no power. In total, with 36 tiles, Knights Landing has 72 cores and 144 VPUs on the chip. Versions of Knights Landing, with fewer active tiles or less memory may be manufactured as well.

TILE

The core is a newly designed two-wide, out-of-order core derived from the Intel® Atom processor code named Silvermont. Knights Landing includes significant modifications to the Silvermont microarchitecture to incorporate many features targeting high-performance computing. These features include support for four hyperthreads

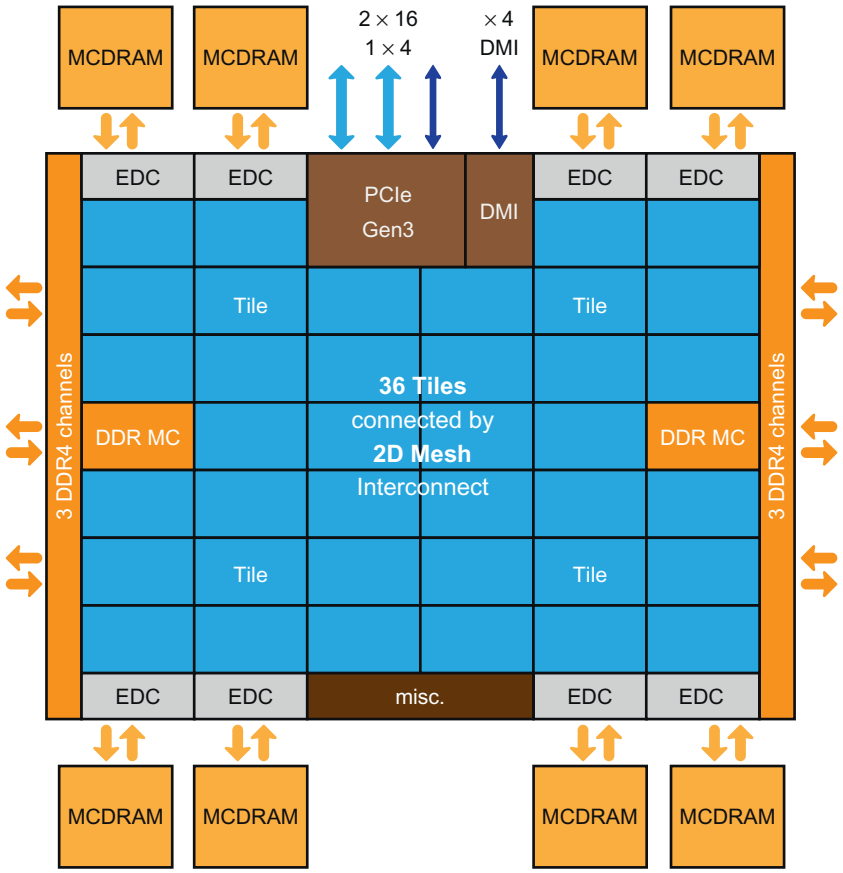


FIG. 2.2

Block diagram showing overview of Knights Landing Architecture.

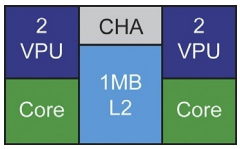


FIG. 2.3

Block diagram of a tile.

per core, deeper out-of-order buffers, higher L1 and L2 cache bandwidths, adding AVX-512 vector instructions, improved reliability features, larger TLBs, and a larger L1 cache. The new core supports all legacy x86 and x86-64 instructions, making Knights Landing completely binary compatible with prior Intel® Architecture processors.

MESH: ON-DIE INTERCONNECT

The cores within a single tile have access to their local L2 cache, and their local processing capabilities, including their local VPUs, all within the single tile. However, in order to access data stored in other L2 caches, memory (MCDRAM or DDR), or perform I/O (PCIe), it is necessary to communicate with other parts of the chip via the on-die interconnect. The tiles are interconnected by a cache-coherent, two-dimensional (2D) *mesh* interconnect. Given its 2D nature, the mesh interconnect provides a more scalable way to connect the tiles by providing higher bandwidth and lower latency compared to the 1D ring interconnect on Knights Corner. The mesh interconnect employs an MESIF cache-coherent protocol — where a cache line can be in (M)odified, (E)xclusive, (S)hared, (I)nvalid, or (F)orward state — to keep the L2 caches in all tiles coherent with each other. The lines present in tile L1 and L2 caches are tracked using a distributed tag directory structure. Each tile holds a portion of this distributed tag directory structure in the box (i.e., hardware design unit) called CHA (caching/home agent). The CHA also serves as the point where the tile connects with the mesh.

The mesh on-die interconnect architecture is based on a ring architecture that has been widely utilized in the current Intel Xeon processor family. The mesh features four parallel networks, each for delivering different types of packets. The mesh has been optimized for the Knights Landing traffic flows and protocols and is capable of delivering greater than 700 GB/s of total aggregate bandwidth.

The mesh is organized into rows and columns of "half" rings that fold upon themselves at the endpoints (e.g., that means that traffic sent off the edge from an edge tile is connected as the input on the same edge of the same tile; this is *not* a torus configuration which would require long connections across the chip resulting in inconsistent delays tile to tile). The mesh enforces a "*YX* routing" rule. This means that a transaction always travels vertically first until it hits the target row, makes a turn, and then travels horizontally until it reaches its destination. Messages arbitrate with the existing traffic on the mesh at injection points as well as when making a turn. The existing traffic on the mesh takes higher priority. The static *YX* routing simplifies the protocol by helping reduce deadlock cases. A single hop on the mesh takes two clocks in the *X*-direction and only one clock in the *Y*-direction.

Cluster modes

The mesh supports three modes of clustered operation, which provide different levels of address affinity to improve overall performance. These are (i) all-to-all mode, (ii) quadrant mode, and (iii) sub-NUMA clustering (SNC) mode. For most systems, quadrant mode will be the default. In some rare configurations, such as when memory capacity is not uniform across all channels, all-to-all mode will be required. Switching from quadrant mode to an SNC mode may boost performance for some applications, generally those using more than one MPI rank on a single Knights Landing. These modes, and the programming implications, are discussed in Chapter 3 with a deeper look in Chapter 4.

MCDRAM (HIGH-BANDWIDTH MEMORY) AND DDR (DDR4)

Knights Landing processor has two types of memory: MCDRAM and DDR. These two types of memory together provide both high bandwidth and large capacity required to support bandwidth hungry applications as a standalone, bootable processor.

MCDRAM is the high-bandwidth memory integrated on-package. There are eight MCDRAM devices integrated, each is of 2 GB capacity, for a total of 16 GB of high-bandwidth memory. These devices are connected to their own memory controller (EDC — named such for historical reasons) via a proprietary on-package I/O (OPIO). Each device has a separate read and write bus connecting it to its EDC. The aggregate Streams Triad bandwidth from all the eight MCDRAMs is over 450 GB/s.

The memory can be configured at boot time in one of the three modes: (i) cache mode — where MCDRAM is a cache for DDR, (ii) flat mode — where MCDRAM is treated like standard memory in the same address space as DDR, and (iii) hybrid mode — where a portion of MCDRAM is cache and the remaining is flat.

The high-bandwidth memory, MCDRAM, is explained in more detail in Chapter 3 along with details on programming including the different use case implications for applications. Much more about the architecture of MCDRAM support and memory modes comes in Chapter 4 including a detailed explanation of its interactions with cluster modes.

DDR offers high-capacity memory that is external to the Knights Landing package. There are two DDR4 memory controllers, on opposite sides of the chip, each controlling three channels. Each channel supports one DIMM per channel for speeds up to 2400 MHz. The total memory capacity will depend on the capacities of the DIMM used, but the maximum total capacity is 384 GB, assuming with 64 GB DIMMs per channel. The aggregate Streams Triad bandwidth from all six DDR4 channels is around 90 GB/s.

I/O (PCIE GEN3)

Knights Landing supports a total of 36 lanes of PCIe Gen3 for I/O, divided into two x16 and one x4 lanes. These lanes are PCIe Root Ports serving as masters on PCIe bus, as opposed to End Points; these are the same as PCIe Root Ports in any other self-hosted processor. The PCIe controller is housed in a box called integrated I/O (IIO) in Fig. 2.2. The IIO also contains four lanes of proprietary direct media interface that is used to connect to the server chipset that provides the platform level legacy functions required to boot a standard operating system.

Knights Landing also integrates the Intel® Omni-Path Fabric on-package in some product versions. The fabric is connected via the 2 x16 lanes of PCIe to the Knights Landing die (leaving x4 lanes for external devices) and provides two 100 Gb/s ports out of the package as shown in Fig. 2.4. The architecture of Intel Omni-Path is the subject of Chapter 5.

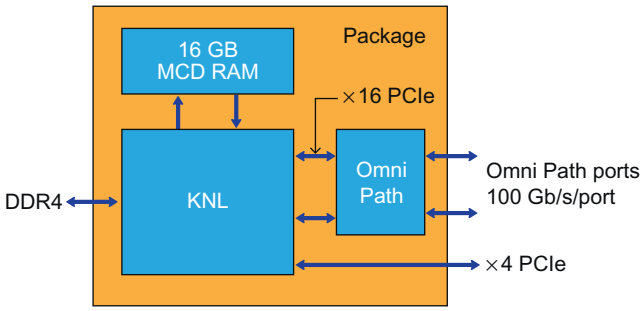


FIG. 2.4

Knights Landing block diagram with Intel Omni-Path Fabric.

MOTIVATION: OUR VISION AND PURPOSE

Having provided an overview, we will share our vision and purpose behind some of the important architectural choices that resulted in the Knights Landing architecture just discussed. Knight Landing started with our desire to remove the overhead of offloading computation from the host processor to a co-processor card connected across the PCIe bus as is present in the first generation (Knights Corner) of Intel Xeon Phi products. The latency and bandwidth of transferring data to and from the coprocessor through a software driver layer, and over a PCIe bus, is a considerable bottleneck. Any performance advantage from faster execution on the card is limited by such transfer overhead for any PCIe compute device. To eliminate transfer overhead, we decided to make Knights Landing a standalone *processor* that can boot off-the-shelf operating systems thereby connecting to the network directly instead of having additional data hops across the PCIe bus. This way, data is not transferred anywhere else for computation; data can stay in main memory while being operated on by the Knights Landing cores. It is no different than any standard processor in that regard; there are many additional benefits to being a host processor.

Once we decided that Knights Landing would be a standalone processor, several other decisions followed from there including the choice of core design. As a standalone processor, Knights Landing would run the entire application, not just its parallel portions. This meant that it needed to be capable of running even single threaded, non-parallel, parts of the application well, while still being significantly more power efficient when executing the parallel portions. This motivated the decision to create a new Knights Landing core that significantly improves single thread performance over the fully parallel-focused Knights Corner — by over 3 × — while maintaining power efficiency when running parallel and vectorized code.

Our second big decision was about the ISA. With Knights Landing as a standalone processor, we needed to maximize support for all types of software including standard operating systems, debuggers, infrastructure management, tools, and legacy libraries that would run on it. Allowing software to run without requiring

recompilation removes a major overhead for usage. This motivated us to make Knights Landing binary compatible with the mainline Intel ISA, enabling all legacy code that run on prior Intel processors to run on Knights Landing as well, without modification and without requiring recompilation.

Our third major decision was on the memory architecture. The memory architecture in Knights Landing was designed to support its large compute capability and its standalone processor status. Being a standalone processor, Knights Landing would run the operating system and all parts of the software stack. This called for a capability to support large memory capacity. Users have requirements for memory capacity numbers such as 2-4 GB per core, and in some cases even higher. To provide enough memory capacity for standard software to run on Knights Landing, we decided to provide DDR memory channels. While DDR memory would provide the required capacity, they could not provide the needed bandwidth to feed the massive compute capability of Knights Landing. For this purpose, we incorporated up to 16 GB of high-bandwidth memory, called MCDRAM, in the Knights Landing package. The MCDRAM can provide over 450 GB/s of bandwidth directly to Knights Landing. Together, the two types of memory provide both large capacity and high bandwidth needed for a high-performance many-core Knights Landing processor.

Finally, we had to decide on the on-die fabric to interconnect all the tiles and other components on the chip. The 2D mesh interconnect was chosen for this purpose. It was designed to provide an on-die network that can easily move in excess of 450 GB/s of bandwidth delivered from the memory across the chip. It also provided lower latency connections, because there are fewer hops required to go from one point to another on the chip. A 1D ring interconnect, as used in Knights Corner, would not have scaled efficiently to the levels of bandwidth provisioned in Knights Landing.

PERFORMANCE

One of the objectives of Knights Landing was to provide a significant boost in performance (and performance per watt) for applications that are optimized to exploit its high compute and bandwidth capabilities, while still providing reasonable good performance for unoptimized code that are run "out-of-box." As users optimize their code on Knights Landing processor, we did not want them to start from a point of significant performance deficit and then have to work hard to merely overcome that deficit before realizing any performance upside from the new processor. Instead, we wanted applications to at least realize "at-par" performance for the unoptimized code on Knights Landing immediately and then reap much higher performance benefits for their code optimization efforts. Fig. 2.5 demonstrates this point with some early performance and performance per watt numbers obtained on pre-production versions of Knights Landing (A0 stepping, the first manufactured parts). The graph shows the performance and relative performance/watt of Knights Landing processor relative to single socket of an Intel Xeon E5-2697-v3 (code named *Haswell*) processor. The workloads are shown on *x*-axis. They range from benchmarks, such as Linpack

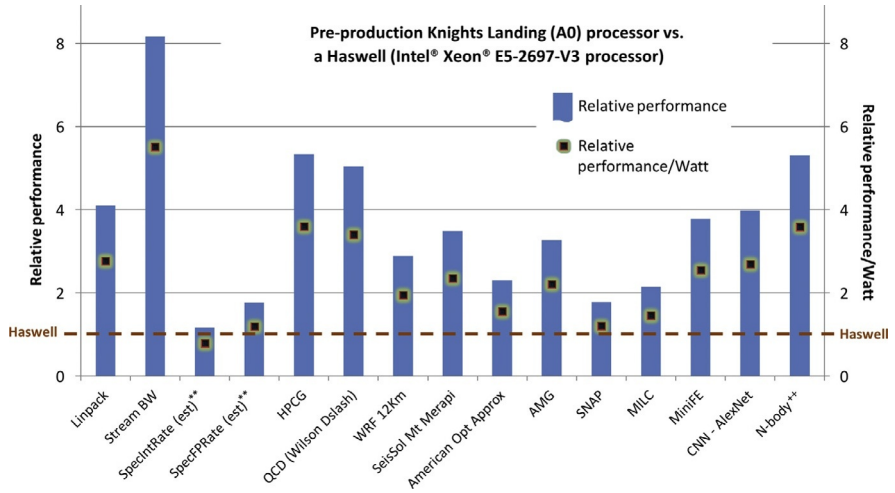


FIG. 2.5

Early performance measurements Knights Landing versus Haswell. Measurements on a pre-production Knights Landing (A0) processor. Results subject to change on production parts.

**Early A0 processor runs, estimated SPEC (not fully compliant nor submitted runs).

++Code and measurement done by colfaxresearch.com.

Data courtesy of Intel Corporation.

and Streams, to various scientific and engineering applications, to general "out-of-box" throughput benchmarks, such as SpecInt Rate and SpecFP Rate. As the early results show, Knights Landing provides at-par performance and performance per watt versus one socket of Haswell on unoptimized, "out-of-box," SpecInt and SpecFP Rate benchmarks (approximately $1.0\times$ to $1.8\times$ on performance and approximately $0.9\times$ to $1.2\times$ on performance per watt) while providing significant improvements (approximately $2\times$ to $8\times$) on other workloads that have varying degrees of optimizations whether compute bound or memory bound.

SUMMARY

In this chapter, we provided an architectural overview of Knights Landing, the second generation of Intel Xeon Phi products. A defining design choice for Knights Landing is that it is architected to be a standalone bootable processor that runs off the shelf operating systems, middleware and applications, putting it on-par with other Intel Architecture family processors. As a highly parallel oriented processor, Knights Landing enables the growing application base of modernized parallel codes to take advantage of its new high-performance features to reach new performance levels to more rapidly advance scientific discovery, big data analytics, machine learning, and

other high-performance needs. The next few chapters will delve more deeply into these high-performance features and explain their benefits and uses in detail. At this point, if your interest is primarily in parallel application development on Knights Landing, you may want to skip right to Section II now, or after reading Chapter 3, and return to Section I for later reference.

FOR MORE INFORMATION

- *Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals*, Intel Corporation, June 2015, https://software.intel.com/isa-extensions.
- *Knights Landing: 2nd Generation Intel® Xeon Phi™ Product*, IEEE Micro Magazine, Hot Chips Special Issue, Volume 36, Issue 2, March–April 2016.
- *Intel® Architecture Instruction Set Extensions Programming Reference*, Intel Corporation, August 2015, https://software.intel.com/isa-extensions.

CHAPTER

Programming MCDRAM and Cluster modes

3

Knights Landing offers an unprecedented variety of configurations which have traditionally been available only as hardwired and unchangeable design decisions. Specifically, the choices realized by the cluster modes and the memory modes. This wide ranging support allows Knights Landing to act like very different machines based on the configuration used to initialize Knights Landing, the operating system, and then the applications.

What is new with Knights Landing in this chapter?

This entire chapter is about application usage of MCDRAM and cluster modes, which are new with Knights Landing.

Fortunately, these different modes match to different programming styles that are already popular. In that sense, Knights Landing can take on a personality to match an application instead of the other way around. Of course, applications can be modified to take advantage of this variety as well. We will describe these features from both vantage points.

Memory types (MCDRAM and DDR), memory modes (cache, flat, hybrid), and cluster modes (All-to-All, Quadrant, Hemisphere, SNC-2, and SNC-4) were briefly introduced in Chapter 2. In this chapter, we cover the essentials of programming to utilize the high-bandwidth memory known as the MCDRAM and to utilize cluster modes. Chapter 4 dives much deeper into the hardware architecture; this chapter is all about the programming interfaces and implications for applications.

The usage models available are dependent upon which memory and cluster modes were set at boot time. While there is nothing an application can do directly to control which memory mode or which cluster mode is selected (aside from changing a BIOS setting and rebooting the system), there are numerous usage choices remaining to discuss.

MPI+X (e.g., MPI+OpenMP) applications may run faster with *cluster mode = SNC-4* instead of the default *cluster mode = quadrant*. Cache friendly applications will benefit from the default of *memory mode = cache*. Beating the performance of cache mode is not easy for most applications. However, an important class of applications is known to be not cache-friendly, and those will likely benefit the most from other memory modes. Because most applications have found ways to be relatively cache friendly, we could guess that most applications will run well enough with the defaults of *cluster mode = quadrant* and *memory mode = cache*, and we would probably be right. However, those are "fighting words" to some. Such

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00003-X

questions are certainly open to debate by those intent on squeezing every last bit of performance out of Knights Landing. Fortunately, the debate can continue because these decisions were not hardwired into the design.

USE CACHE MODE AND DEFAULT CLUSTER MODE (AT FIRST)

Before we explain memory modes and cluster modes, we have a simple point: initially most applications can gain more from fine tuning for parallelism than from fiddling with memory and cluster modes. The default cluster mode of quadrant and default memory mode of cache will likely work well for most applications. We emphasize this because covering MCDRAM programming so early in the book as Chapter 3, in Section I of this book, was done to group it with things new or unique in Knights Landing. Being so early in the book is not a statement about it being more important than tuning for general parallel programming, which is covered in Section II of this book.

Now that we have that out of the way, we do want to introduce and explain memory modes and cluster modes from a programmer's point of view. Chapters 4 and 6 supply additional information for those inclined to understand the underlying hardware in greater detail. This chapter focuses on the key interfaces for programmers to build a firm foundation for using memory and cluster modes.

PROGRAMMING FOR CLUSTER MODES

We are going to say very few words about cluster modes in this chapter because they are fairly simple to explain. It is very important to know that, regardless of the cluster mode selected, all of memory, DDR and MCDRAM, is always available to every core. Also, all memory is fully cache coherent. This means that every core has the same view of what data is in every location. What differs between the modes is whether the view of a particular memory (MCDRAM or DDR) from the cores is UMA (Uniform Memory Access) or NUMA (Non-Uniform Memory Access). In any case, even if MCDRAM and DDR are each used as UMA memory, the combination is NUMA.

There are five cluster modes, which are often thought of as three modes with two variations: all-to-all, quadrant (variation: hemisphere), and SNC-4 (variation: SNC-2).

The default cluster mode is quadrant, unless you have a machine where the DDR is not evenly populated. If the DDR DIMMs connected to the Knights Landing are not equal in capacity, then the only cluster mode available is the all-to-all mode. When the DDR DIMMs are equally populated, we will likely choose to not run in all-to-all mode. The cluster modes, other than all-to-all, optimize internal tables within Knights Landing assuming evenly distributed DDR DIMMs. Programs written for all-to-all, quadrant, and hemisphere are unlikely to be different. Most programmers will simply consider these modes to be the same from an application standpoint.

Therefore, for programming—the key issues arise from all-to-all/quadrant/hemisphere versus SNC-4/SNC-2 cluster modes. The key difference is whether each memory type (MCDRAM and DDR) is UMA or NUMA.

In quadrant mode, each memory type is UMA meaning that the latency from any given core to any memory location within the same memory type (MCDRAM or DDR) is essentially the same. In a strictly technical sense, the actual latencies will vary a little across the mesh. However, we cannot distinguish between memory regions by different latencies. While nothing is completely uniform in a strictly technical sense, we program to it as UMA because we cannot deterministically change the effective latency solely through the choice of memory locations.

In SNC-4, each memory type is NUMA meaning that the cores and memory are divided into (four) quadrants with lower latency for "near" memory accesses (within the same quadrant) and higher latency for "far" (within a different quadrant) memory accesses. SNC-4 is well suited for MPI applications that utilize four, or a multiple of four, ranks per Knights Landing. If an application is not NUMA aware, and is not divided into *multiple* MPI ranks, then quadrant mode is almost certainly the right choice. For instance, an application written with one MPI rank per node, which uses OpenMP for all the cores on the node, may not be the best fit for SNC. Such a case is at odds with the purpose of SNC, to isolate and localize traffic to distinct lower latency memory partitions.

The other two cluster modes, that are often called out as variations on *quadrant* and *SNC-4*, are *hemisphere* and *SNC-2*. These two modes are identical to quadrant and SNC-4, except they logically divide the cores and memory (by types) into halves instead of quarters. Compared with their corresponding four-way modes, these two-way modes have higher latency that leads to reduced bandwidth as a result. They may be a better fit for the way a particular application is structured, or if the number of active tiles in Knights Landing does not divide by four. For example, a 68-core Knights Landing has 34 tiles (two cores per tile) that divide into halves as 34 and 34 cores, but into quarters as 16, 16, 18, and 18 cores, because tiles do not divide. In such configurations, SNC-2 may be a better choice for a balanced MPI program running multiple ranks per Knights Landing.

PROGRAMMING FOR MEMORY MODES

Knights Landing features high-bandwidth on-package memory known as MCDRAM. This on-package memory is not faster for a single data access than main memory (DDR), but it can support much high bandwidth (more simultaneous data accesses) than main memory. Therefore, when used as a cache, or used directly to store the data most used within an application, substantial speed-ups are possible.

All of the MCDRAM can be used as program allocatable memory in what is known as *flat mode* on Knights Landing. In flat mode, the entirety of DDR space and MCDRAM space is visible to the operating system and applications. It is exposed as separate NUMA nodes.

Flat mode is always available to choose, and it is the only option available on the Knights Landing coprocessor (see Chapter 18) since Knights Landing coprocessors have only MCDRAM and no DDR support. Most often a system, using Knights Landing processors, also has a main memory consisting of DDR attached to a Knights

Landing in addition to the MCDRAM on-package. In cases where there is both MCDRAM *and* DDR memories, all or some of the MCDRAM (25%, 50%, or 100%) can be used as a cache for data in the DDR. In this case, some or all of the MCDRAM acts as a memory-side cache. When 100% of the MCDRAM is used as cache, we call this *cache mode*. In the case where only 25% or 50% of the MCDRAM is used as a cache, we call this *hybrid mode* because the remaining MCDRAM is available as memory as in flat mode but smaller in size. Summaries of these modes are shown in Fig. 3.1. These modes are set at boot time based upon the current BIOS settings. Changes to these BIOS settings, depending on the BIOS used, may be possible through interactive menus at boot time, or tools after boot time, or both.

While memory modes and cluster modes are largely orthogonal, concepts of "near" and "far" come into play. The cluster modes called SNC-4 and SNC-2 can be used to effectively subdivide Knight Landing into smaller grouping of cores (four parts, or two parts, respectively. They also divide the MCDRAM and DDR to align with the grouping of cores. In order to get the full value of these SNC modes, applications should use "near" memory (MCDRAM or DDR) instead of using all of the memories ("near" and "far"). When we focus on "near" only, the size of MCDRAM and DDR can be half or a quarter of the otherwise expected "flat" or "hybrid" memory sizes. This division of MCDRAM is discussed in depth in Chapter 4 in a section titled *Interactions of Cluster and Memory Modes*. Our programming advice is simple: if we choose to utilize MCDRAM directly, we should parameterize our applications to be able to deal with differing sizes of MCDRAM, and avoid naming NUMA nodes by number explicitly (see a later section *How to Not Hard Code the NUMA Node Numbers*).

Memory (MCDRAM and DDR) do not change sizes under SNC-4 and SNC-4 cluster modes versus the other cluster modes. However, the benefit of using SNC cluster modes depends in part on focusing data usage to be in "near" memories. The SNC cluster modes have "near" memory because they affinitize a portion of each memory type to the quarter (SNC-4) or half (SNC-2) of the Knight Landing tiles.

Memory Mode	MCDRAM (HBM)	DDR (DDR4)
Cache	100% caches (caches DDR data)	Memory (*DDR is required in this mode*)
Flat	All used as memory (0% as cache)	Memory (*if present*)
Hybrid	Partitioned as cache (25% or 50%) and memory (75% or 50%)	Memory (*DDR is required in this mode*)

FIG. 3.1

Summary of memory modes.

MEMORY USAGE MODELS

Applications generally will do nothing special when running in cache mode. Of course, applications which maintain a working set to match the MCDRAM size (16 GB) or otherwise block their data usage to maximize reuse knowing the MCDRAM size will benefit from better caching in a manner similar to what you would expect from any cache. In practice, for high-performance applications the cache mode behavior is generally quite good and offers the majority of performance boost available from MCDRAM. Nevertheless, for applications with larger data sets, the non-cache memory modes do offer some performance upside when used. Applications with "cache unfriendly" data are candidates for using memory modes other than cache.

When using some or all of the MCDRAM as memory (flat or hybrid mode), we have summarized the options in Fig. 3.2.

Option	Summary
Do nothing	A program that is well blocked into L2, may be happy with DDR alone. Some experiments to verify may be wise. Cache-friendly applications may still benefit from being in cache mode. Programs which do poorly with caches could conceivably benefit from putting MCDRAM in flat mode and leaving unused.
Cache mode	Trivial to try; no source code changes. Knight Landing can dedicate 100%, 50% or 25% of the MCDRAM as a memory-side cache.
Remaining options are only useful *flat* and *hybrid* memory modes to access the MCDRAM as memory.	
numactl (NUMA Control utility)	No code changes: use numactl to have all data allocations (including the data segments and stack) come from MCDRAM. This works regardless of programming language.
autohbw (comes with memkind package)	No code changes: use the autohbw library (part of the memkind project) to have allocations, of a certain size range, come from MCDRAM. This definitely works for Fortran, C and C++ because their allocation routines ultimately utilize allocation routines associated the standard C library, e.g. glibc. This can work for all programming languages to the extent that their allocations routines also rest on top of the standard C library routines; autohbw could be extended to intercept most any allocation routine. Like memkind, data segments and stacks are not allocated from MCDRAM.
memkind (includes hbw_ routines)	Change code: use the memkind library, or the FASTMEM directives in Intel Fortran, to have specific allocations be mapped to MCDRAM. Applications in any language can find ways to use these explicit APIs. In a later Section, *Approaches to Determining What to Put in MCDRAM*, we discuss some approaches to figuring out what to allocate using memkind. These approaches could guide autohbw usage as well.

FIG. 3.2

Options for using MCDRAM, or *not*!.

WHAT IS THE MEMKIND LIBRARY (AND HBWMALLOC)?

Because we are going to start mentioning the memkind library, we need to explain it now. The memkind library is a user extensible heap manager, built on top of jemalloc, which enables control of memory characteristics and a partitioning of the heap between kinds of memory. This combination of jemalloc and memkind starts to address a long-standing software need; there has simply been no good solution for managing multiple kinds of memory (e.g., DRAM, MMIO, RDMA slabs, symmetric heaps) in an operating system and runtime agnostic way. The advocates for the memkind project believe they have helped move us all forward.

The memkind library delivers two interfaces defined in the header files hbwmalloc.h and memkind.h. The developers themselves describe hbwmalloc as the more stable of the two and the API recommended for high-bandwidth memory (MCDRAM) use cases. They describe memkind as more of a work in progress to develop a very generic API for a broad range of memory types. The hbwmalloc API is built on top of memkind as a convenience for users of high-bandwidth memories.

The hbwmalloc APIs are specifically for high-bandwidth memory; the memkind APIs are a more generic interface that can support additional memory types in the future, as well as explicit DDR allocations.

The advantage of the hbwmalloc interfaces is that they minimize the code changes needed—just add a prefix of "hbw_" for popular heap allocation library calls (i.e.,malloc(), calloc(), realloc(), posix_memalign(), and free()). The advantage of the memkind interfaces is that they are more flexible for additional memory types. The memkind library is also extensible to support user-defined memory types as well.

Linux backs virtual addresses with 4 KB physical pages by default, though some distributions now have enabled transparent huge pages that can give you automatic 2M page allocation. The memkind library supports explicit allocation of large pages by allowing an application to select which data structures should be allocated with large pages; see the function hbw_posix_memalign_psize for more details.

Both interfaces should be considered part of a memkind "convenience library." All this functionality is built upon existing memory allocation APIs, albeit with a little more work. Since memkind is open-source, it is also possible to customize as needed, or simply adopt the techniques for use in custom memory allocators.

MAXIMIZING PERFORMANCE WITH MEMORY USAGE MODELS

We wrote a very simple application to repeatedly read and write memory, from 64 cores in parallel, and we call it *Hello MCDRAM*. The key routine is shown in Fig. 3.3; the full source code is available from our website—see *For More Information* at the

```
#define CACHE 16
void exercisememory(int *start,
                    int *endplus1,
                    unsigned count) {
  unsigned i;
  int s, *a0=start, *a;
  int mythreadid = omp_get_thread_num();
  for (a=a0;a<(endplus1+CACHE*4);*a++=0);
  a=a0;
  for (i=0;i<count*3;i++) {
    s = *a;
    s -= ++*(a+=CACHE);
    s += --*(a+=CACHE);
    s -= ++*(a+=CACHE);
    if (a >= (endplus1-CACHE*4)) a = a0;
    /* dear compiler: we needed to compute s */
    start[0] += s;
  }
}
```

FIG. 3.3

The key to Hello MCDRAM—the exercising of a consistent number of memory locations (count is always the same), but varying run-to-run the range of memory touched (working set—determined by endplus1-start).

end of this chapter. No single application can possibly be representative of every workload, and our *Hello MCDRAM* is no exception. Our toy application is very deliberate at focusing on a single chunk of memory to exercise and it is a short running application. Nevertheless, it helps illustrate the challenges that we face if we try to do better through completely explicit usage of MCDRAM versus a number of other options available to us without modifying our application.

Our *Hello MCDRAM* application always does the same number of memory accesses in any run of the application, but it varies how large the section of memory is that it exercises (ranging from only 4 bytes, to 4 MB of memory, touched per core). In our graphs, we show how many bytes per core are exercised by the application. We then compare *memory mode = cache* with four ways to utilize MCDRAM in flat mode by recording the amount of time for the application to complete. Fig. 3.3 shows very clearly that anything is better than letting the MCDRAM go unused. The performance of our toy application is over 5 × worse for larger data sets than any other option. Cache mode sets a pretty high bar for performance. Measuring performance in "flat mode," but not using MCDRAM at all as we have effectively done in Fig. 3.3, helps give a perspective of the value of MCDRAM. If use of cache mode had not lifted performance significantly, we could conclude that prescriptive uses of MCDRAM (such as autohbw or memkind/FASTMEM) may be able to perform better. In pursuit of performance, MCDRAM should not be left unused. Any application that makes a choice to manage the MCDRAM explicitly is competing with the other usage models including the cache mode.

Cache mode sets a high bar in terms of performance. The true baseline is not using MCDRAM at all; measuring performance in "flat mode" but not using MCDRAM at all gives the true baseline. In pursuit of performance, MCDRAM should not be left unused. If use of cache mode did not lift performance significantly, that may signal opportunities for more prescriptive uses of MCDRAM. Most often, any application which makes a choice to manage the MCDRAM explicitly faces tough competition from the cache mode to reach better performance. Fig. 3.3 shows very clearly that bandwidth hungry applications should not let the MCDRAM go unused.

The effects of 16K L1 caches per core, and 1M L2 caches shared between pairs of cores (giving ~512K/core effectively for our application), are evident in Fig. 3.4. After the working sets exceed the L2 caches, the effects of the MCDRAM govern all the options. The "unused flat" line helps us see that the effects of MCDRAM do indeed

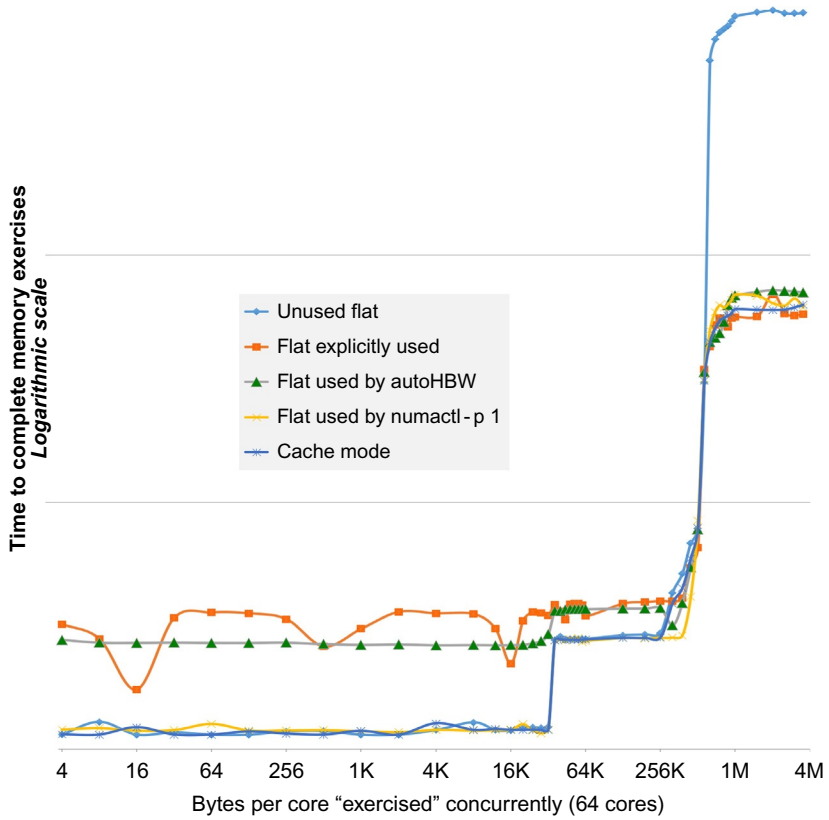


FIG. 3.4

The worst way to use MCDRAM: ignore it (see line marked *unused flat*). The best way to use MCDRAM: we have four choices to match a particular application needs. Three of them need no changes to an application.

consistently and profoundly affect performance for the better, note that the graph is logarithmic on the scales, the difference is very substantial!

In Fig. 3.4, we also see that the less prescriptive methods (cache mode, and using numactl to move all allocations to MCDRAM) yield noticeably better results than the more explicit methods (autohbw and explicit memkind/FASTMEM usage) in which we are prescribing to the system which allocations to place in MCDRAM. The lesson is clear: if explicit usage is going to leave much of the MCDRAM idle, other methods that fully utilize the MCDRAM, even for seemingly less important uses, will very often offer the best performance.

Choosing to explicitly manage the MCDRAM in an application is unlikely to offer the best performance if any significant portion of the MCDRAM is underutilized.

Critical review for our Hello MCDRAM

Our "Hello MCDRAM" program is both relatively cache-friendly and bandwidth hungry. Therefore, Fig. 3.4 reflects those effects. There are some applications, perhaps small in number but important, that are known to be cache-unfriendly. Such applications are much more likely to be rewarded by using the flat memory mode instead of the cache memory mode. Also, there are applications that block into L2 with such high reuse that MCDRAM is not needed. Unless such application would benefit from blocking into the larger capacity of MCDRAM, the existence of MCDRAM may not matter.

numactl -H; Learning NUMA Node Numbering

A key command to know is "numactl -H." We have a few sample outputs, to illustrate actual usage, from a 64-core Knights Landing system, with 16 GB of MCDRAM memory and 64 GB of DDR memory. Fig. 3.5 shows the output when booted in cluster mode = quadrant and memory mode = cache. In such a case, there is only one NUMA memory node, numbered zero (0). The output of "numactl -H" for cluster modes *all-to-all* and *hemisphere* is identical to those for *quadrant* because the division of memory and MCDRAM is always identical.

Fig. 3.6 shows the two NUMA nodes created for cluster mode = *quadrant* with memory mode = *flat*. In this case, NUMA node one (1) is the MCDRAM. Since the NUMA nodes would be identical, the output shown in Fig. 3.6 is also exactly the output when using all-to-all and hemisphere cluster modes with the same memory mode. The DDR node is listed before MCDRAM because the distance to MCDRAM is purposefully higher to avoid making it become the default for all allocations by the operating system. The distances are discussed in detail in Chapter 4.

Fig. 3.7 shows the most complex example, which is cluster mode = *SNC-4* and memory mode = *flat*. In this case, there are eight NUMA nodes created. Note that available DDR and MCDRAM are divided into multiple NUMA nodes when dividing Knights Landing up using a NUMA cluster mode (i.e., SNC-4 or SNC-2). The

```
     available: 1 nodes (0)
     node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86
87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102
103 104 105 106 107 108 109 110 111 112 113 114 115
116 117 118 119 120 121 122 123 124 125 126 127 128
129 130 131 132 133 134 135 136 137 138 139 140 141
142 143 144 145 146 147 148 149 150 151 152 153 154
155 156 157 158 159 160 161 162 163 164 165 166 167
168 169 170 171 172 173 174 175 176 177 178 179 180
181 182 183 184 185 186 187 188 189 190 191 192 193
194 195 196 197 198 199 200 201 202 203 204 205 206
207 208 209 210 211 212 213 214 215 216 217 218 219
220 221 222 223 224 225 226 227 228 229 230 231 232
233 234 235 236 237 238 239 240 241 242 243 244 245
246 247 248 249 250 251 252 253 254 255
     node 0 size: 65432 MB
     node 0 free: 62887 MB
     node distances:
     node   0
       0:  10
```

FIG. 3.5

numactl output for system in Quadrant+Cache modes.

```
     available: 2 nodes (0-1)
     node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86
87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102
103 104 105 106 107 108 109 110 111 112 113 114 115
116 117 118 119 120 121 122 123 124 125 126 127 128
129 130 131 132 133 134 135 136 137 138 139 140 141
142 143 144 145 146 147 148 149 150 151 152 153 154
155 156 157 158 159 160 161 162 163 164 165 166 167
168 169 170 171 172 173 174 175 176 177 178 179 180
181 182 183 184 185 186 187 188 189 190 191 192 193
194 195 196 197 198 199 200 201 202 203 204 205 206
207 208 209 210 211 212 213 214 215 216 217 218 219
220 221 222 223 224 225 226 227 228 229 230 231 232
233 234 235 236 237 238 239 240 241 242 243 244 245
246 247 248 249 250 251 252 253 254 255
     node 0 size: 65432 MB
     node 0 free: 60405 MB
     node 1 cpus:
     node 1 size: 16384 MB
     node 1 free: 15917 MB
     node distances:
     node   0   1
       0:  10  31
       1:  31  10
```

FIG. 3.6

numactl output for system in Quadrant+Flat modes.

```
     available: 8 nodes (0-7)
     node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
15 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79
128 129 130 131 132 133 134 135 136 137 138 139 140
141 142 143 192 193 194 195 196 197 198 199 200 201
202 203 204 205 206 207
     node 0 size: 16280 MB
     node 0 free: 15413 MB
     node 1 cpus: 16 17 18 19 20 21 22 23 24 25 26 27
28 29 30 31 80 81 82 83 84 85 86 87 88 89 90 91 92 93
94 95 144 145 146 147 148 149 150 151 152 153 154 155
156 157 158 159 208 209 210 211 212 213 214 215 216
217 218 219 220 221 222 223
     node 1 size: 16384 MB
     node 1 free: 15818 MB
     node 2 cpus: 32 33 34 35 36 37 38 39 40 41 42 43
44 45 46 47 96 97 98 99 100 101 102 103 104 105 106
107 108 109 110 111 160 161 162 163 164 165 166 167
168 169 170 171 172 173 174 175 224 225 226 227 228
229 230 231 232 233 234 235 236 237 238 239
     node 2 size: 16384 MB
     node 2 free: 15617 MB
     node 3 cpus: 48 49 50 51 52 53 54 55 56 57 58 59
60 61 62 63 112 113 114 115 116 117 118 119 120 121
122 123 124 125 126 127 176 177 178 179 180 181 182
183 184 185 186 187 188 189 190 191 240 241 242 243
244 245 246 247 248 249 250 251 252 253 254 255
     node 3 size: 16384 MB
     node 3 free: 15886 MB
     node 4 cpus:
     node 4 size: 4096 MB
     node 4 free: 3983 MB
     node 5 cpus:
     node 5 size: 4096 MB
     node 5 free: 3982 MB
     node 6 cpus:
     node 6 size: 4096 MB
     node 6 free: 3982 MB
     node 7 cpus:
     node 7 size: 4096 MB
     node 7 free: 3979 MB
     node distances:
     node   0   1   2   3   4   5   6   7
       0:  10  21  21  21  31  41  41  41
       1:  21  10  21  21  41  31  41  41
       2:  21  21  10  21  41  41  31  41
       3:  21  21  21  10  41  41  41  31
       4:  31  41  41  41  10  41  41  41
       5:  41  31  41  41  41  10  41  41
       6:  41  41  31  41  41  41  10  41
       7:  41  41  41  31  41  41  41  10
```

FIG. 3.7

numactl output for system in SNC-4 + Flat modes.

DDR nodes are listed first, and the MCDRAM nodes are listed last. The distances reflect the affinitization of DDR and MCDRAM to the divisions of Knights Landing in this mode. No matter which core you run on, with SNC modes there is a "near" MCDRAM and a "near" DDR. With SNC-4, there are also three (one on SNC-2) "far" MCDRAM nodes and three (one on SNC-2) "far" DDR. Because the number of NUMA nodes change, the actual "node" numbers for MCDRAM are not consistent between SNC modes or the other non-SNC mode. We have a later section *How to Not Hard Code the NUMA Node Numbers* to address how to stay independent of node numbering.

If you find "numactl -H" captivating, you really need to look at `/proc/<pid>/numa_maps`!

WAYS TO OBSERVE MCDRAM ALLOCATIONS

If looking at numactl -H output fascinates you, we encourage you to look at `/proc/<pid>/numa_maps`. These files contain information about each memory area used by a given process. This allows us to determine which NUMA nodes were used for the pages while running. Knowing about numa_map files can be a useful for debugging use of NUMA nodes and huge pages, or just verifying we have set up our program correctly. We do need to remember that allocations occur on first touch, and the NUMA maps will reflect this.

A read operation on numa_maps file will cause the kernel to scan the memory area of a process to determine how memory is used. One line is displayed for each memory area of the process. The numa_maps includes information about the memory policy currently in effect for particular memory areas as well.

Other ways to observe details about allocations include:

- numastat -m (includes huge page info too!)
- numastat -p *{pid,exec_name}*
- *cool idea:* watch -n 1 -p numastat *exec_name*
- cat/sys/devices/system/node/node*/meminfo
- cat/proc/meminfo
- *memkind_install_dir*/bin/memkind-hbw-nodes
- *numactl -hardware*
- lscpu

NUMACTL: MOVE ALL ALLOCATIONS TO MCDRAM

When we can size our application to fit all or most data within the MCDRAM, we can leave an application relatively unmodified but benefit from data being placed into MCDRAM. In an extreme case, if the entire application and all its data fit in MCDRAM, there would be no performance benefit in changing the application to do anything explicit with memory. The "numactl" (NUMA ConTroL) utility can be installed easily on Linux systems with commands such as "yum install numactl."

In order to run an application, with all the allocations, including the stack, going to MCDRAM, we need to use the "-m" option with the NUMA node(s) specified after the option. For quadrant/flat, we would use (see Fig. 3.5 to understand the node number "1"):

```
numactl -m 1 myprogram
```

For SNC-4/flat, we could use (see Fig. 3.6 to understand the node numbers "4 through 7"):

```
numactl -m 4-7 myprogram
```

In the SNC-4 case, we have not taken advantage of "near" versus "far" MCDRAM by lumping them all together as "4-7." In this case, allocations would first come from the "near" MCDRAM and then the other MCDRAM sections. Since `numactl` operates at a program level, there is no simple way to say "use near MCDRAM and then DDR" because the node number for "near" differs from core to core.

In both cases, we suggest always using "-p" instead of "-m" because it states a preference for MCDRAM instead of a requirement. If MCDRAM is exhausted, the application will still run if "-p" is used because it can also allocate from regular memory (DDR).

Combining these thoughts, it is not pretty, but we can do something like this:

```
mpirun -perhost 16 \
numactl -p 4 a.out : \
numactl -p 5 a.out : \
numactl -p 6 a.out : \
numactl -p 7 a.out ...
```

It is reasonable to assume that special support in each MPI implementation will offer this capability since this style of `numactl` command would not be a practical way to invoke thousands of nodes! The Intel MPI product team is planning support that will be controlled by an environment variable `I_MPI_NUMA_POLICY`. Consulting MPI documentation, looking for such support, is highly recommended to get the best behavior while running in SNC cluster modes.

OVERSUBCRIPTION OF MCDRAM: A KILLER OR AN OPPORTUNITY?

A simple "`grabmemory`" application, shown in Fig. 3.8, repeatedly calls `malloc()` for as many gigabytes of data as we specify. Fig. 3.9 shows that we can use "`numactl -p 1 grabmemory 50 touch`" to have our application allocate and use (touch) 50 GB of memory. However, if we require MCDRAM by using "`numactl -m1 grabmemory 50 touch`," as shown in Fig. 3.10, we see the application is aborted by the operating system. Note that the `malloc()` was successful, and the application did not print the failure message it was designed to print. This is because a memory allocation does not fail until a memory page (a page is a "chunk" of memory, generally 4 KB in size but larger sizes exist too) within the allocation is used and the required memory is not actually available. Such failures cannot be trapped by the application; the application must be aborted. In our example, that happens when the `memset()` call is made. Fig. 3.11 makes this very clear by showing that "`numactl -m 1 grabmemory 100`" is able to allocate 100 GB of MCDRAM even though our system has only 16 GB of MCDRAM. Since we did not actually touch that memory, it is allocated but using too much will definitely cause the application to be aborted. Fig. 3.9 serves as a warning that actually ensuring use of MCDRAM is *not* a matter of first-come-first-served as measured by calling `malloc()`. If an application allocates more space than can fit in MCDRAM, and does it with a preference for MCDRAM and not a requirement, the first usage grants the MCDRAM while

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SZALLOC (1024*1024*1024)
#define MAXTRY 25000
/* usage: grabmemory <#gigabytes> [touch] */
int main(int argc, char *argv[])
{
  int *ptr[MAXTRY];
  int i, maxtry=20, touch = (argc > 2);
  if (argc > 1) maxtry = atoi(argv[1]);
  if (maxtry > MAXTRY) maxtry = MAXTRY;
  for (i=0;i<maxtry;i++) {
    ptr[i]=malloc(SZALLOC);
    if (!ptr[i]) {
      printf("Failure after %dGB were "
          "successfully allocated.\n",i);
      return 1;
    }
    printf("Allocated        %dGB        ",i+1);
fflush(stdout);
    if (argc>2) {
      memset(ptr[i],i&255,SZALLOC);
      printf("and touched successfully.\n");
    } else
      printf("no touch\n");
    fflush(stdout);
  }

  printf("Allocated %s%dGB without problems.\n",
      (argc>2) ? "and touched " : "",
      maxtry);
  return 0;
}
```

FIG. 3.8

Application "grabmemory.c" to illustrate effects of numactl.

```
Allocated 1GB and touched successfully.
Allocated 2GB and touched successfully.
… output lines ommitted for space …
Allocated 49GB and touched successfully.
Allocated 50GB and touched successfully.
Allocated and touched 50GB without problems.
```

FIG. 3.9

Output of "numactl -p 1 grabmemory 50 touch." When usage exceeds MCDRAM capacity, pages will be assigned from regular memory when assignments have filled MCDRAM.

```
Allocated 1GB and touched successfully.
Allocated 2GB and touched successfully.
… output lines ommitted for space …
Allocated 14GB and touched successfully.
Allocated 15GB and touched successfully.
Allocated 16GB Killed
```

FIG. 3.10

Output of "numactl -m 1 grabmemory 50 touch." The application is KILLED by the operating system when it tried to *use* too much.

```
Allocated 1GB no touch
Allocated 2GB no touch
… output lines ommitted for space …
Allocated 98GB no touch
Allocated 99GB no touch
Allocated 100GB without problems.
```

FIG. 3.11

Output of "numactl -m 1 grabmemory 100."The allocations far exceed the size of MCDRAM, but the application did not fail because it did not *use* too much.

it is still available. Therefore, oversubscribing of MCDRAM may be a technique to consider for some applications.

Applications that request too much MCDRAM do not fail at `malloc()` time. Such applications are aborted when the number of "touched" memory pages, which are marked as "MCDRAM required," exceed the available MCDRAM. This also means that actually getting MCDRAM is NOT a matter of first-come-first-served as measured by calling malloc(). The allocation on usage can be a surprise if we were not expecting it. It also offers an opportunity to oversubscribe MCDRAM, via allocations with a preference for MCDRAMs, and defer to actual first usage to grant the MCDRAM, if available, or fall back to regular memory, if not.

AUTOHBW: MOVE SELECTED ALLOCATIONS TO MCDRAM

The autohbw library is an interposer library that redirects an application's use of standard heap allocators (i.e., `malloc()`, `calloc()`, `realloc()`, `posix_memalign()`, and `free()`) to utilize MCDRAM. The selection is done purely based on size for specific allocations within our application. Fig. 3.12 shows how to run an application such that allocations, between 250K and 6M in size, will try to allocate from MCDRAM.

Our example assumes we are in a non-SNC mode (e.g., quadrant) with a memory mode other than cache mode. We make this assumption and hard code NUMA node 1 in the setting. This is a bad idea for anything meant to last; we encourage you to learn more later in this chapter in the section *How to Not Hard Code the NUMA Node Numbers*.

When MCDRAM runs out, the application will allocate from regular memory. All allocation smaller than 250K in size, or larger than 6M in size, will not be diverted for special allocations from MCDRAM. In order to use with applications using MPI, we would need to put the LD_PRELOAD in a shell script and then invoke the `mpirun` command on the shell script.

```
export MEMKIND_HBW_NODES=1
export AUTO_HBW_SIZE=250K:6M
LD_PRELOAD=libautohbw.so:libmemkind.so myapplication
```

FIG. 3.12

Example of using autohbw.

In order for `autohbw` to work, the application must be dynamically linked with the memory allocation library that it normally uses. This is default behavior normally, so using `autohbw` commonly does not require worrying about this.

Also, the `autohbw` and `memkind` libraries need to be on the library path (LD_LIBRARY_PATH). Applications using deprecated `valloc()` or `memalign()` cannot be supported and will be aborted if those functions are called (a message will be printed suggesting the use of `posix_memalign()` instead).

The `autohbw` library is open-source, like `memkind`, so it is possible to contribute by extending for additional memory allocation routines if a special need arises beyond the `malloc()`, `calloc()`, `realloc()`, `posix_memalign()`, and `free()` routines.

MEMKIND/FASTMEM: EXPLICIT USAGE OF MCDRAM

Explicit memory usage in C/C++: memkind

When faced with an application with data needs in excess of the size of MCDRAM, we can explicitly choose which data is allocated into MCDRAM. We do this with a set of language constructs or library calls to direct allocations to be from the MCDRAM part of memory. Later in this chapter, we will discuss some tools and techniques to help determine what data will benefit most from using MCDRAM. For now, we will show how to explicitly allocate from MCDRAM assuming we know what we want.

C programmers have MCDRAM versions of the standard heap allocation routines via the functions `hbw_malloc()`, `hbw_calloc()`, `hbw_realloc()`, `hbw_free()`, `hbw_posix_memalign()`, and `hbw_posix_memalign_psize()`. Each of these functions mimic the behavior of the standard routines (with the same names less the "`hbw_`" prefix).

The default policy is for these routines to "prefer" MCDRAM and not require it. This means that when the allocated pages are eventually used, they will be assigned to pages from MCDRAM. In a prior section on numactl, we demonstrated the difference between allocations and page allocations on usage. The same applies for `memkind`, and the allocation does not fail based on MCDRAM availability. It is possible to set a different policy using `hbw_set_policy(HBW_POLICY_BIND)`. That can be done once in an application before allocations are done. Attempts to set policy more than once will fail.

C++ notes

The memkind library ships with some excellent examples, one of which shows how to override `new` to create types that allocate into MCDRAM. Look in the examples directory for memkind_allocated_example.cpp and memkind_allocated_example.hpp.

Explicit memory usage in Fortran: FASTMEM

Fortran programmers can redirect `ALLOCATE()` to use MCDRAM. A summary of the extensions to support this is shown in Fig. 3.13.

Feature	Description
!dir$ ATTRIBUTES FASTMEM *allocatable-variable/ allocatable-field*	Variable or field name tagged so that ALLOCATE for it will be from MCDRAM subject to the current FASTMEM policy.
!dir$ FASTMEM	Lexically next ALLOCATE directed to be from MCDRAM subject to the current FASTMEM policy.
!dir$ NOFASTMEM	Lexically next ALLOCATE directed to be from DDR (not MCDRAM).
FOR_GET_HBW_AVAILABILITY	ifcore module routine to tell if either the memkind library was not linked in or that MCDRAM is unavailable (return values discussed in the text of this chapter)
FOR_SET_FASTMEM_POLICY(*X*)	ifcore module routine to set FASTMEM policy for MCDRAM allocation. Parameter *X* can be: FOR_K_FASTMEM_INFO: return the current FASTMEM policy value. FOR_K_FASTMEM_NORETRY: issue an error (check ALLOCATE stat variable) if MCDRAM is not available. FOR_K_FASTMEM_RETRY_WARN: warn if MCDRAM is not available, then use DDR. FOR_K_FASTMEM_RETRY: if MCDRAM is not available, then automatically (and silently) use DDR.

FIG. 3.13

Summary of Fortran extensions to support MCDRAM usage.

These are extensions to Fortran and currently are unique to the Intel Fortran compiler on Linux. The initial directives were originally introduced in 2015 with the 16.0 compiler. The more recent 17.0 compiler (beta and release in 2016) offers additional control over policies centering on the concepts of "BIND" (require) versus "PREFER" (preferred) allocations. A hot area of debate is failure mode—in other words, what to do if MCDRAM is oversubscribed and how to fail. After we introduce how to use the extensions for MCDRAM, we have a section titled *Fortran FASTMEM failure modes* to discuss options for controlling MCDRAM over-allocation policies.

Extending Fortran ALLOCATE, for MCDRAM, is an emerging capability subject to change based on experiences and customer feedback. To augment our introduction here, our web site has the latest information on this topic—lotsofcores.com/ALLOCATE.

The ATTRIBUTES FASTMEM directive enables MCDRAM memory allocation for an allocated object. The Fortran documentation, like memkind, calls this "HBW" memory. The directive syntax is:

```
!DIR$ ATTRIBUTES FASTMEM :: object
```

The *object* must be an allocatable array. It may be a component of a derived type. It cannot be a local variable, a common block, or an array that will be allocated on the stack.

`FASTMEM` enables MCDRAM memory allocation for an allocated object at runtime. When the specified object is allocated using the `ALLOCATE` statement at runtime, the Fortran Run-Time Library (RTL) allocates the memory in MCDRAM. When the specified object is deallocated using the `DEALLOCATE` statement at runtime, the Fortran RTL deallocates the memory from MCDRAM.

When we use this directive in an application, we must specify the option `-lmemkind` to the compiler/linker. If the libraries required for MCDRAM support are not linked successfully, no warning will be given at compile time and the allocations will fail at runtime.

The Fortran compiler will link executables correctly even if we forgot to link in libmemkind. There is a Fortran runtime called `for_get_hbw_availability()` in the ifcore module to tell us that either the memkind library was not linked in or that MCDRAM is unavailable. The return value represents one of the following values in the ifcore module:

- `FOR_K_HBW_AVAILABLE`—MCDRAM is available.
- `FOR_K_HBW_NO_ROUTINES`—The routines needed for MCDRAM memory allocation are not linked-in (i.e., no libmemkind).
- `FOR_K_HBW_NOT_AVAILABLE`—MCDRAM memory is not available.

We can use ATTRIBUTES option ALIGN with FASTMEM; for example:

```
!DIR$ ATTRIBUTES FASTMEM, ALIGN:64 :: A
```

We can specify MCDRAM usage on each ALLOCATE instead of relying on a directive for the declaration which affects all ALLOCATE statements for affect variables. This gives us more control. We use a directive before the ALLOCATE statement which we want to go to MCDRAM as shown in Figs. 3.14 and 3.15. The FASTMEM directive has a counterpart NOFASTMEM to add clarity or for

```
real, allocatable :: x (:)
! …code…
!dir$ FASTMEM      ! FASTMEM just for the following
ALLOCATE
allocate (x (10), stat = istat)
```

FIG. 3.14

FASTMEM directive used for a specific ALLOCATE.

```
TYPE T
REAL, ALLOCATABLE :: x (:)
END TYPE T
TYPE (T) :: r
…
!dir$ FASTMEM
ALLOCATE (r%x (10000))   ! allocates in MCDRAM
```

FIG. 3.15

FASTMEM directive used to affect allocation of a field.

overriding a FASTMEM directive used on the original declaration. It is best to use NOFASTMEM on an ALLOCATE used to recover from a FASTMEM ALLOCATE.

Fortran FASTMEM failure modes

In this section, we discuss options for controlling MCDRAM over-allocation policies. This section is only important if our program might ever request an ALLOCATION for MCDRAM when sufficient MCDRAM was not available.

Extending Fortran ALLOCATE, for MCDRAM, is an emerging capability subject to change based on experiences, customer feedback, and eventually the standardization processes. To augment our introduction here, our web site has the latest information on this topic, see *For More Information* at the end of this chapter.

We can envision these policies for MCDRAM over-subscription:

1. *ALLOCATE prefers MCDRAM* but will allocate from DDR if MCDRAM is not sufficiently available. It does not fail due to "near" MCDRAM over-subscription, leaving physical pages to be allocated either at ALLOCATE or at first touch from MCDRAM is available but use DDR if MCDRAM is not available. If a page of MCDRAM is not available to allocate, then allocations should be from the DDR. If this fails, the program would be aborted as would happen for other out-of-memory errors. Linux NUMA software calls this a "PREFERRED" policy.
2. *ALLOCATE requires MCDRAM* and will fail if MCDRAM is not sufficiently available. One challenging question is whether the failure will be reported by ALLOCATE, or will happen at "first touch" triggering an out-of-memory program abort. The latter is the behavior in C/C++ program. The former, the desire to have ALLOCATE fail, may be an experimental feature of the Intel beta 17.0 compiler for feedback from the user community. Linux NUMA software calls this a "BIND" policy.

Two side-note notes (not critical to learning how to use MCDRAM): In all cases, we say "MCDRAM" but really mean "near MCDRAM." The "near" concept only matters for SNC-2 and SNC-4 modes; otherwise MCDRAM is always "near." Linux supports an "INTERLEAVE" policy also, intended for users that have a NUMA system but prefer to spread allocations out to present a mix of near/far allocations to even out

as if on an UMA machine. Even though Knights Landing can be configured in such a mode, we do not anticipate INTERLEAVE policy being used on Knights Landing.

ALLOCATE prefers MCDRAM

A Fortran program may set a policy to cause ALLOCATE to use DDR when MCDRAM is not otherwise available using an ifcore.mod function:

```
FOR_SET_FASTMEM_POLICY (FOR_K_FASTMEM_RETRY)
```

This function should be called early in the program and only once. It should be called before the first ALLOCATE statement. Failure to adhere to those restrictions will probably not provide the desired results. There are additional options as shown in Fig. 3.13.

ALLOCATE requires MCDRAM

A Fortran program may set a policy to force ALLOCATE to use only MCDRAM using an ifcore.mod function:

```
FOR_SET_FASTMEM_POLICY (FOR_K_FASTMEM_NORETRY)
```

This function should be called early in the program and only once. It should be called before the first ALLOCATE statement. Failure to adhere to those restrictions will probably not provide the desired results. There are additional options as shown in Fig. 3.15.

There is considerable disagreement about which policy should be used. Some believe the best policy would be to have ALLOCATE fail at allocation time if MCDRAM is over subscribed. With this policy, if insufficient MCDRAM is available to allocate the object to MCDRAM, the allocation may fail, leaving the status in the variable specified in the `STAT=` clause.

Combined with the FASTMEM directive for a specific ALLOCATE statement, we show how to handle a failure at an ALLOCATE statement in Fig. 3.16. The FASTMEM directive has a counterpart NOFASTMEM that is useful for clarity and for overriding any directive that may have been used on the original declaration. It is best to use NOFASTMEM on an ALLOCATE used to recover from a FASTMEM ALLOCATE.

```
real, allocatable :: x (:)
! …code…
!dir$ FASTMEM       ! FASTMEM just for the following ALLOCATE
allocate (x (10), stat = istat)
if (istat .ne. 0) then
  !dir$  nofastmem    ! we'll take  any kind of memory for X
  allocate (x (10), stat = istat)
  if (istat .ne. 0) then …      ! no standard memory either
```

FIG. 3.16

Handing an ALLOCATE failure (experimental Fortran extension).

QUERY MEMORY MODE AND MCDRAM AVAILABLE

Applications designed to use MCDRAM in flat or hybrid mode should be sensitive to the amount of MCDRAM available. Assuming a 16-GB MCDRAM on a Knights Landing, the memory mode and cluster mode selected can both influence the amount of memory available. The memory mode may make 100%, 75%, or 50% of the MCDRAM available as flat memory (16, 12, or 8 GB). Of course, if booted in cache mode, the amount of high-bandwidth memory available will be 0. Furthermore, as we will discuss in Chapter 4 (section titled *Interactions of Cluster and Memory Modes*), the Knight Landing may be subdivided into halves or quarters. In such a case, 16 GB can become 16, 12, or 8 GB. Likewise, 12 GB can become 12, 9, or 6 GB and 8 GB can become 8, 6, or 4 GB of capacity available to the application per subdivision of the Knights Landing. Therefore, applications should not be written to assume a certain memory capacity.

The memkind library can be used to check for high-bandwidth memory starting with a `memkind_check_available(MEMKIND_HBW)` or `hbw_check_available()` to check for existence at all. A nonzero return would indicate no MCDRAM is available, which would happen on Knights Landing when booted in cache mode, or on processors with no MCDRAM at all. The function `memkind_get_size(MEMKIND_HBW,&total,&free)` is currently very limited. It will return an amount of memory that is available for use, but it has limitations that prevent it from reporting memory available in the free pool. It also reports memory available for use and is unaffected by allocations of memory without any usage occurring.

These functions are not efficient and therefore should not be called repeatedly in an application especially in any performance-sensitive code. It is best to call once and cache the value for an application. Fig. 3.17 offers a quick example of using these functions in C. Fig. 3.18 shows the output when run on the same 64-core Knights Landing which we have used previously in this chapter (equipped with 64 GB of memory plus 16 GB of MCDRAM).

SNC PERFORMANCE IMPLICATIONS OF ALLOCATION AND THREADING

For the best performance, allocating memory for threads should be done carefully in order to ensure memory is allocated "near" to the work. We can imagine a master thread in an application that allocates and initializes all the memory for the process and then lets the thread pools work on that memory. If the master thread and the worker threads reside in different NUMA nodes, then they will have suboptimal performance. Another variation is that you have multiple thread pools in different NUMA domains, but the first thread pool did the initial "touch," which allocated the physical space behind the virtual allocation. Careful planning and understanding of these problems will be helpful to achieve maximum performance.

```
#include <stdio.h>
#include <memkind_hbw.h>
#include <hbwmalloc.h>

int main(int argc, char *argv[]) {
 size_t total, free;
 int policyis, w, int *mem;
 char *hbw = getenv("MEMKIND_HBW_NODES");
 if (hbw)
  printf("Environment variable "
         "'MEMKIND_HBW_NODES' = '%s'\n",hbw);
 printf("memkind_check_available(MEMKIND_HBW) ="
        "%d (zero means yes)\n",
        memkind_check_available(MEMKIND_HBW));
 printf("memkind_get_size(MEMKIND_HBW) = %d\n",
        memkind_get_size(MEMKIND_HBW,&total,&free));
 printf("hbw_check_available() = %d (zero means yes)\n",
        hbw_check_available());
 printf("total = %15.6fM; free = %15.6fM\n",
        total/1024/1024.0,free/1024/1024.0);
 printf("total = 0x%012lx; free = 0x%012lx\n",
        total,free);
 policyis = hbw_get_policy();
 printf("policy=%d\nHBW_POLICY_BIND=%d\n"
        "HBW_POLICY_PREFERRED=%d\n",
        policyis,
        HBW_POLICY_BIND==policyis,
        HBW_POLICY_PREFERRED==policyis);
 mem = hbw_malloc((size_t)512);
 if (mem<0) printf("Error: ");
 printf("hbw_malloc = %012lx\n",mem[w]);
 return 0;
}
```

FIG. 3.17

Simple "Hello memkind" application.

```
memkind_check_available(MEMKIND_HBW) =0 (zero means yes)
memkind_get_size(MEMKIND_HBW) = 0
hbw_check_available() = 0 (zero means yes)
total =     16384.000000M; free =     15926.421875M
total = 0x000400000000; free = 0x0003e366c000
policy=2
HBW_POLICY_BIND=0
HBW_POLICY_PREFERRED=1
hbw_malloc = 7fcbf740e200
```

FIG. 3.18

Output of our simple "Hello memkind" application.

ALLOCATION WITH SNC

Since SNC is software visible, it requires some special considerations to utilize well in the context of memory allocators. Because modern operating systems use an allocation-on-demand strategy to assign physical pages to virtual addresses only when accessed, some considerations will help performance. For optimal performance data should be allocated and first touched by code running on the NUMA node which will use the particular data the most. This warrants extra thought if memory is allocated, used, freed and reallocated by a central memory allocator. While Linux does support migration of memory from one NUMA node to another, this is an expensive operation, just like returning memory to the operating system, and is worth avoiding if we can.

The right NUMA node for a given NUMA allocation request depends upon the thread asking for it, since SNC subdivides memory and the cores/threads. By default software processes and threads can migrate across the operating system-managed hardware threads. If a calling software thread allocates memory from one node, it may be moved to another node, making the access slower. Intel MPI and OpenMP implementations allow us to affinitize their threads, typically through environment variables so that the migration problem does not happen.

HOW TO NOT HARD CODE THE NUMA NODE NUMBERS

While writing this chapter, we heard the terse summary: "If the system administrator has installed memkind correctly, `MEMKIND_HBW_NODES=1` is not required at all." We will go further: We should avoid the `MEMKIND_HBW_NODES` environment variable all together. This short section is about how to do that.

Avoid using the `MEMKIND_HBW_NODES` environment variable all together.

The Platform Memory Topology Table (PMTT) is a relative newcomer to the ACPI specification (ACPI 5.0, Dec. 6, 2011) and seems likely to be expanded or displaced in the future by other more comprehensive tables. No application should try using the PMTT directly, but the memkind library does so for us. The PMTT contains both bandwidth and latency information that is critical in the reliable identification of MCDRAM. Its existence allows MCDRAM allocation support to automatically select the appropriate MCDRAM from which to allocate.

It may be the case that some systems lack PMTT support, or the memkind library was not installed with PMTT support. For such cases, the environment variable `MEMKIND_HBW_NODES` exists to specify the NUMA node to utilize. This environment variable will override the PMTT table, and we strongly advise against using it if a PMTT table exists. We recommend figuring out how to get PMTT supported on every system!

Assuming the PMTT exists and memkind is properly installed to take advantage of it, the memkind will allocate MCDRAM using the "near" MCDRAM and then, if the request was a preference instead of a requirement, use the "near" DDR. This is essential in the SNC cluster modes because a single value for `MEMKIND_HBW_NODES` would not get the best results.

Since the PMTT is not readable by users, there are specific things done by the installation of memkind that manage access to the information. This is an installation issue, and the details are documented in the memkind package.

The `memkind_hbw_check_available()` call will return a value of `MEMKIND_UNAVAILABLE` if the PMTT table information does not exist, or if it does not indicate the existence of MCDRAM (which does happen when memory mode = cache).

With the PMTT and MCDRAM both in place, `hbw_malloc` and other memkind functions that request MCDRAM, will allocate from "near" MCDRAM and then fall back on DDR allocations ("near" DDR first). This should eliminate our temptation to control with `MEMKIND_HBW_NODES`.

APPROACHES TO DETERMINING WHAT TO PUT IN MCDRAM

Hopefully, earlier sections illustrated that managing MCDRAM explicitly is up against some stiff competition for high performance without resorting to modification of an application. Nevertheless, as programmers we can "know best" and get better performance for our applications if the application has a working set larger than the MCDRAM. Of course, if all our data fits inside the MCDRAM, we have little hope of performance upside! In fact, it does not have to be literally all the data; if the working set fits into MCDRAM, then we will see benefits essentially identical to having everything fit in MCDRAM. There is a strong attraction to simply scaling-out an application to use more Knights Landings so that the working set per Knights Landing fits in MCDRAM.

Assuming we decide to try explicit management of MCDRAM (in flat or hybrid memory modes) using the `memkind` library or `FASTMEM` directives, we need to decide what data to place into MCDRAM.

This section examines two approaches for identifying which application data structures should be explicitly allocated within MCDRAM in order to maximize performance.

> Approach 1: Timings with select data items allocated fast/slow. The best method to do this is using Knights Landing (Approach 1a), but we also show how to approximate it with a dual-socket system based on Intel® Xeon® processors (Approach 1b).
> Approach 2: Use the "memory profiling" capability of the Intel® VTune™ Amplifier tool.

Approach 1a and 1b require code changes. Approach 1b utilizes the two memory regions present on a dual-socket Intel Xeon system (a NUMA environment) to emulate a Knights Landing node containing both MCDRAM and DDR memory, while

Approach 2 utilizes the Intel VTune memory profiling technology to identify the most suitable candidates on the program without requiring coding changes.

We will demonstrate the utilization of these techniques with a real-world hydrodynamics application. This application, called CloverLeaf, is widely utilized for research purposes. Although CloverLeaf is a proxy application (mini-app), the code-base is sufficiently representative to allow meaningful conclusions to be formed regarding the performance of these techniques within larger, more complex applications. Performance results using a Knights Landing, obtained through the application of each of these approaches, are presented in Fig. 3.22. For further information on CloverLeaf, see *For More Information* at the end of this chapter.

APPROACH 1: OBSERVING OR EMULATING MCDRAM EFFECTS

These approaches start by identifying candidate data structures, and running timing tests to determine the benefits of moving into MCDRAM. We can utilize an automated trial-and-improvement-based methodology to examine the performance of an application when executed with alternative setup parameters.

We offer Approach 1a using actual runs on Knights Landing machines, and slightly less accurate Approach 1b using a NUMA system and forcing fast/slow through memory allocations to "near" and "far" memories.

The Approach 1b execution environment is configured such that only one Intel Xeon processor within the system is utilized to actually execute the application. The MCDRAM targeted memory allocations, using the `hbw_malloc` function, within the application are placed within the memory subsystem of the local processor. The standard DDR memory allocations, however, are placed within the memory subsystem of the "remote" processor ("far" memory). Applications will therefore experience higher latency and lower bandwidth for the memory accesses which are serviced from the "remote" memory subsystem, relative to those allocated from within the "local" memory domain. This is due to the fact that these accesses need to traverse the QPI links between the two processor sockets. Using the two different memory regions in this manner effectively emulates the different memory performance

```
#ifdef mcdram_array1
!DIR$ ATTRIBUTES FASTMEM :: array1
#endif

And for C/C++ codes modify the allocations as follows:
float *array1
#ifdef mcdram_array1
array1 = (float *) hbw_malloc( sizeof(float) * N);
#else
array1 = (float *) malloc( sizeof(float) * N);
#endif
```

FIG. 3.19

Examples of changes needed for Approach 1.

characteristics which an application experiences when executing on a Knights Landing node.

It is *not* recommended, however, that applications should be optimized for Approach 1b, as it is not a perfectly accurate representative of the bandwidth and latency characteristics found within Knights Landing. This approach does, however, represent a reasonable, low-effort way of determining which application data structures are most bandwidth sensitive and therefore should be preferably allocated within the MCDRAM memory domain in order to maximize overall performance.

Prerequisites:

1. Obtain access to a Knights Landing or a dual-socket Intel Xeon processor-based system running Linux
2. Install the memkind library

Stage 1: Code modification

Modify the application such that the allocation of its most significant data structures can be individually controlled via preprocessor macros. For Fortran codes, include FASTMEM directives. In the CloverLeaf codebase, our changes can be found in the `definitions.f90` file. For C/C++ codes, switch explicit allocations to utilize the memkind `/hbwmalloc` interfaces. Examples are shown in Fig. 3.19.

Stage 2: Manual Execution

1. Manually build multiple versions of the application, each of which places a different data structure into the memory region emulating the Knights Landing "MCDRAM," all other allocations should continue to be allocated within the memory region emulating the Knights Landing "DDR" resources.
   To activate our sample code in Fig. 3.20, one of the compilations would include `-Dmcdram_array1`. This could also be an opportunity to print out, or otherwise capture, the allocation sizes as input into our later thinking of how to use this information. In the optional "Stage 3," we suggest ways to automate this for larger applications.
2. Ensure MCDRAM allocations will be located where we need them.
   a. For Approach 1a—we want them actually in the MCDRAM, set the `MEMKIND_HBW_NODES` environment variable with a command such as this bash shell command (node 1 assumes Quadrant/Flat configuration):

      ```
      $ export MEMKIND_HBW_NODES = 1
      ```

   b. For Approach 1b—we want them located within the local "near" memory domain, set the `MEMKIND_HBW_NODES` environment variable with a command such as this bash shell command:

      ```
      $ export MEMKIND_HBW_NODES=0
      ```

3. Configure the execution environment (e.g., `OMP_NUM_THREADS`, etc.) such that the application does not require computational resources beyond those available within one processor of the target platform.

```
#!/bin/bash

export OMP_NUM_THREADS=<# cores on one processor>
export MEMKIND_HBW_NODES=0

compile_and_run() {
    make clean 1>/dev/null 2>&1
    make COMPILER=INTEL \
      MPI_COMPILER=mpiifort \
      C_MPI_COMPILER=mpiicc \
      FASTMEM=$1  1>/dev/null 2>&1
    mv clover_leaf clover_leaf_S1
    numactl --membind=1 \
      --cpunodebind=0 \
      ./clover_leaf_$1 1>clover_output_$1 2>&1
    mv clover.out clover.out_$1
    rm clover.in.tmp
}

compile_and_run "density0"
compile_and_run "density1"
…(repeat for each data array)…
```

FIG. 3.20

Pseudocode (based on the bash shell) for use with the CloverLeaf application.

4. Execute each instance of the application. For Approach 1b (only), we want to use a numactl command that binds memory allocations by default to the "far memory":

   ```
   $ numactl -m 1 -N 0 ./app <app options>
   ```

5. Record the runtimes for each execution of the application and sort these with execution times from the lowest to the highest.
6. Identify the data structures which achieved the lowest overall execution times.
7. Within the final application source code/binary, allocate the top *N* data structures within the high-bandwidth memory, up to the capacity limitations of the MCDRAM memory resources available within the Knights Landing nodes.

Revisiting steps 1–5 with multiple "-D" definitions active at once might be helpful, making sure the allocations either fit within the MCDRAM size we expect. Also, we can consider the order that we do allocations into MCDRAM both from the perspective of priority and packing within different MCDRAM sizes. Recall that when using an SNC cluster mode, we will most likely consider available MCDRAM size to be reduced by focusing on the "near" portion of MCDRAM instead of all of it.

Stage 3: Autotuning Configuration (Optional)

For large complex applications, it may not be practical to manually build and execute each different variant of the application. In this scenario, we recommend utilizing your favorite autotuning framework (or scripting language) such that the application can be repeatedly built, using different preprocessor definitions, and executed on the

target platform using the approach outlined in stage 2. Our example pseudocode for use with the CloverLeaf application, to implement Approach 1b, is shown in Fig. 3.20.

APPROACH 2: USING INTEL VTune TO DETERMINE MCDRAM CANDIDATE DATA STRUCTURES

In this approach, the memory profiling facilities of the Intel VTune™ Amplifier (Chapter 14, although we do not duplicate a discussion of "memory profiling" in that chapter) are utilized to identify which application data structures should be explicitly allocated within MCDRAM in order to maximize performance.

Prerequisite:

- Obtain access to a Knights Landing or an Intel Xeon processor-based system, with the Intel Compiler suite and VTune profiler technology installed.

Stage 1: Profiling Data Collection

1. Build the application such that the Intel compiler generates full debugging information within the resulting object files (e.g., by employing the -g compiler option on Linux or the/Z7 compiler option on Windows).
2. Configure the execution environment such that the application will execute across all of the available processing resources within the target node (i.e., set `OMP_NUM_THREADS`, etc., as required).
3. Execute the application under VTune such that the profiler conducts a "memory analysis." The following command demonstrates how to achieve this using the command line version of VTune on a Linux-based system:

```
$ amplxe-cl -collect memory-access -data-limit=0 -knob analyze-mem-objects=true -knob mem-object-size-min-thres=1024 -r <insert results directory name> -app-working-dir /path/to/working_directory--/path/to/working_directory/application binary
```

Stage 2: Profiling Data Analysis

1. After the profiling analysis has completed, load the graphical version of the VTune profiler.
2. On the "Welcome" screen, select "Open Result" and navigate to the directory in which you saved the profiling result directory generated in Stage 1. Open the resulting `<name>.amplxe` file.
3. The "Memory Analysis" profiling results will be displayed and ensure that the "Memory Usage" viewpoint is selected.
4. Navigate to the "Bottom-up" tab.
5. Select the "Bandwidth Domain/Bandwidth Utilization Type/Memory Object/Allocation Stack" grouping.

6. In the resulting profiling data, expand the "DRAM GB/sec" and then the "High" entries.
7. Order the results based on the "LLC Miss Count" field.
8. This will provide an ordering for the data structures within the application that are most memory bandwidth sensitive.

Stage 3: Code Modification

- Modify the application code such that the highest priority data structures (as identified from the list derived from Stage 2) are allocated within the MCDRAM available on the target Knights Landing. Recall that available MCDRAM sizes are reduced when dividing Knights Landing up using a NUMA cluster mode.

RESULTS ANALYSIS OF THE TWO APPROACHES

Using the approaches described above, several experiments were conducted to identify which application data structures within the 3D version of CloverLeaf should be explicitly allocated within MCDRAM in order to deliver the largest speedup in application execution time. The priority orderings for the data structures obtained through each of the above methods are presented in Fig. 3.21, together with results in Fig. 3.22 obtained from experiments on Knights Landing using the autotuning methodology outlined previously in "Approach 1."

The results show that both approaches deliver a preference ordering, for the allocation of application data structures within MCDRAM, which closely matches the performance results observed on a Knights Landing.

Additionally, Fig. 3.22 shows the speedup achieved in the execution time of the CloverLeaf 3D application by explicitly allocating data structures within the MCDRAM resources, relative to only utilizing standard DDR memory. In these experiments, the 600^3 cell benchmarking simulation problem was utilized and executed for 87 time-steps. The eight highest priority data structures, as identified by each of the previous approaches, were able to be explicitly allocated within the MCDRAM (up to the capacity limits of this memory region running in Quadrant/Flat—16 GB).

SUMMARY OF TWO APPROACHES TO "WHAT GOES IN MCDRAM"

Overall the profiling tool-based methodology described in "Approach 2" represents the easiest and least time consuming method for conducting this analysis. This "memory profiling" capability in VTune is relatively new, and is being refined and extended. We highly recommend it. The auto-tuning/trial-and-improvement-based methodology described in "Approach 1" requires significantly more effort and potentially requires substantial source code modifications. It also takes significantly longer to complete due to the multiple application builds and executions which it requires. Although the experimental results show that for this application, the autotuning-based approach can deliver improved performance results and that the

MCDRAM Allocation Preference Order	Approach 1a: Code changes to explore actual MCDRAM performance on Knights Landing	Approach 1b: Code changes to get predictions on MCDRAMeffects using dual-socket Xeon (NUMA system)	Approach 2: No code changes needed –use VTune Memory Profiling feature
1	**work_array1**	work_array1	work_array2
2	**work_array5**	work_array5	work_array3
3	**work_array7**	work_array3	work_array5
4	**work_array3**	work_array7	work_array1
5	**work_array4**	work_array 2	vol_flux_z
6	**density1**	work_array4	volume
7	**work_array6**	work_array6	yvel1
8	**work_array2**	density1	zvel1
9	vol_flux_z	vol_flux_z	xvel1
10	energy1	zvel1	pressure
11	volume	energy1	density1
12	xvel1	yvel1	energy0
13	zvel1	pressure	vol_flux_y
14	pressure	volume	vol_flux_x
15	yvel1	xvel0	density0
16	xvel0	xvel1	viscosity
17	viscosity	zvel0	zarea
18	yvel0	yvel0	yarea
19	vol_flux_x	vol_flux_x	energy1
20	vol_flux_y	viscosity	xarea
21	zvel0	yarea	xvel0
22	xarea	xarea	mass_flux_z
23	density0	vol_flux_y	yvel0
24	zarea	zarea	zvel0
25	yarea	density0	mass_flux_y
26	mass_flux_y	soundspeed	mass_flux_x
27	mass_flux_x	energy0	work_array7
28	mass_flux_z	mass_flux_z	soundspeed
29	soundspeed	mass_flux_y	celldx
30	energy0	mass_flux_x	celldy
31	cellx	celldz	celldz
32	vertexdx	vertexdx	vertexdx
33	vertexdz	vertexx	vertexdy
34	celldy	vertexdy	vertexdz
35	celly	celldx	work_array4
36	cellz	vertexz	work_array6
37	vertexdy	vertexy	vertexz
38	vertexy	celldy	cellx
39	vertexz	vertexdz	cellz
40	vertexx	cellx	vertexy
41	celldx	cellz	celly

FIG. 3.21

Preference ordering for the allocation of data structures within MCDRAM found empirically and by the various approaches. The top eight on the list do fit into MCDRAM in our case.

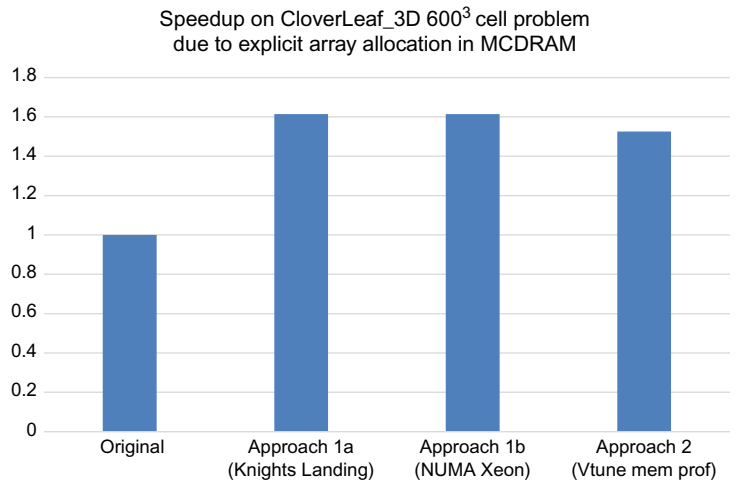


FIG. 3.22

Speedups achieved by explicitly allocation into MCDRAM based on recommendations.

array allocation preference ordering which it identifies is closer to that observed on actual Knights Landing silicon.

Utilizing the approach which employs dual-socket Intel Xeon processors to emulate a Knights Landing node (Approach 1) also facilitates implementation of more advanced explicit memory management strategies using the `memkind` library, and to examine their performance without needing actual Knights Landing-based systems. These may, for example, include the implementation of solutions in which data structures are explicitly staged into the MCDRAM and then, following the completion of computational operations, staged back to the DDR Knights Landing memory regions.

Before modifying applications to explicitly allocate data structures within MCDRAM, we would encourage the use of one or both approaches to gather useful information if the utilization of the autotuning-based approach proves to be infeasible for particular applications. The speedup results obtained through the application of the VTune profiling tool-based methodology (Approach 2) show that this approach can deliver performance improvements which are comparable to those achievable using the substantially more time-consuming autotuning-based method (Approach 1).

WHY REBOOTING IS REQUIRED TO CHANGE MODES

A common question that is asked about Knights Landing is "Why is a reboot necessary to change modes?"

By designing Knights Landing to offer MCDRAM as cache or memory, and to allow the cores to be configured in five different cluster configurations which span NUMA and UMA cluster designs, we have a single-chip solution that can change things about the design of a machine at a level that have traditionally been unchangeable. Operating systems and applications are not prepared to have NUMA distances change or go away, key memories come and go, and caching structures change. It is clear that when modes change that applications need to be stopped, the operating system needs to rebuild structures around the new shape of the machine, and the processor needs to purge most internal tables associated with memory tags, routing, and caching. The way to do that is a reboot of the system, which reinitializes all the information of the transformed Knights Landing when a cluster or memory mode is changed at all levels: applications, operating systems, and the processor itself.

As we will see, the ability to change the parameters for the next boot, coupled with efforts to accelerate reboots, is the best solution available for this unusual ability to pack multiple system designs in a single processor.

BIOS

The memory and cluster modes of Knights Landings are set during the boot process by the system BIOS. An obvious way to affect these settings is to boot into the BIOS, change the settings, exit, and reboot (Figs. 3.23–3.25). Since this requires two reboots to change modes, system vendors offer utilities to change BIOS settings from the command line so that new settings will take affect on the next boot.

We will describe how to do this on systems with Intel motherboards and an Intel-supplied BIOS. We will gladly add information for other vendors if we are contacted with pointers to the information. We have reserved a URL for us to share information that we mention in *For More Information* at the end of this chapter.

SAVE/RESTORE/CHANGE ALL BIOS SETTING

We can save BIOS settings to a file using the command:

```
sudo /bin/syscfg/syscfg /s BIOSQcache.ini
```

And we can restore them using the command:

```
sudo /bin/syscfg/syscfg /r BIOSQcache.ini/b
```

But more interestingly, we can change the three mode settings with individual commands. Figs. 3.26–3.28 show how to display the mode settings from a Linux command line (Intel offers Windows support as well).

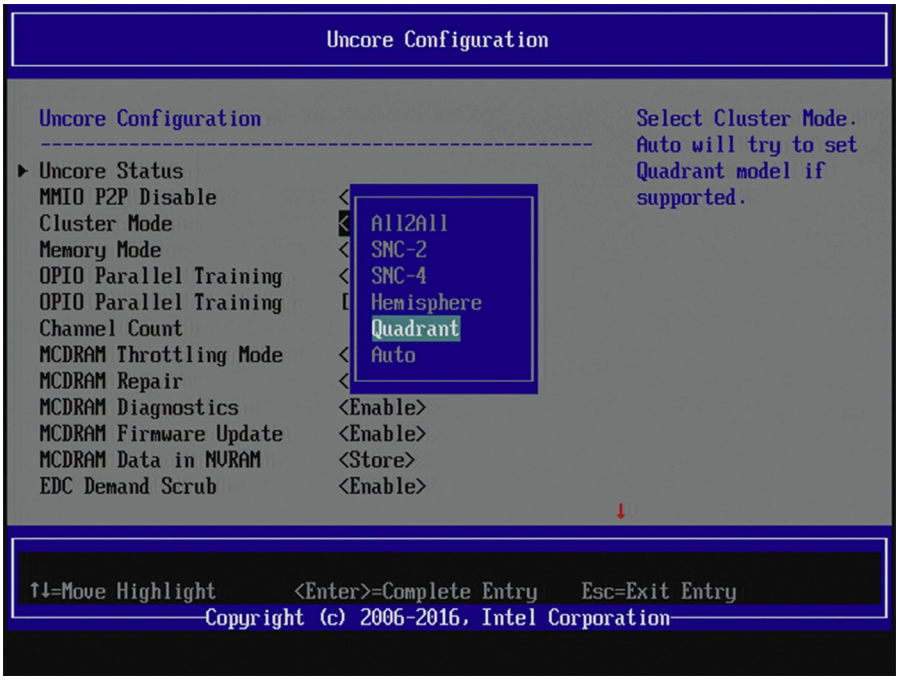


FIG. 3.23

Example BIOS settings for the Cluster Modes.

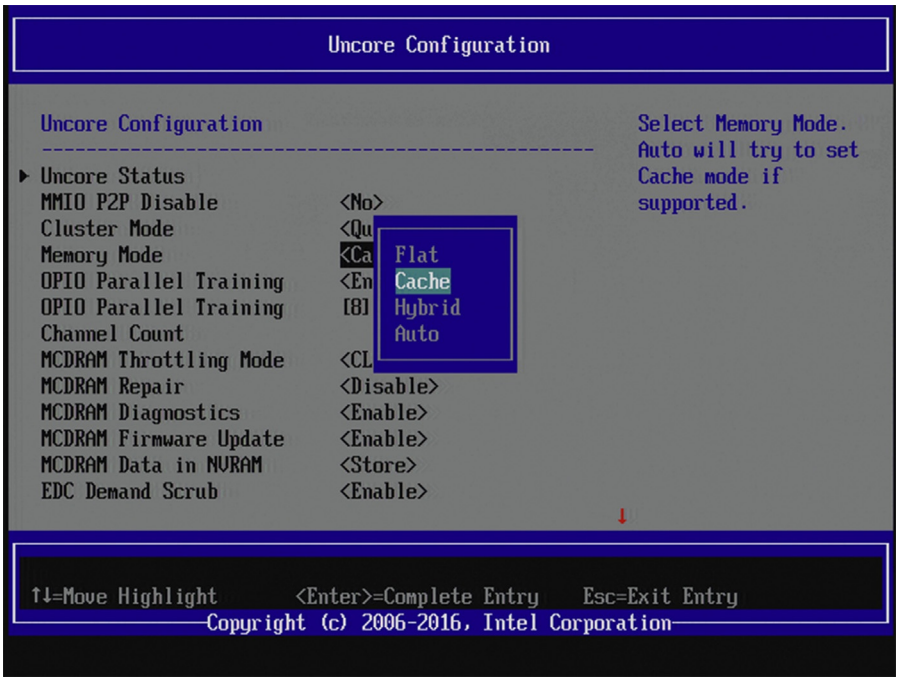


FIG. 3.24

Example BIOS settings for the Memory Modes.

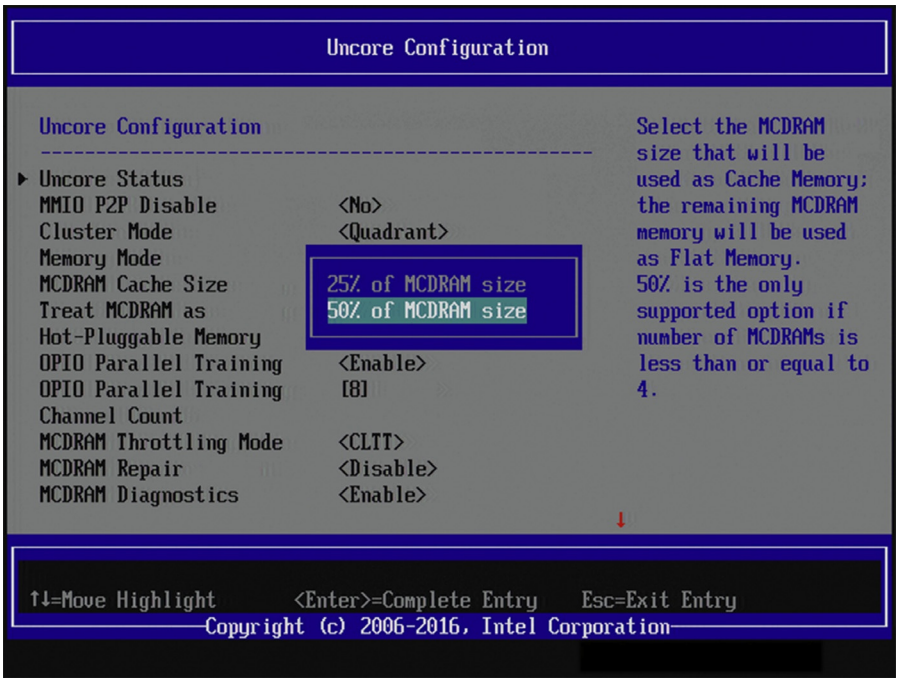


FIG. 3.25

Example BIOS settings for when "Hybrid" memory mode is used.

```
sudo /bin/syscfg/syscfg /d biossettings "Cluster Mode"

Cluster Mode
============
Current Value : Quadrant
------------------------
Possible Values
---------------
All2All : 00
SNC-2 : 01
SNC-4 : 02
Hemisphere : 03
Quadrant : 04
Auto : 05
```

FIG. 3.26

Displaying the current Cluster Mode setting using the Intel syscfg tool.

```
sudo /bin/syscfg/syscfg /d biossettings "Memory Mode"

Memory Mode
===========
Current Value : Cache
---------------------
Possible Values
---------------
Cache : 00
Flat : 01
Hybrid : 02
Auto : 03
```

FIG. 3.27

Displaying the current Memory Mode setting using the Intel syscfg tool.

```
sudo /bin/syscfg/syscfg /d biossettings "MCDRAM Cache Size"

MCDRAM Cache Size
=================
Current Value : 25% of MCDRAM size
----------------------------------
Possible Values
---------------
25% of MCDRAM size : 01
50% of MCDRAM size : 02
```

FIG. 3.28

Displaying the current MCDRAM Cache Size setting using the Intel syscfg tool.

```
SET: sudo /bin/syscfg/syscfg /bcs "" "Cluster Mode" 4
Successfully Completed
SET: sudo /bin/syscfg/syscfg /bcs "" "Memory Mode" 2
Successfully Completed
SET: sudo /bin/syscfg/syscfg /bcs "" "MCDRAM Cache Size" 1
Successfully Completed
```

FIG. 3.29

Setting the three modes using the Intel syscfg tool.

```
#!/bin/bash -f
if [ $# -lt 2 ]; then
  echo "usage: $0 ClusterModeCode MemoryModeCode [MCDRAMCacheSizeCode]"
fi
HYBRID=`sudo /bin/syscfg/syscfg /d biossettings "Memory Mode" | \
       grep -c "Current Value : Hybrid"`
if [ $# -gt 1 ]; then
  sudo /bin/syscfg/syscfg /bcs "" "Cluster Mode" $1
  sudo /bin/syscfg/syscfg /bcs "" "Memory Mode" $2
  HYBRID=`sudo /bin/syscfg/syscfg /d biossettings "Memory Mode" | \
       grep -c "Current Value : Hybrid"`
  if [ $# -gt 2 -a $HYBRID -gt 0 ]; then
    sudo /bin/syscfg/syscfg /bcs "" "MCDRAM Cache Size" $3
  fi
  echo "New values:"
else
  echo "Current values:"
fi

sudo /bin/syscfg/syscfg /d biossettings "Cluster Mode"
sudo /bin/syscfg/syscfg /d biossettings "Memory Mode"
if [ $HYBRID -gt 0 ]; then
  sudo /bin/syscfg/syscfg /d biossettings "MCDRAM Cache Size"
fi
echo " "
```

FIG. 3.30

Our "`setKNLmodes`" script to set and/or display current mode settings.

BIOS utilities can display the current BIOS setting. Since the settings can be changed and they only take affect at reboot time, the displayed value does not necessarily represent the mode the machine is currently in.

Fig. 3.29 shows how to set SNC-4 cluster mode with MCDRAM set to 25% cache. We wrote a little script to that can do this with a single command "`setKNLmodes 4 2 1`" which we show in Fig. 3.30. The script carefully avoids referring to the "MCDRAM Cache Size" parameter if the memory mode is not previously set to "Hybrid" since the Intel syscfg imposes that restriction.

An administrator password can be created for the BIOS, which would impact all the commands we have been shown. The documentation for the BIOS utilities (titled the *Intel® System Configuration Utility*) shows that we have to add "`/bap <BIOS administrator password>`" to the restore command line in such a case, or add the password where we show an empty string ("") in the single-setting-at-a-time `/bcs` commands.

SUMMARY

Key design choices have traditionally been made by hardware designers and software has had to adapt. Knights Landing turns this around and offers an unprecedented flexibility to leave that decision to the user of the machine. This ability to configure to suit an application is exciting but finding the optimal fit requires some understanding of the options. The appeal of directly programming for the "software controlled cache" (flat/hybrid memory modes) is obvious, but it needs to be tempered by the high-performance Knights Landing offers programs without modification through cache modes, numactl and autohbw. We should avoid using the environment variable `MEMKIND_HBW_NODES` and other methods to hard code NUMA numbers in our applications and scripts. Also, with careful usage the more sophisticated SNC cluster modes may benefit programs, especially MPI+X (e.g., MPI+OpenMP) style programming.

Ask your system provider for utilities to set BIOS options so as to avoid needing to boot into the BIOS to interactively change BIOS options related to cluster and memory modes. This will greatly reduce the time to leave a running system, reconfigure it, and be up and running again.

Such flexibility is new, and it is reasonable to assume more techniques and software support will emerge. We welcome feedback, and we will post errata and additional insights on our book website (http://lotsofcores.com).

FOR MORE INFORMATION

- Fortran allocation for MCDRAM is an emerging capability subject to change based on experiences and customer feedback. Therefore, we created a URL where we willl have pointers to the best information on this topic of which we are aware. http://lotsofcores.com/ALLOCATE.
- Memkind library information and github, https://github.com/memkind, clone via: "git clone https://github.com/memkind/memkind.git."

- Knights Landing BIOS configuration article on our website, which we will happily update with any pointers submitted to us on this topic, http://lotsofcores.com/KnightsLandingBIOS.
- Christopher Cantalupo, Vishwanath Venkatesan, Jeff R. Hammond, Krzysztof Czurylo, and Simon Hammond, *User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies*, Mar. 18, 2015, http://tinyurl.com/memkind-arch.
- Knights Landing "numactl" listings for all 20 modes, from one particular system. While your results may vary, it is interesting can be interesting to review. http://lotsofcores.com/KnightsLandingNUMActl.
- Overcommit settings in Linux are changeable, see http://tinyurl.com/overcommit-accounting.
- Configuring the out-of-memory killer in Linux: http://tinyurl.com/oom-killer.
- Further information on CloverLeaf, as well as the actual source code, can be found at: https://github.com/UK-MAC.
- Information about PMTT, and other aspects of ACPI tables, http://acpi.info.
- Download code from this chapter, and other chapters, at http://lotsofcores.com.

CHAPTER

Knights Landing architecture 4

In this chapter, we provide further details on the Knights Landing architecture that we introduced in Chapter 2. We describe the tile and core architecture in detail. We dive deeper into the cluster modes and memory modes supported by Knights Landing, explain how they interact, and mention important programming considerations when using them.

What is new with Knights Landing in this chapter?
This entire chapter is about Knights Landing.

TILE ARCHITECTURE

The Knights Landing tile (Fig. 4.1) is the basic unit that gets replicated across the chip. It consists of two cores, each core is connected to two vector processing units (VPUs), and shared 1 MB L2.

The Knights Landing core is a new core design. As described in Chapter 2, it was derived from a low power core, code named Silvermont originally designed for an Intel® Atom processor, but with heavy modifications to make it suitable for High-Performance Computing. It is a 2-wide, *out-of-order* core that supports up to four threads. This new core is one of the primary reasons for the significant improvement in single thread performance (>3×) in Knights Landing over the prior generation product in the Intel® Xeon Phi family code named *Knights Corner*.

The VPU is the heart of computation on Knights Landing. It is home to all the floating-point (FP) compute capability, ranging from legacy instructions like SSE and x87 through AVX and AVX2 to the latest AVX-512 vector instructions. Each core is connected to two VPU units and hence can execute two 512-bit vector multiply-add instructions per cycle. Thus, each core can deliver 32 dual-precision (DP) or 64 single-precision (SP) FP operations each cycle.

The L2 cache is 1 MB in capacity with 16-way associativity with 64B cache lines that is shared between both cores in the same tile. The L2 cache supports a bandwidth of 1 cache line read and ½ cache line write per cycle, which is shared by both cores as depicted in Fig. 4.2. The bus interface unit (BIU) controls the L2 cache and coherency between the two cores in the tile. One core can use all the available read and write bandwidth if the other core is not accessing the L2. The L2 cache is private to each

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00004-1

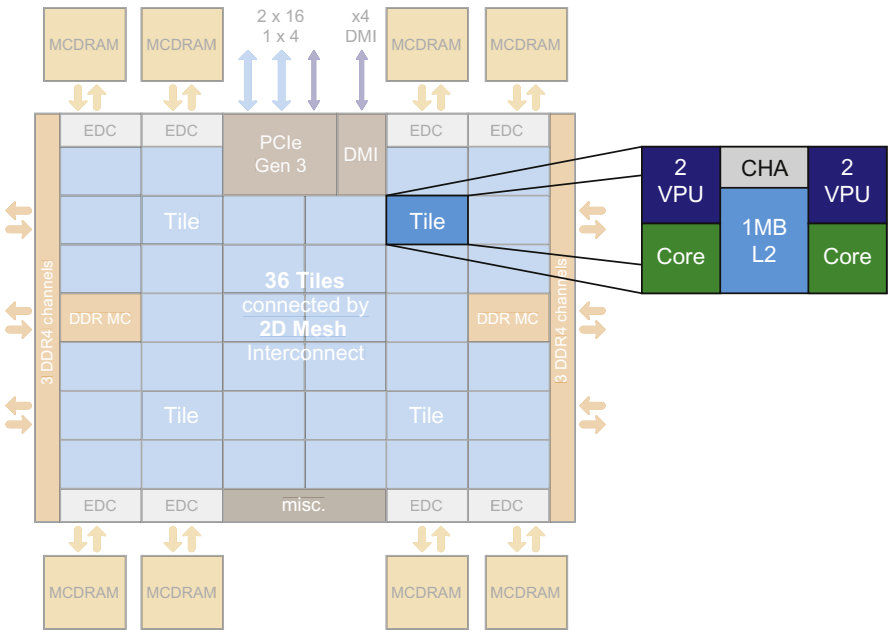


FIG. 4.1

Knights Landing tile within the larger processor die.

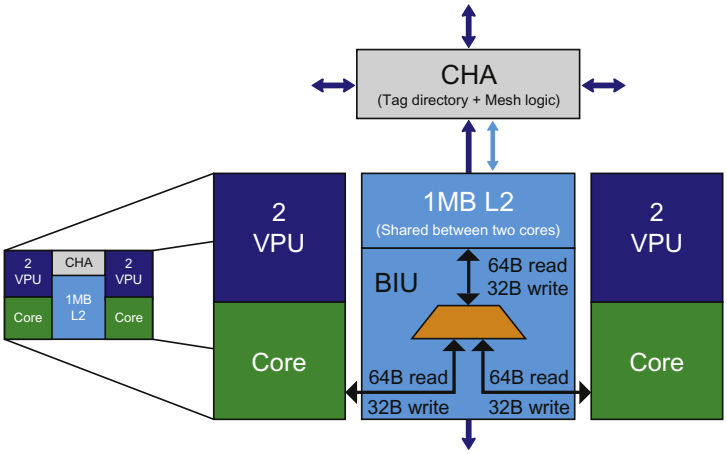


FIG. 4.2

Connections within a tile between cores, BIU, L2, and CHA.

tile, which means multiple tiles reading the same cache line will have their own private copy in their respective L2. However, all L2 caches are coherent; a write operation to a cache line from one tile will invalidate all other copies of that line in other tiles.

The cache coherency across all L2 caches is enforced by a distributed directory-based MESIF protocol that tracks the cache lines present inside all the tiles in *M*odified, *E*xclusive, *S*hared, *I*nvalid, or *F*orward state. The directory is distributed across all Knights Landing tiles with a portion stored in each tile in the box labeled *CHA*, for *Caching-Home Agent*, as shown in Fig. 4.2. While the CHA box physically resides in the tile, it is logically part of the Knights Landing on-die interconnect *mesh*. The portion of tag directory present in CHA does not have any connection to the lines stored in the colocated L2 cache; the cache line addresses are distributed among all the CHAs using an address hash.

CORE AND VPU

In this section, we describe the core and VPU architecture in more detail. As mentioned before, several major modifications were made to the original low-power Intel Atom processor core design, to create the Knights Landing core. List of some of these changes are below.

- Added four threads per core to provide higher-throughput performance.
- More than doubled the number of inflight instructions, that is, the Reorder Buffer (ROB) size, to 72 slots, to improve the single thread performance.
- Added vector instructions (AVX, AVX2, AVX-512) support.
- Added two VPUs to the main out-of-order pipeline.
- Increased size of the L1 data cache (DL1) to 32 KB to match the DL1 size in Knights Corner.
- Added two 64B loads ports in the DL1 to feed the two VPUs.
- Increased sizes for TLBs and added 1 GB page support.
- Implemented gather/scatter engine to insert the gather loads and scatter stores without consuming fetch and decode slots in the pipeline.
- Added support to complete quickly the accesses that are unaligned or split across multiple cache-lines.
- Doubled the L2 cache bandwidth to 1 cache line read and ½ cache line write per cycle.

Fig. 4.3 shows the block diagram of Knights Landing core and VPU. It is divided into five units: (1) front-end unit (FEU), (2) allocation unit (AU), (3) integer execution unit (IEU), (4) memory execution unit (MEU), and (5) VPU. While the core is 2-wide from the point of view of decoding, allocating, and retiring operations, it is capable of executing up to six operations per cycle, namely, two integer, two FP, and two memory operations. Fig. 4.4 shows a table with sizes for important structures in the core and how these structures are divided among different threads. Later in the chapter, we provide a detailed description of the core's threading support. The division of structures among threads will become clearer after reading that description.

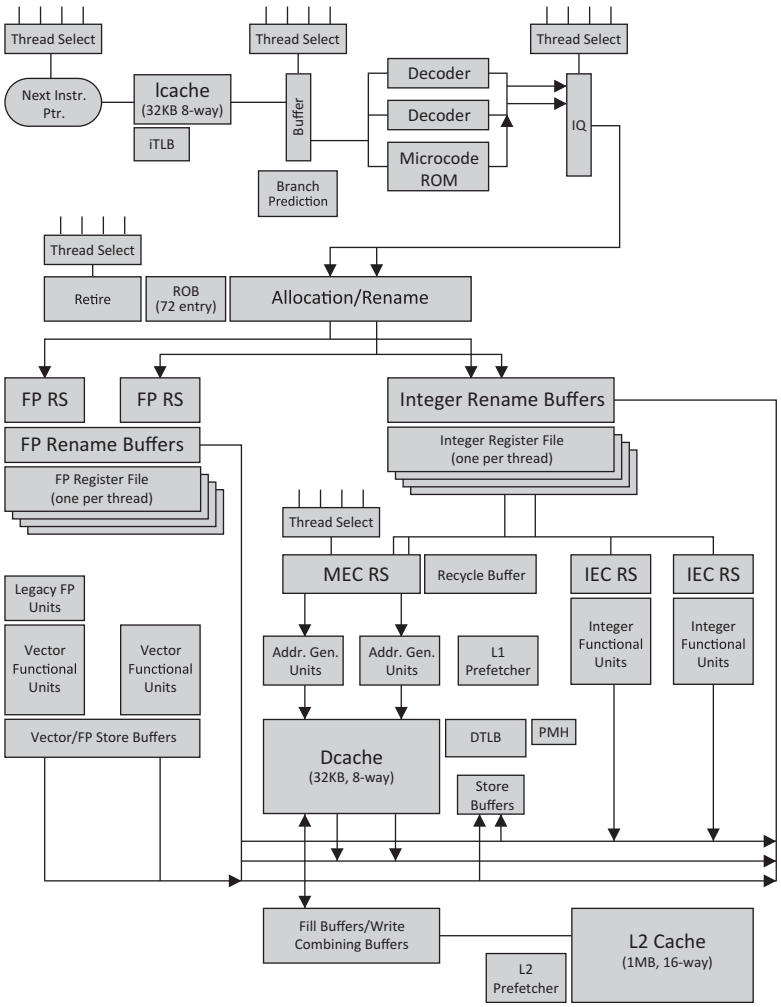


FIG. 4.3

Knights Landing core microarchitecture.

Front-end unit

The FEU is responsible for fetching instructions, decoding them, predicting branches, breaking instructions into simpler *micro-operations* (μops) and delivering them to the AU. Converting instructions into a flow of μops allows handling complex instructions without building complex hardware. Most instructions convert into single μop, however.

The process begins with the next instruction pointer block sending the next instruction to be fetched to the *Instruction Cache* (*Icache*). There is a thread selector at this point which decides which thread to pick for the next instruction fetch. Instruction

Structures	Sizes	Thread Sharing Policy
Instruction L1 Cache	32 KB, 8-way	Shared
iTLB	48	Shared
Instruction Queue	32	Dynamically Partitioned
ReOrder Buffer (ROB) Size (Out of Order Window)	72	Dynamically partitioned
Rename Buffers (RB)	72	Dynamically Partitioned
Register File (RF)	# of x86 architecture registers	Replicated
Store Data Buffers	16 entries	Dynamically Partitioned
Integer Reservation Station	12 x2	Dynamically Partitioned
Memory Reservation Station	12	Dynamically partitioned
Floating-point Reservation Station	20 x 2	Dynamically Partitioned
Data L1 Cache	32KB, 8-way	Shared
uTLB	64	Shared
DTLB	4K – 256 1M - 128 1G - 16	Shared
Inflight Gathers/Scatters Table	4	Dynamically Partitioned
Fill Buffers (outstanding misses per core)	12	Shared
L2 Cache	1MB, 16-way	Shared
Outstanding misses per tile	48 Reads and 32 Writebacks	Shared
L2 Prefetch Streams	48	Shared

FIG. 4.4

Table with sizes and thread-sharing policies for important structures.

bytes are read from the 32 KB, 8-way associative, Icache and written in an intermediate buffer. In parallel, the 48-entry *Instruction TLB* (*ITLB*) is accessed to obtain virtual to physical address translation. The thread is stalled in case of Icache or ITLB miss until the miss is resolved and other threads are selected for instruction fetch.

Instructions are selected from the intermediate buffer for decode. An intelligent thread selector ensures that instructions from all the active threads are selected in a fair manner, while taking into account the ability of that thread to make forward progress. Up to two instructions can be decoded every cycle. If an instruction is a complex instruction (i.e., multi-μop), then the *Microcode ROM* is accessed to obtain the corresponding μop flow for that instruction. Eventually, up to two μops are written per cycle in the *Instruction Queue* (*IQ*).

The *branch prediction unit* provides prediction on direction and target of branches. The unit consists of several types of predictors: Branch target buffer, Return Stack Buffer, Indirect predictor, and an enhanced form of gskew predictor.

Allocation unit

The AU is responsible for preparing μops for out-of-order execution. It assigns the necessary pipeline resources required by μops, such as ROB entries, Rename Buffer (RB) entries, Store Data buffers, gather/scatter table entries, and Reservation Station (RS) entries. It also renames the logical register sources and destinations in the μops to the RB entries. The RB provides storage for the results of in-flight μops until they retire, at which time these results are transferred to the architectural Register File (RF). After assigning the required pipeline resources and renaming the registers, the AU sends the μops to one of the three execution units, Integer, Memory, or Vector Processing, based on their type.

The AU is 2-wide, meaning it operates on two μops every cycle. It reads two μops from the IQ every cycle. A thread selector decides which thread is read from the IQ. The μops that do not have all the resources they need in order to proceed (i.e., a ROB entry, an RS entry, store data buffer entries) are stalled in the allocation until those resources become available. This does not prevent other threads, which may have available resources, from proceeding. The thread selector takes the stalls into account when reading μops from the IQ. This ensures we keep the downstream pipeline busy while some threads may be blocked for resources in AU.

The AU writes up to two μops in Integer, FP, or Memory RS per cycle.

Integer execution unit

The IEU executes integer μops, which are defined as those that operate on general-purpose registers R0–R15 (i.e., RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8…R15). There are two IEUs in the core. Each IEU contains 12-entry RS that issues one μop per cycle. The Integer RSes are fully out-of-order in their scheduling. Most operations have 1-cycle latency and are supported by both IEUs, but a few operations have 3- or 5-cycles latency (e.g., multiplies) and are only supported by one of the IEUs.

Memory execution unit

The MEU executes memory μops and services fetch requests for I-cache misses and ITLB misses. Up to two memory operations, either a load or a store, can be executed in the MEU per cycle. Memory operations are issued in-order from the 12-entry memory RS, but they can execute and complete out-of-order. The μops that were not able to complete successfully are placed into the Recycle Buffer and reissued to the MEU pipe once their conflict conditions are resolved. Completed loads are kept in the memory ordering queue, to maintain consistency, until they are retired. While stores are kept in the store buffer after address translation, they can forward data to any dependent loads. Stores are committed to the memory in the program order, one per cycle.

The 64-entry, 8-way set-associative, L1 μTLB is backed up by the 256-entry, 8-way set-associative, L2 DTLB. The DTLB also contains an 8-way, 128-entry table for 2 MB pages and a fully-associative, 16-entry table for 1 GB pages. The writeback, nonblocking, 32 KB, 8-way set-associative DL1 supports two simultaneous 512-bit reads and one 512-bit write, with a load-to-use latency of four cycles for integer loads and five cycles for FP loads. The L1 hardware prefetcher monitors memory address patterns and generates data prefetch requests to the L2 cache to deliver cache lines in advance.

The MEU supports unaligned memory access without any penalties and supports accesses that are split across two cache lines with a two-cycle penalty. The fast unaligned and cache split accesses help workloads where memory accesses may not always be aligned to natural data type sized boundaries.

The MEU supports 48 virtual address bits and 46 physical address bits (up to 39 physical bits for addressing cacheable memory and the rest for uncacheable memory) to provide the large memory addressing capability commensurate with a standalone, standard Intel® Architecture (IA) processor.

The MEU contains specialized logic to handle gather and scatter instructions efficiently. A single gather/scatter instruction can access multiple memory locations. In Knights Landing, such multiple accesses are generated by a special engine close to the L1 cache pipeline, instead of being a series of μops that flow through the entire core pipeline. This allows for executing the gather and scatter instructions while consuming minimal resources in rest of the core (e.g., FEU, AU, RSs, and Retire), allowing other types of μops to make forward progress in parallel.

Vector processing unit

The VPU is the vector and FP arithmetic execution unit of Knights Landing and is responsible of providing support for x87, MMX, SSE, AVX, and AVX-512 instructions, as well as integer divides. There are two VPUs connected to the core. These are tightly integrated into the pipeline, with AU dispatching instructions directly into the VPUs. The VPUs are mostly symmetrical, and each is able to provide a steady-state throughput of one AVX-512 instruction per cycle, providing a peak of 64 SP or 32 DP FP operations per cycle from the pair of VPUs available per core (i.e., 2 VPUs per core, 1 AVX-512 instruction per cycle per VPU, FMA offers 16 DP, or 32 SP operations per instruction using AVX-512 registers that are 8 DP or 16 SP wide,

$2 \times 1 \times 16 = 32$ DP or $2 \times 1 \times 32 = 64$ SP). One of the VPUs is extended to provide support for the legacy FP instructions, such as x87, MMX, and a subset of byte and word SSE instructions.

Each VPU contains a 20-entry FP operation RSs that issues out-of-order one μop per cycle. The FP RS are different from the IEU RS and MEU RS in that they, to help reduce their size, do not hold source data; the FP μops read their source data from FP RB and FP RF after they issue from the FP RS, spending an extra cycle between RS and execution compared to integer and memory μops. Most FP arithmetic operations have a latency of six cycles, while other arithmetic operations have a latency of two or three cycles, depending on the operation type.

The VPU also supports the new AVX-512 transcendental and reciprocal instruction extensions, that is, AVX-512ER, and vector conflict detection instruction extensions, that is, AVX-512CD, introduced in Knights Landing.

THREADING

The core supports four threads of execution. These are implemented as *hyper threads* where the instructions from all threads flow through the pipeline simultaneously. Thread-selection points in the pipeline select from which thread instructions move forward. This decision is made each cycle and takes into account such factors as thread fairness, instruction availability, thread-specific stalls, and other constraints, to help maximize the utilization of hardware.

Fig. 4.4 details how the threads share some of the key core structures. Structures are shared by threads, dynamically partitioned among them, or replicated for each thread. Dynamically partitioned structures readjust their partitions as threads go to sleep or are awoken so that the active threads get to use the full physical structure. For example, when one thread is active, the thread will use the full 72 entries in the ROB. However, when two threads are active, then each thread will use 36 entries, and when three or four threads are active, each thread will use 18 entries. Shared resources do not enforce partitioning; threads get shared resources on first-come-first-served basis. Replicated resources have dedicated copies for each thread, irrespective of whether they are active are not. Very few structures in the core are replicated per thread.

Four threads are not required to be active per core for performance. Threading can be turned off with BIOS settings, in which case there will be only one thread per core. But if threading is on, then it is a software decision about how many threads to start on a core. Software can start 1, 2, 3, or all 4 threads. Threads that are not active stay in HALT state or in monitor mwait state. In many cases, unlike Knights Corner, a single active thread per core is sufficient to utilize all the core resources. (Knights Corner required at least two active threads per core.) Using two active threads is often the best general choice as the out-of-order processing can often hide instruction and cache access latencies for many workloads. Four active threads per core can provide additional significant benefit for memory latency sensitive workloads and for applications with strong thread scaling and datasets that do not increase per thread.

As is explained more in Chapter 6, three active threads per core will generally underperform other options.

L2 ARCHITECTURE

On a Knights Landing tile, the two cores share a 16-way associative, 1 MB unified L2 cache (Fig. 4.1). Intratile coherency is maintained by the BIU, which also acts as the local shared L2 cache management unit (Fig. 4.2). Lines in the L2 cache are maintained in one of the MESIF states. The cores make requests to the BIU via a dedicated request interface to each core. Cacheable requests lookup L2 tags, while other requests are bypassed directly out to the mesh to be serviced by a specific CHA as determined by the built-in address hash.

Knights Landing implements a unique cache topology to minimize coherency maintenance traffic. The L2 cache is inclusive of DL1 but is not inclusive of IL1. The lines brought into IL1 fill into the L2, but when those lines are evicted from L2, the corresponding IL1 lines are not invalidated. This avoids invalidations due to Hot-IL1-Cold-L2 scenarios, where a line in active use in IL1 gets invalidated due to eviction of the corresponding line due to inactivity from L2. The "presence" bits per line are stored in L2 to track which lines are in active use in DL1. This information is used to filter the coherency probes for inclusive DL1 when not applicable. It also factors into L2 victim selection algorithm to minimize eviction of in-use lines.

BIU also contains a L2 Hardware Prefetcher that trains based on requests coming from the cores. The BIU supports up to 48 independent prefetch streams. Once a stream is detected to be stable, in either the forward or backward direction, prefetch requests are issued to successive cache lines in that stream (i.e., unit-cache line stride distance).

CLUSTER MODES

In this section, we describe the cluster modes on Knights Landing, which we introduced in Chapter 2, in further detail. Cluster modes are ways to divide the chip into separate virtual regions with the intention of keeping the on-die communications to be as local as possible. The goal is to lower the latency and increase the bandwidth of on-die communications. We support three primary cluster modes: (1) All-to-all mode, (2) Quadrant mode, and (3) Sub-Numa Cluster (SNC) mode. The only issue is performance; software will run in any cluster mode, and cache-coherency is always maintained across the entirety of a Knights Landing.

These modes can be best understood by observing how the data flow required to service an L2 cache miss would vary between the different modes. A typical L2 miss flow is shown in Fig. 4.5 (which is also the figure that describes all-to-all cluster mode). There are three main agents that participate in an L2 miss flow: (a) the tile that generates the miss, (b) the tag directory (as part of CHA) that "owns" the missing address and tracks whether any other tile on the chip has that address in its caches, and lastly, (c) the memory that supplies the data for servicing the miss. On an L2 miss, the tile sends the request to the CHA to check if any other tile on the chip has that address in its

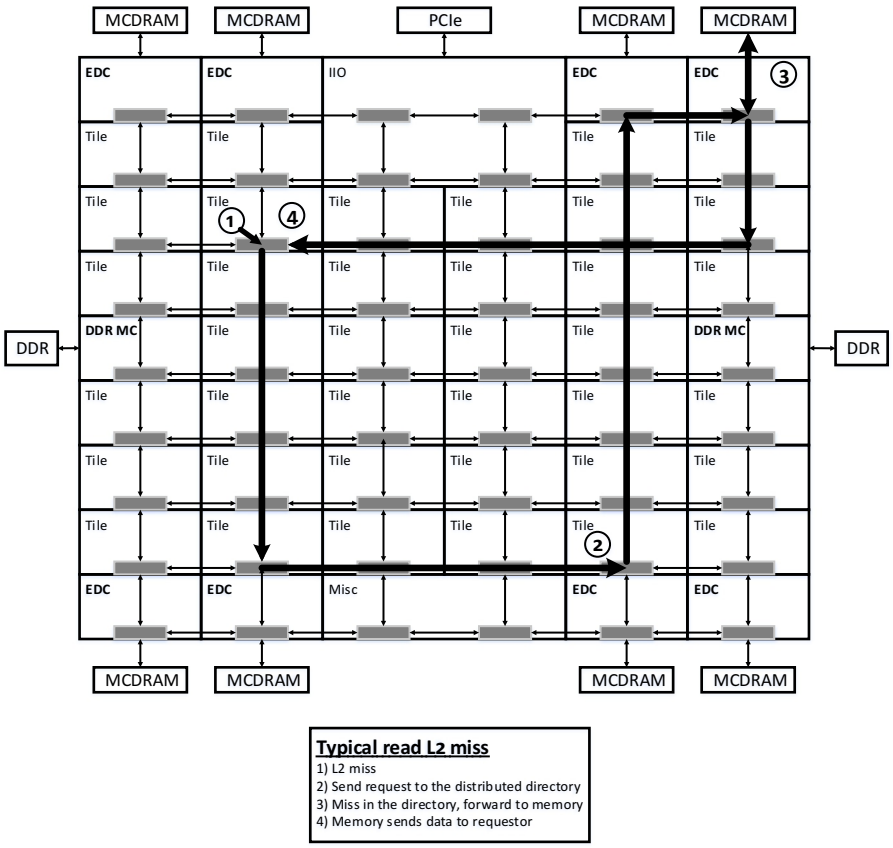


FIG. 4.5

An L2 cache miss flow example in all-to-all cluster mode.

caches. If the CHA finds that no other tile has the address cached, then it sends the request to the memory. The memory reads the data and sends it to the requesting tile. If the CHA finds that some other tile has the data in its caches, then it will send the request to that tile to send the data to the tile that had the miss (that case is not illustrated in the figure). The three cluster modes create different levels of "affinity" between the three agents (i.e., tile, CHA, and memory) to localize the communication between them.

The cluster modes are chosen via BIOS options and can only be set once during boot time. The cluster mode cannot be changed again until after the next system reset.

ALL-TO-ALL CLUSTER MODE

The all-to-all cluster mode (Fig. 4.5) is the most general among all three modes. This mode will be lowest in general performance than the other memory modes, but it can be used with any DDR DIMM configuration. It is the only mode which can be used

when DDR DIMMS are not identical in capacity. In this mode, there is no affinity between the tile, CHA, and the memory. The addresses are uniformly hashed across all CHAs and over all memory, independently. Any tile may request data at an address that is tracked by a CHA in any part of the chip, and the memory location may reside in any part of the chip. A L2 miss transaction will traverse longer distances, on average, across the chip to get the data in all-to-all mode.

The purpose of this mode is to offer flexibility in supporting any combination of allowed DDR DIMM configurations. This mode can route physical addresses from any tile to any CHA, to any memory controller. While not the default, this mode is automatically chosen in the event of any memory asymmetries or other configuration irregularities are detected in the system at boot time.

QUADRANT CLUSTER MODE

The quadrant mode (Fig. 4.6) will be the mode in which most software would want to run. This mode is expected to be the default mode set in the BIOS although all-to-all mode will be entered if DDR DIMMs are not found to be identical in capacity. This mode is transparent to software, in that software does not need to take any specific actions to benefit from this mode. In this mode, we establish affinity between the two of the three agents, that is, the CHA and the memory. The chip is divided into four virtual quadrants. The memory addresses are hashed such that the CHA that has the tag directory for a memory address is in the same quadrant as the memory where data for the address resides. This way a memory access transaction, as shown in Fig. 4.6, travels locally within the same quadrant after checking the tag directory in the CHA to reach the memory. This reduces the latency of memory accesses compared to all-to-all mode. Since reduced latency allows hardware buffers involved in memory access transactions to be freed sooner, the quadrant mode allows more memory transactions to occur in a given time; therefore it offers higher bandwidth than all-to-all mode.

The only requirement for using quadrant mode is that the memory configuration be symmetric, that is all DDR DIMMS be of the same capacity. This is needed for the address hashing to work to enable the quadrant division. Once booted in quadrant mode, software does not need to do anything special to benefit from this mode.

SNC-4 MODE

The SNC mode (Fig. 4.7) provides the shortest latency and most localized communications among all modes. It extends the two-agent affinity in the quadrant mode to three-agent affinity which includes all three agents: the tile, the CHA, and the memory.

In this mode, each quadrant is exposed as a separate cache-coherent NUMA cluster that is visible to operating system, via BIOS ACPI tables. The memory addresses are distributed such that a continuous region of memory is mapped to each cluster and the tag directories for memory addresses mapped in a cluster are in CHAs that are also all within the same cluster.

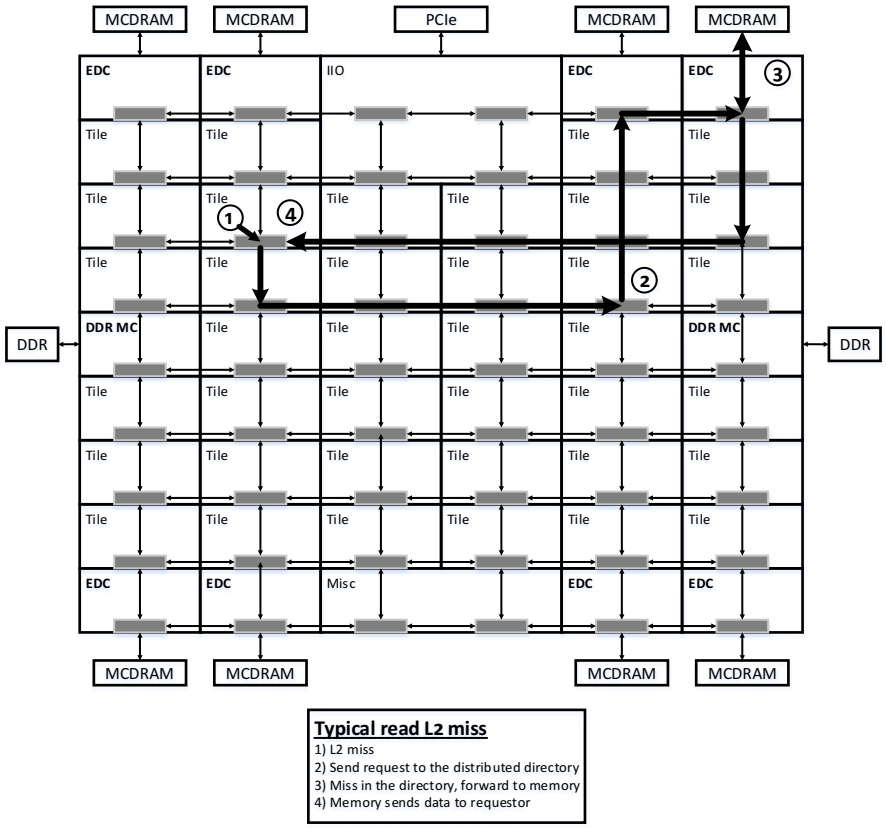


FIG. 4.6

An L2 cache-miss flow example in quadrant cluster mode.

A NUMA-optimized software (i.e., software that allocates memory from the same cluster where it runs) will see all its memory access transactions complete locally so that they will stay entirely within the same cluster as shown in Fig. 4.7. Such memory accesses from a tile will go to a CHA that is in the same cluster and, thereafter, will go to a memory which is also in that cluster. This local communication enables shortest communication latencies in this mode. As with quadrant mode, SNC-4 mode enables higher bandwidths due to better utilization of hardware buffers that support memory transactions resulting in higher bandwidth than all-to-all and quadrant mode.

Unlike quadrant and all-to-all modes, this mode is not entirely software transparent. The software needs to be NUMA optimized to benefit from this mode. The only issue is performance; software will work in any cluster mode including SNC. However, software that is not NUMA optimized is most likely to run better in quadrant mode.

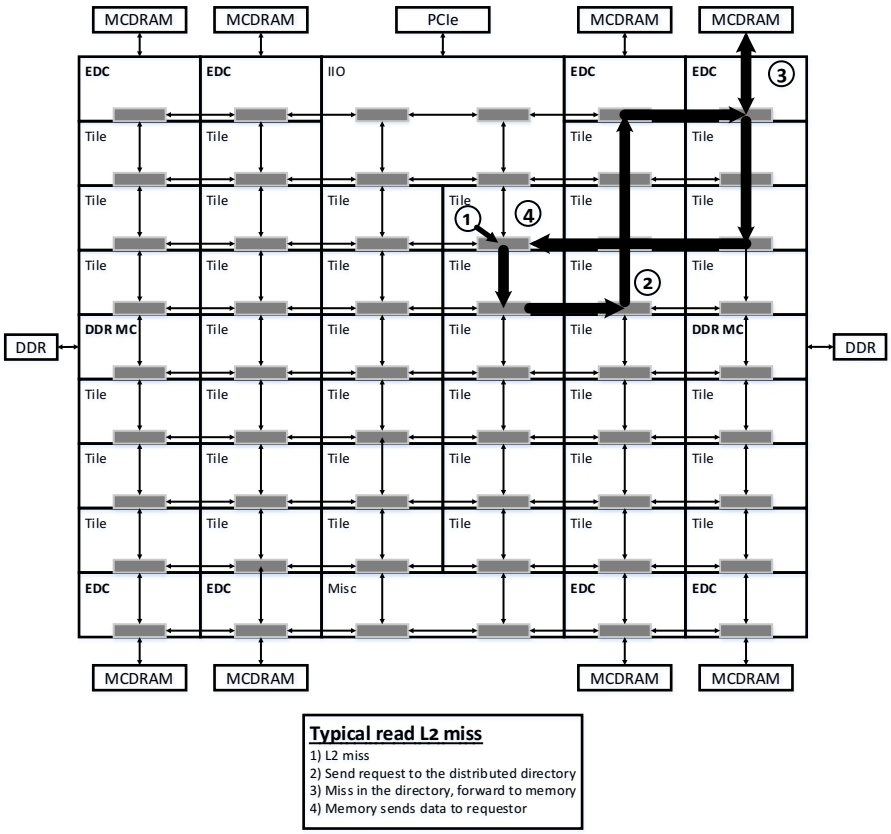


FIG. 4.7

An L2 cache-miss flow example in sub-NUMA cluster mode.

Knights Landing in SNC-4 mode with four clusters will look analogous to a four socket Intel Xeon processor-based machine (Fig. 4.8) in terms of NUMA nodes and memory allocation. In fact, any question on how a particular scenario will work in SNC-4 mode on Knights Landing can be answered by asking how that same scenario will be handled in a multisocket Intel Xeon processor-based machine.

HEMISPHERE CLUSTER AND SNC-2 MODES

For the quadrant and SNC-4 modes, we described the ability to divide Knights Landing into four parts (quadrants). Knights Landing also supports variations that divide into hemispheres (two parts) instead of quadrants (four parts). Everything described about quadrant and SNC-4 modes is identical, excepting that the Knights Landing is logically divided into two regions instead of four. The average latencies will be higher due to the increased average distances for resolving memory requests.

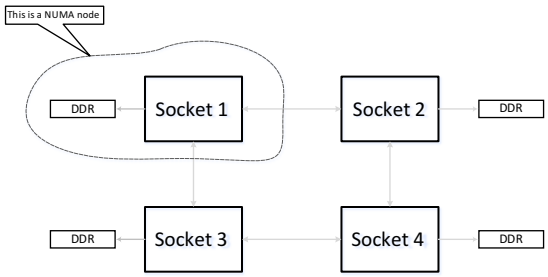


FIG. 4.8

A four socket with four NUMA nodes using Intel Xeon processors. This is analogous to Knights Landing with SNC-4 mode with four clusters.

Since the latencies are higher than quadrant and SNC-4 modes but lower than all-to-all mode, the memory bandwidth is higher than all-to-all mode but lower than quadrant and SNC-2 modes. The most interesting mode would seem to be the SNC-2 mode offering NUMA clusters equal to half the cores mimicking a dual socket processor platform, but the lower bandwidth probably makes this mode less interesting than quadrant or SNC-4 mode.

CLUSTER MODE SUMMARY

Fig. 4.9 summarizes the different characteristics of the three cluster modes in a table. The table in Fig. 4.9 also describes *memory interleaving* in each mode, which we expand upon in the following section.

MEMORY INTERLEAVING

Memory interleaving is a technique to spread out consecutive memory access across multiple memory channels, in order to parallelize the accesses to increase effective bandwidth. The interleaving granularity is always a cache line, which is 64 bytes for Knights Landing; this is the same size cache line used by all other current Intel processors. In this section, we describe how memory interleaving works with the various cluster modes.

In all-to-all, quadrant and hemisphere modes, addresses are uniformly distributed across the memory channels, as shown in Fig. 4.10. The distribution pattern is different for each mode as it depends on the specific hash function used in each mode to assign memory addresses to different CHAs. The net effect is that the addresses are uniformly distributed across the memory channels. In flat memory mode, contiguous ranges of memory are assigned to DDR and MCDRAM, respectively, with the MCDRAM range above the DDR range. The addresses in the DDR memory range are uniformly distributed among the DDR channels, while the addresses in the

Cluster Mode	Transaction Routing	Memory Interleaving	NUMA	DIMM (DDR4) Configuration
All-to-all	Originate from any tile, route to any tag directory, to any MC	Single address space uniformly interleaved across all MCs	None	All configurations are possible regardless of DDR population characteristics
Quadrant/ hemisphere	Originate from any tile, to a tag directory colocated in the same quadrant/hemisphere as the memory	Single address space uniformly interleaved across all memory channels	None	Equally populated equal capacity DDR DIMMs installed
SNC-4/ SNC-2	Tile, tag directory, and target MC are all colocated within the same cluster. SNC-4: 4 clusters SNC-2: 2 clusters	Contiguous address space per cluster, uniform interleaving within cluster	SNC-4: 4 nodes SNC-2: 2 nodes	Equally populated equal capacity DDR DIMMs installed

FIG. 4.9

Cluster mode summary.

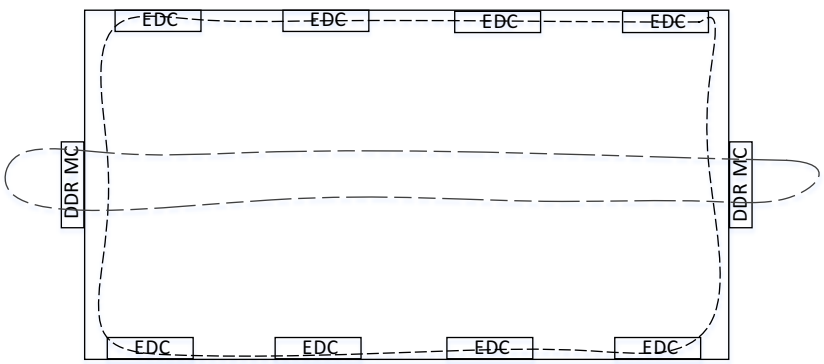


FIG. 4.10

Memory interleaving in all-to-all, quadrant, and hemisphere cluster modes with flat memory mode.

MCDRAM memory range are uniformly distributed among the MCDRAM channels, as shown in Fig. 4.10. In cache memory mode, since only DDR memory is visible to software (as MCDRAM is the cache), the entire memory range is uniformly distributed among the DDR channels. The address distribution in hybrid memory mode is

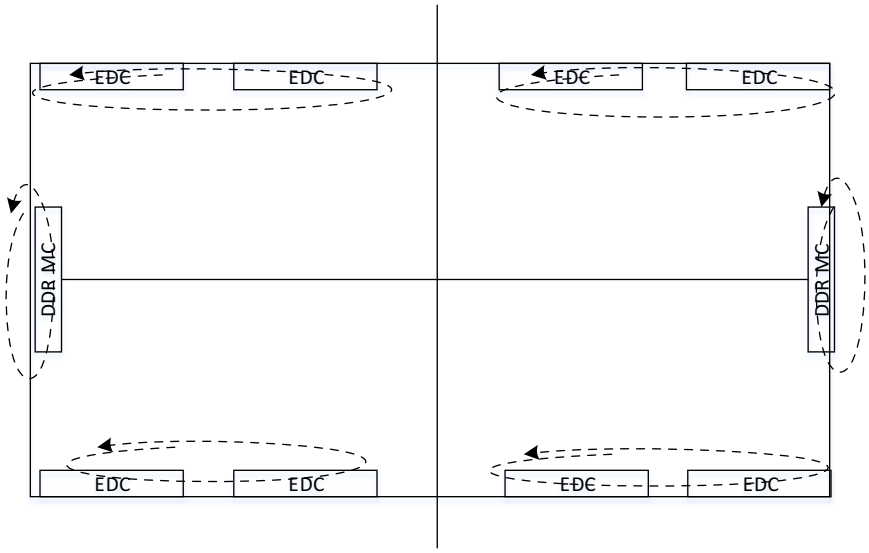


FIG. 4.11

SNC-4 memory interleaving for flat memory mode.

similar to the distribution patterns of flat memory mode and cache memory mode for the address ranges that are mapped as flat and cache, respectively.

In SNC-4 and SNC-2 cluster modes, contiguous regions of memory are assigned to each cluster (also a NUMA node) and are cache line interleaved among the memory channels within that NUMA node, as shown in Figs. 4.11 and 4.12, respectively. In flat memory mode, the memory region mapped to each SNC cluster is divided into two contiguous portions, one for MCDRAM and other for DDR. These portions are interleaved over the MCDRAM and DDR channels that are in that cluster (for SNC-4, since DDR channels are not entirely within a cluster, the interleaving is over all the three channels that are closer to the cluster; this looks similar to SNC-2). In cache memory mode, the addresses are interleaved over the DDR channels, since MCDRAM is a cache and is hidden behind the memory. In hybrid mode, the address interleaving will be similar to flat and cache mode based on whether the memory region is mapped to flat or cache portion of the memory.

MEMORY MODES

As mentioned in Chapter 2, Knights Landing supports three memory modes for configuring the MCDRAM and DDR memory (Fig. 4.13): (1) cache mode, (2) flat mode, and (3) hybrid mode. In cache mode, MCDRAM is configured as a cache for the DDR memory; in flat mode, MCDRAM is configured as memory just like DDR

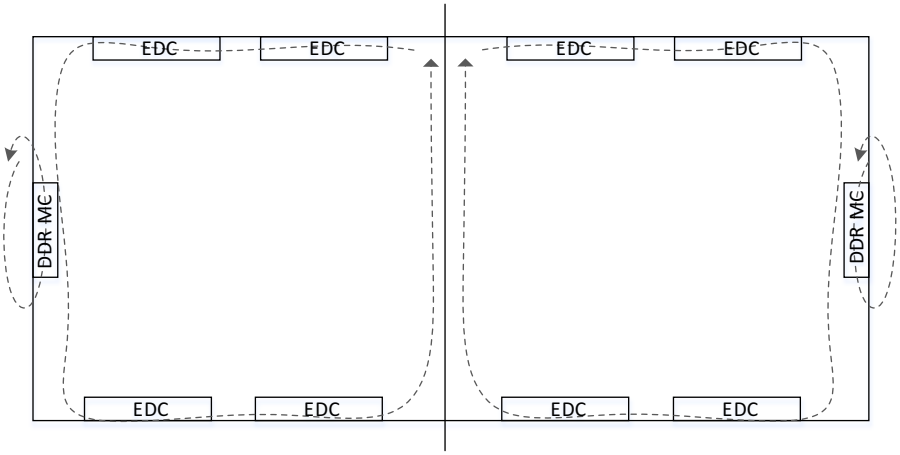


FIG. 4.12

SNC-2 memory interleaving in flat memory mode.

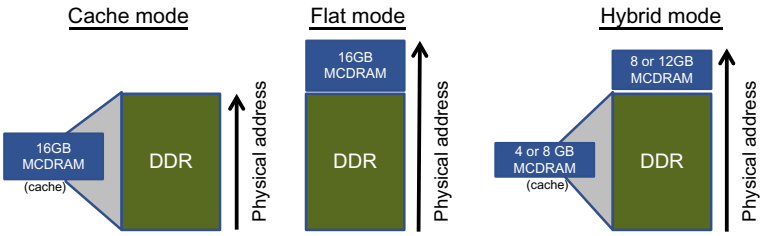


FIG. 4.13

Three modes for MCDRAM usage and how they related to DDR memory.

and is part of the same address space; and in hybrid mode, a portion of MCDRAM is configured as cache and the remainder is configured as flat in the same address space as DDR. Like cluster modes, the memory modes are also selected at boot time via BIOS options and can only be set once during boot time. The memory mode cannot be changed again until the next system reset.

CACHE MODE

In cache mode, MCDRAM is configured as a cache for the DDR memory. The MCDRAM cache is completely hardware managed, requiring no software enablement. Legacy, unmodified software can use this mode and benefit from the high bandwidth provided by MCDRAM. The benefits will, of course, depend on the hit rate observed by the software.

MCDRAM cache is a direct-mapped cache with 64-byte cache lines. The tags are stored in the MCDRAM, as part of extra bits available in each cache line. These extra

bits are read at the same time as the cache line, thereby allowing tag and data to be read on the same access, and hence, enabling the determination of a hit or miss in cache without needing a separate read access. Since MCDRAM size is much larger than traditional caches, in most cases there should be no significant change in conflict misses due to the direct-mapped policy (see Chapter 6 for further discussion on direct-mapped policy).

In this mode, all requests first go to the MCDRAM for a cache look-up and then, in case of a miss, a request is sent to DDR. The memory accesses will never bypass the MCDRAM and go to the DDR. On a miss, the data is read from the DDR and sent to the MCDRAM, to fill the cache, and the requesting tile, simultaneously.

MCDRAM cache is a *memory-side cache*, as opposed to *CPU-side caches* such as an L1, L2, or last level caches, in that a memory-side cache is closer to memory in terms of its properties as compared to CPU-side caches on the cores or tiles. It acts more like a high-bandwidth buffer sitting on the way to memory, exhibiting memory semantics, instead of a cache sitting closer to the cores. Unlike caches on the cores or tiles, MCDRAM cache does not need to be snooped by external transactions since any access to memory first goes through MCDRAM cache. An "uncacheable" memory type that does not allocate in core cache will allocate in MCDRAM cache, since MCDRAM cache is but a part of memory, just higher bandwidth.

MCDRAM cache is made inclusive of all modified lines in the L2 caches. In other words, if a cache line resides in a modified state in an L2 cache, it must also be in MCDRAM. This ensures that when a modified line is written back from an L2 cache, there is no need to query the MCDRAM cache first before writing the line into it. There is no risk of overwriting a different line, since with the modified-inclusive property, the line being written back is guaranteed to be present in the MCDRAM cache. To maintain this modified-inclusive property, before a line is evicted from MCDRAM, a snoop is sent to check if a *modified* copy of that line exists in L2 cache and, if so, downgrade it from *modified* to *shared* by forcing a writeback of the modified line. The line does not get evicted from the cache.

As mentioned before, the MCDRAM cache mode is completely transparent to software and will work out of the box on unmodified code. The benefits derived from the MCDRAM cache will depend on the hit rate seen by software. The cache works well for many applications, but for applications that do not show good hit rates in the MCDRAM, the other two memory modes provide more control for applications to directly manage their use of MCDRAM (also see Chapter 3).

FLAT MODE

In flat mode, both MCDRAM and DDR are presented as regular memory mapped in the same system address space. There is no difference between the MCDRAM region and the DDR region in terms of semantics; both act as regular load/store memory. Unlike cache mode, flat mode is not software transparent. The software needs to explicitly allocate best suited data in the MCDRAM for it to benefit from the provided high bandwidth. However, once data is allocated in MCDRAM, software will

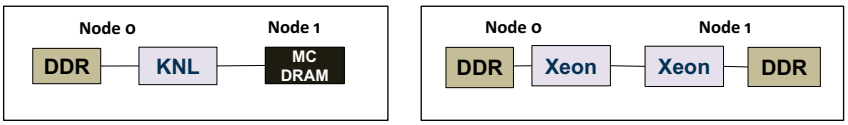


FIG. 4.14

Analogy between Knights Landing (KNL) in flat mode and an Intel Xeon processor-based machine with two sockets. Both have two NUMA nodes.

see stable high bandwidth access to that data, since there is no dependency on "hit" rate like in cache mode.

For software to allocate explicitly into MCDRAM, we enabled a way to "name" the MCDRAM region such that we keep software portable to processors that do not provide MCDRAM. The naming scheme exposes MCDRAM and DDR as two separate NUMA nodes. If a Knights Landing system has only DDR or only MCDRAM, then there would be only one NUMA node. SNC-2 and SNC-4 complicate this slightly by multiplying the number of NUMA nodes by two or four, respectively. This is covered in more detail toward the end of this chapter in the *Interactions of Cluster and Memory Modes* section.

The software can then allocate into MCDRAM referring to the NUMA node assigned to MCDRAM. Knights Landing in flat mode, with two NUMA nodes, looks analogous to a two-socket Intel Xeon processor-based machine with two NUMA nodes (Fig. 4.14). In fact, almost in all cases we can answer a question on how a certain memory allocation scenario would work in flat mode by referring to how a similar scenario would work for the two NUMA nodes in a two-socket Intel Xeon processor-based machine.

As described in Chapter 3, we provide the *memkind* software library that contains functions for managing MCDRAM memory. This library uses the NUMA node information to do the allocation into MCDRAM. The code written with this library remains portable, in that the functions resort to standard memory allocation when code is run on a system that does not have MCDRAM memory or is in MCDRAM cache mode.

There are two modes that are degenerate forms of the flat mode: MCDRAM-only mode and DDR-only mode. In these cases, there is only one type of memory present in the system, that is, MCDRAM or DDR, but not both. Since these modes are similar to standard systems with one type of memory, no software modification is needed; programs will run by allocating all their memory in MCDRAM or DDR, whichever is available.

HYBRID MODE

The hybrid mode, as the name suggests, is a combination of the cache mode and the flat mode. A portion of MCDRAM is configured as a cache (25% or 50%) and the remaining is configured as flat memory (75% and 50%). This is ideal for applications

	Capacity	Bandwidth	Idle Latency
MCDRAM	Up to 16GB	Up to 450GB/sec	Approx. 150ns
DDR	Up to 384GB	Up to 90GB/sec	Approx. 125ns

FIG. 4.15

MCDRAM and DDR characteristics.

that benefit from general caching and can also take advantage of high bandwidth memory by storing critical or frequently accessed data in flat memory. The portion of MCDRAM that is used as cache serves all of DDR memory. Using the cache portion of MCDRAM does not require any application changes. The flat memory portion of MCDRAM will get exposed to software as a separate NUMA node, like in flat mode, and the application will need to use the memkind library or other NUMA-based allocators to use this portion of MCDRAM. As with flat mode, SNC-2 and SNC-4 complicate this slightly by multiplying the number of NUMA nodes by two or four, respectively. This is covered in more detail toward the end of this chapter in the *Interactions of Cluster and Memory Modes* section.

Hybrid is an excellent choice to allow benefiting from cache mode, without software changes, but allowing experimentation with flat mode to get better performance. It will also be useful for systems that are used by variety of users with different types of applications, some that may benefit from flat mode optimization, while others may do fine with cache mode. Since changing modes requires a reboot, it may be convenient to boot the system in hybrid mode, thereby providing both cache mode and flat mode support to the users (albeit with a smaller capacity for each compared to all cache or all flat mode).

CAPACITY, BANDWIDTH, LATENCY

Fig. 4.15 compares the capacity, streams bandwidth, and the idle latencies of MCDRAM and DDR memories.

INTERACTIONS OF CLUSTER AND MEMORY MODES

The cluster modes and the memory modes are selectable completely independently of each other; any of the three cluster modes can be selected with any of the three memory modes. There are however some interactions between SNC cluster modes and the hybrid/flat memory mode that are worth understanding. These interactions arise because both these modes use NUMA enumeration: SNC uses NUMA nodes to partition the chip, while the flat memory mode uses NUMA nodes to distinguish between DDR and MCDRAM memory. Thus, while the SNC-4 mode exposes four

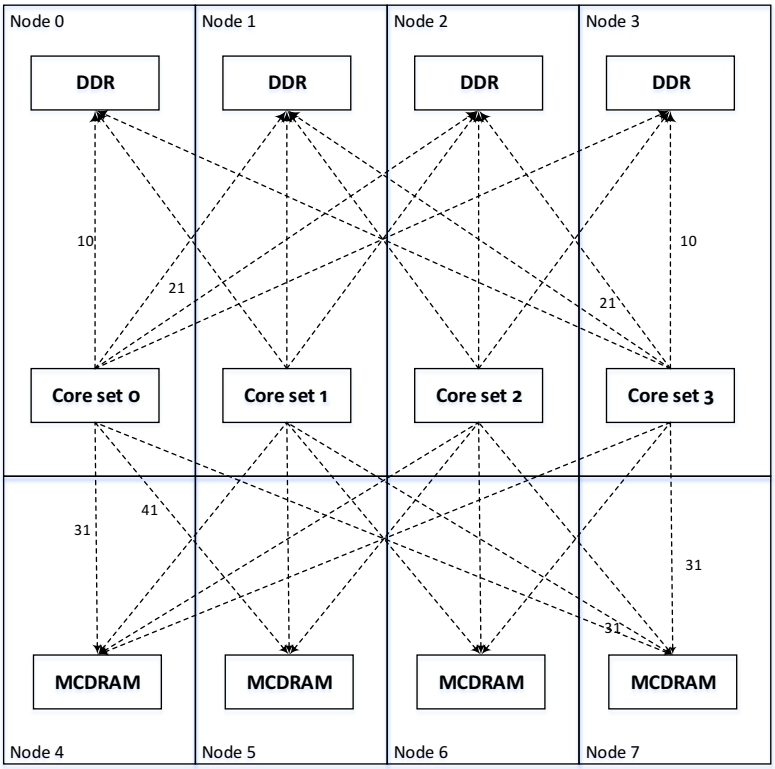


FIG. 4.16

Flat mode with SNC4.

NUMA nodes and the flat memory mode exposes two NUMA nodes (assuming MCDRAM and DDR are both present), when used together they expose eight NUMA nodes. This is illustrated in Fig. 4.16. The arrows with values 10, 21, 31, and 41 show the default *distance* values between the various NUMA nodes as programmed in BIOS. These distance settings determine the affinity between different NUMA nodes.

The hybrid mode and SNC-4 interaction will look the same as in Fig. 4.16 from the NUMA node point of view. The hybrid mode has a portion of MCDRAM as cache, but that does not change the NUMA topology. Similarly, the interactions of flat mode and hybrid mode with SNC-2 will result in four NUMA nodes in all: two for the memory and two for the SNC-2 clusters. Fig. 4.17 enumerates the key cluster and memory mode combinations and shows their various attributes.

BIOS Setup Option			Additional Information	
Cluster Mode	**Memory Mode**	**MCDRAM Cache Size**	**NUMA Nodes**	**DIMMs Configuration Allowed**
All-to-all	**Cache**	MCDRAM 100% cache, 0% as memory	None	All configurations are possible regardless of DDR population characteristics.
	Flat	MCDRAM 0% cache, 100% as memory	Total 2: 1 node for MCDRAM, 1 node for DDR	
	Hybrid	MCDRAM 25% or 50% as cache, and 75% or 50% as memory		
Quadrant or Hemisphere	**Cache**	MCDRAM 100% cache, 0% as memory	None	Both memory controllers have same number of equal capacity DIMMs installed.
	Flat	MCDRAM 0% cache, 100% as memory	Total 2: 1 node for MCDRAM, 1 node for DDR	
	Hybrid	MCDRAM 25% or 50% as cache, and 75% or 50% as memory		
SNC-4 (with notes for SNC-2)	**Cache**	MCDRAM 100% cache, 0% as memory	Total 4: 4 clusters, each with 1 node for DDR (SNC-2: total 2)	Both memory controllers have same number of equal capacity DIMMs installed.
	Flat	MCDRAM 0% cache, 100% as memory	Total 8: 4 clusters, each with 1 node for MCDRAM and 1 node for DDR (SNC-2: total 4)	
	Hybrid	MCDRAM 25% or 50% as cache, and 75% or 50% as memory		

FIG. 4.17

Memory and cluster mode combinations, assuming systems with MCDRAM and DDR both.

SUMMARY

In this chapter, we described the details of the Knights Landing architecture. We dove deeper into the details of the tile architecture, described the various cluster and memory modes, and explained the interactions between them.

FOR MORE INFORMATION

- "Knights Landing: 2nd generation Intel® Xeon Phi™ Product," Hot Chips Special Issue of IEEE Micro Magazine, 36.2 (2016): 34–46. http://doi.ieeecomputersociety.org/10.1109/MM.2016.25.

CHAPTER

Intel Omni-Path Fabric

5

What is new with Knights Landing in this chapter?

Some versions of Knights Landing integrate Intel® Omni-Path Fabric in the same package (a.k.a. "on-package").

The Intel® Omni-Path Architecture (Intel® OPA) interconnect fabric design enables a broad class of multiple node computational applications requiring scalable, tightly coupled processing, memory, and storage resources. Furthermore, options for close "on-package" integration between Intel® OPA family devices, Intel® Xeon® processors, and Intel® Xeon Phi™ (Knights Landing) processors enable significant system-level packaging and network efficiency improvements. When coupled with recent user-focussed open standard Application Program Interfaces (APIs) developed by the Open Fabrics Alliance (OFA) Open Fabrics Interface (OFI) workgroup, Host Fabric Interfaces (HFIs; known as NICs in Ethernet), and switches in the Intel OPA family, systems are optimized to provide low latency, high bandwidth, and high message rate needed by large-scale high-performance computing (HPC) applications. Intel OPA provides important innovations for a multi-generation, scalable fabric, including link layer reliability, extended fabric addressing, and optimizations for many-core processors such as Knights Landing. High-performance datacenter needs are also a core Intel OPA focus, including: link-level traffic flow optimization to minimize datacenter-wide jitter for high-priority packets, robust partitioning support, quality of service (QoS) support, and a centralized fabric management system.

OVERVIEW

Intel OPA delivers a next-generation fabric with heritage from the Intel® TrueScale product line and the Cray Aries interconnect. See the *For More Information* section, at the end of this chapter, for resources on these earlier interconnects. Intel OPA integrates fabric components with processor and memory components enabling the low latency, high bandwidth, dense systems required for next-generation datacenters. Fabric integration takes advantage of processing, cache and memory subsystems, and communication infrastructure locality enabling rapid hardware innovation. Enhancements found in the first generation of Intel OPA include higher overall bandwidth, lower latency, and denser form factor systems.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00005-3

Intel OPA-based first-generation products focus on HPC systems including large supercomputing environments. Nevertheless, Intel OPA is generally applicable to any class of datacenter-level computation requiring scalable, tightly coupled, CPU, memory, and storage resources. Intel OPA defines an Open Systems Interconnect (OSI) layers 1 and 2 architecture that provides connectivity between elements in energy efficient supercomputer systems (e.g., based on Intel Xeon Phi processors), mission critical enterprise computer systems (e.g., based on Intel Xeon processors), and inexpensive datacenter servers (e.g., based on Intel® Atom™ processors).

To enable the largest scale systems in both HPC and the datacenter, fabric reliability is substantially enhanced by combining the link-level retry typically found in HPC fabrics, with the conventional end-to-end retry used in traditional networks. Layer 2 network addressing is extended to account for systems with over 10 million endpoints, thereby enabling use on the largest scale datacenters for years to come. To enable support for a breadth of topologies, Intel OPA provides mechanisms for packets to change Virtual Lanes (VLs) as they progress through the fabric. In addition, higher priority packets are able to preempt lower priority packets to provide more predictable system performance, especially when multiple applications are running simultaneously. Finally, fabric partitioning is provided to isolate traffic between jobs or between users.

The software ecosystem includes three key APIs.

1. The OFA OFI represents a long-term direction for high-performance, user-level and kernel-level network APIs.
2. The Performance-Scaled Messaging (PSM) API provides HPC focussed transports and an evolutionary software path from the Intel TrueScale fabric.
3. OFA Verbs provides support for existing applications and includes extensions to support the Intel OPA fabric manager.

Higher level communication libraries, such as the Message Passing Interface (MPI — see Chapter 15), and Partitioned Global Address Space (PGAS — see Chapter 16) libraries are layered on top of these low-level OFA APIs. See the *For More Information* section for resources on these APIs and more in depth treatments of the software ecosystem supporting Intel® OPA.

HOST FABRIC INTERFACE

Each host is connected to the fabric via an HFI (Fig. 5.1). HFIs bridge between the semantics of the host processor and the semantics of the fabric. The HFI minimally consists of the logic necessary to implement the physical and link layers of the fabric architecture, such that a node can attach to a fabric and send and receive packets to other servers or devices. An HFI may include specialized logic for executing or accelerating upper layer protocols. An HFI must also support whatever logic is necessary to respond to messages from network management components.

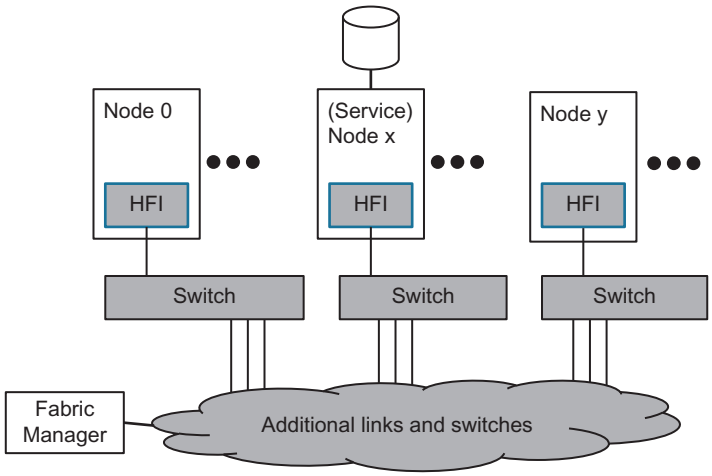


FIG. 5.1

Intel Omni-Path Architecture elements.

INTEL OPA SWITCHES

Intel OPA switches (Fig. 5.1) are OSI Layer 2 (link layer) devices, and act as packet forwarding mechanisms within a single Intel OPA fabric. Intel OPA switches are responsible for implementing QoS features, such as adaptive routing and load balancing. Switches also implement the Intel OPA congestion management functions. Switches are centrally provisioned and managed by the fabric manager (FM) software, and each switch includes a management agent (MA) to handle management transactions. Central provisioning means that switch configurations are programmed by the FM, including managing the forwarding tables to implement specific fabric topologies, configuring the QoS parameters, and providing alternate routes for adaptive routing. As such, all OPA switches must include MAs to communicate with the Intel OPA FM.

INTEL OPA MANAGEMENT

The Intel OPA fabric (Fig. 5.1) is centrally managed, and supports redundant FMs that provision and manage every device (i.e., HFIs, switches) in the fabric through MAs associated with those devices. The primary FM is an Intel OPA fabric software component selected during the fabric initialization process. The primary FM is responsible for: (1) discovering the fabric's topology; (2) setting up Fabric Identifiers and other necessary values needed for operating the fabric; (3) creating and populating the Switch-forwarding tables; (4) maintaining the Fabric Management Database; and (5) monitoring fabric utilization, performance, and error rates.

The fabric is managed by sending management packets over the fabric. These packets are sent *in-band* (i.e., over the same wires as regular network packets) using dedicated buffers on a specific VL (VL15). This specific VL for management may be configured to operate with or without flow control. Without flow control, management packets will be dropped if queue resources are not available at a port. End-to-end reliability protocols are used to detect dropped packets.

PERFORMANCE AND SCALABILITY

Large system cluster sizes continue growing because application performance demands are outpacing even Moore's law. See the *For More Information* section for information on the top 500 supercomputers. As a result, this increasing demand for performance has resulted in the transition from large Symmetric MultiProcessing (SMP) and vector-processing machines popular in the early 1990s into today's tightly connected microprocessor-based clusters interconnected via high-speed fabrics. Cluster sizes are continuing to grow significantly each year.

Given this trend, one of the objectives of Intel OPA fabric product family is to continually improve application performance and scalability. There are three key attributes (i.e., extreme message rates, low latency, and addressing) that drive these improvements which we will explain in more detail now.

EXTREME MESSAGE RATES

As applications are redesigned for increasing numbers of nodes and processing cores in a cluster, algorithm decomposition often results in more frequent communication and smaller message sizes. This trend requires fabrics and APIs that optimize for message rate.

To meet this need, Intel OPA is architected toward small message efficiency and performance. Intel OPA first-generation fabric achieves MPI application message rates up to 160 million packets per second. In addition, Intel OPA switches are capable of 100 Gbits/s (with 64 byte packets) full wire speed yielding packet rates up to 195 million packets per second per switch port.

While two-sided communication via the MPI API (see Chapter 15) is very popular for HPC applications today, another set of APIs and programming models (often referred to as one-sided or PGAS) are also emerging as well suited for certain types of graph analytic algorithms (see Chapter 16). PGAS APIs such as OpenSHMEM, Universal Parallel C (UPC), Coarray Fortran, and MPI-3 permit applications to efficiently access memory on remote nodes and often perform best with very high message rates that support smaller size data accesses.

Scalable message rate and latency is essential to support ever growing cluster sizes.

LOW LATENCY

Latency is very often as important as message rate. It is often difficult to implement applications to accurately prefetch data before it is needed, and in many cases, the data is not yet ready to fetch or send until it is immediately needed. This drives increasing importance for low data request latency.

While latency is easily measured at small scales, such as two-node tests, what really matters for applications is latency behaviors at scale as cluster size and application node count grow. To this end, Intel OPA fabric has been designed with scalable communication algorithms which ensure that the latency at scale is very similar to the observed latency for two nodes communication.

Having scalable latency is more than just building a fast wire speed. All contributions to latency must be considered. For interconnects, this includes wire speed, error detection and correction mechanisms, HFI hardware latency, HFI software latency, and processor cache behaviors in HFI software and applications.

Intel OPA fabric's first generation achieves latency of 100–110 ns in switches with full link layer resiliency enabled. Link layer resiliency is handled locally, per link, using an efficient link replay mechanism which adds no latency for successful packets and only adds a local link round trip (on the order of 100 ns) to recover the portions of a packet with detected errors.

ADDRESSING

The Intel OPA Link Layer (Layer 2) is designed for large-scale systems. OSI Layer 2 fabric packets enable 24-bit fabric addresses, as well as optimized formats for smaller systems. The OFA APIs generally hide these details from the application. Applications which are well behaved and make use of standard OFA APIs for address resolution and connection establishment will continue to function at scale on Intel OPA fabrics. It is important for programmers to avoid bypassing these standard mechanisms, for instance by exchanging LIDs directly. Using such mechanisms will not successfully scale and may not work with Intel OPA fabric extended addressing features in larger fabrics.

It is important to not bypass these standard addressing mechanisms. Doing so will degrade scaling and may not work in larger fabrics.

MULTICAST

Some applications make use of multicast, most notably IP applications doing Address Resolution Protocol (ARP), DHCP, or broadcast. Intel OPA fabric fully supports the OFA APIs for multicast, and Intel OPA switches implement full multicast packet handling using multicast spanning tree algorithms provided by the Intel OPA-centralized FM.

For scalable applications, using multicast mechanisms is generally discouraged. Multicast can have some negative side effects impacting the performance and scalability of the application:

- Multicast group join and leave operations require FM interactions and switch multicast routing table changes. On a large fabric, making frequent multicast membership changes can be inefficient and may outweigh multicast benefits. As such, multicast tends to work best when multicast groups are created and changed infrequently, such as creating IP multicast groups only on server boot and shutdown.
- Multicast traffic may go to uninterested nodes. Generic multicast groups, such as the IP broadcast groups, will include all nodes in the fabric. As such, it is likely that many nodes will not be interested in the message being sent. However, these nodes must interrupt their processor to handle the message. At small scale such interruptions are tolerable. However at large scale, such interruptions can become frequent and may inject processor and OS jitter effects delaying overall application progress.

TRANSPORT LAYER APIs

Intel OPA fabric provides three main transport layer APIs: OFA OFI, PSM, and Open Fabrics verbs. Above these transport layer APIs, numerous additional middleware and higher level APIs have been developed, such as MPI, OpenSHMEM, and sockets. The transport layer APIs are often used directly by I/O applications desiring maximum scalability and/or performance, such as parallel filesystems (e.g., Lustre, NFS).

OFA OPEN FABRIC INTERFACE

The OFI provides a general purpose software framework that is capable of handling a variety of fabric hardware and provides a standardized set of communication operations to higher code layers. The framework provides a library called libfabric that user-level applications may use.

The communication operations of libfabric are layered on top of the APIs mentioned later in this section. The message queue and tagged message queue operations are layered on PSM matched queue (MQ), and collectives can be synthesized in terms of PSM MQ operations. Put, get, atomics, and other operations designed to support PGAS or MPI3 RMA operations are layered on top of PSM active message (AM) implementation. Verbs semantics are layered on top of the standard Verbs access libraries. See the *For More Information* section for resources on this API.

PERFORMANCE-SCALED MESSAGING

PSM is a user-level library that provides a reliable fabric transport API for the Intel OPA HFI. The PSM API provides an MQ semantic that is a building block for MPI tag matching send and receive calls. The PSM API has been extended for the Intel

OPA architecture to provide 96 bits of tag matching supporting up to 32-bit user tags, up to 32-bit source rank information, and up to 32-bit communicator contexts. This allows significant scaling beyond the 64-bit tag matching provided by prior PSM implementations. The message-passing primitives provided by PSM are point-to-point. The Intel OPA PSM implementation is designed to scale to millions of MPI ranks.

Collective MPI operations can be synthesized from the point-to-point send and receive primitives using optimized algorithms that can be selected by parameters such as message size, collective communicator size, and topology. Additionally, PSM provides an active message (AM) API that can be used to implement arbitrary communication protocols using the AMs paradigm. This is used to implement a PGAS programming model using the OpenSHMEM API running over a GasNET conduit. See the *For More Information* section for resources on GASNet.

In the PSM library, short message send and receive is implemented using direct access to the send and receive hardware giving low latency and high message rate. For receive, PSM actively polls for arriving packets to eliminate host interrupt overheads. A registration caching scheme is used to reduce receive side overhead for typical MPI applications that reuse destination memory buffers often.

For each connection between a PSM sender and a PSM receiver, state is required to hold identification, status, sequencing, and control information. This is termed connection state or flow state. The Intel OPA HFI architecture holds no connection state in the HFI. This means that there are no hardware limits for capacity, caching, or scaling as the cluster size increases. Instead, all such connection state is held in the host benefiting from the host memory system capacity, cache capacity, and available bandwidth.

OPEN FABRICS VERBS AND COMPATIBILITY

The Intel OPA HFI supports a fully compliant OFA Verbs implementation. This includes support for reliable connected, unreliable datagram, and unreliable connected queue pairs. Shared receive queues are also supported. The standard user-level and kernel-level Verbs library interfaces are provided. All standard Verbs connection management protocols are supported as well as the 2 and 4 KB packet maximum transfer unit (MTU) sizes supported in InfiniBand implementations, Intel OPA fabric introduces an 8 KB MTU usable by a Verbs implementation to reduce the required packet rate for large messages. In addition, Verbs mechanisms are used for fabric management via the OFA user management datagram (umad) interface, with Intel OPA introducing a 2 KB management packet MTU.

The Intel OPA fabric implementation is partitioned between hardware capability in the HFI and host software. On the receive side, the incoming Verbs protocol packets are spread across receive contexts and processor cores using lower order bits from the queue pair number value and a mapping table. Interrupt coalescing moderates the host interrupt rate without overly delaying incoming latency sensitive packets. These features allow the Verbs performance to scale as more processor core resources are assigned to running the Verbs protocol code.

QUALITY OF SERVICE

Within Intel OPA fabrics, QoS features provide a number of capabilities, among them are job separation/resource allocation; service separation/resource allocation; application traffic separation within a given job; protocol (i.e., request/response) deadlock avoidance; fabric deadlock avoidance; traffic prioritization and bandwidth allocation; and latency jitter optimization by allowing traffic preemption.

SERVICE LEVELS

Within OFA APIs, the Service Level (SL) is the application identifier of a QoS domain. On the wire, a Service Channel (SC) is used as a per packet QoS identifier. Across each link, VLs are used to manage the hardware resources available. The FM handles the association of SLs to SCs and VLs.

TRAFFIC FLOW OPTIMIZATION AND PACKET INTERLEAVING

Traffic flow optimization allows Intel OPA fabric to support both high- and low-priority traffic on the same link while not sacrificing latency of the high-priority traffic. This feature permits storage and bulk I/O to be optimized using large packet sizes for maximum efficiency while latency sensitive compute traffic runs on the same links at higher priority.

Intel OPA fabric accomplishes this by permitting different packets on different VLs to be interleaved when they are sent across the link, allowing both higher link utilization, and lower latency for high-priority packets. A packet using a high-priority VL arriving at the link egress point can preempt and suspend an in-progress packet minimizing the latency of the high-priority packet. Once the high-priority packet is transmitted the suspended packet resumes. A low-priority packet can be preempted multiple times at an individual link egress point. The link egress point monitors the time a low-priority packet is preempted, and completes the packet's transmission if an FM-configured limit is exceeded.

CREDIT-BASED FLOW CONTROL

Intel OPA fabric uses credit-based link flow control. For a fabric port to send a packet, it must have sufficient flow control credits at the receiving port. Credit-based flow control means that data transfers on links are rigidly managed; there are no unauthorized data transfers, and it also means that Intel OPA fabric is so-called "lossless." In this case, lossless means simply that during normal operations packets are not dropped due to congestion.

Applications generally need not be aware of flow control, however scalable applications should be designed to be congestion aware and should avoid intentionally creating congestion trees for bulk data. In general, if an application wants to send large amounts of data to a shared server, it is better to send a small request

and let the server pull the data from the client when it is ready. An approach such as this is used in many scalable filesystem solutions. Under the covers, MPI and PSM use this mechanism for performance and scalability, but applications coded directly to OFI or verbs should also consider this approach.

If coding directly to OFI or Verbs, scalable applications can benefit from being congestion aware by using the same mechanisms MPI, PSM, and scalable filesystems use under the covers.

SECURITY

As cluster sizes grow, the need to share resources among multiple users, departments, and even multiple companies likewise grows. This need brings with it the implicit requirement of security. Security can be a matter of traditional separation and protection from malicious users, but it also can be a matter of protection from mistakes by well-meaning users.

PARTITION-BASED SECURITY

In Intel OPA fabric, Partitions are an isolation mechanism that operates at the Link Layer, with every Fabric packet being associated with a single partition. A partition provides isolation to the group of endpoints that are partition members for different types of traffic (i.e., all Transport Layers); however, this does not prevent a Transport Layer from providing finer grained security mechanisms. A partition in an Intel OPA fabric contains a group of Intel OPA endpoints. Communication is allowed within the partition, and is prevented with endpoints that are not in the partition. This allows partitions to be used to provide isolation between applications running on a fabric or between users on a fabric.

Individual endpoints may be identified as a full or limited member of a given partition. Full members are permitted to communicate with any partition member, but limited members are only permitted to communicate with full members. This mechanism permits the fabric to have shared services, such as management or a common global filesystem, while preserving isolation between nonservice partitions. Such services often require all end points to be members of the partition for that shared service. By making the providers of the shared service full members and the clients limited members, clients may access the service while still preventing clients from communicating directly to each other.

Each endpoint is a member of at least the management partition, and may be a member of multiple partitions. A typical endpoint will be a member of at least one application partition. When a Host node has multiple endpoints, each endpoint may have membership in different partitions. There are no architectural restrictions regarding which endpoints may be members of which partitions. For example, partitions may overlap and partitions need not be related to the physical topology of the fabric.

Partition security is enforced in the edge port of the switch connected to the HFI. The security mechanisms ensure that a source injects packets only into partitions of

which it is a member, and the packets are delivered only to endpoints in the same partition.

Partitions are created and managed by the Fabric Manager (FM). It initializes the registers and tables used to enforce partition security, and modifies them as partitions are redefined. Software running on the Host Nodes does not control the partition security registers and tables.

Within OFA APIs, the P_Key is the identifier for a security domain. There can be up to 32,767 unique partitions in a fabric. However, a given host can be a member of a modest number of P_Keys, in Intel OPA first-generation products, there is a limit of 32 P_Keys per switch port.

Fig. 5.2 shows a sample fabric with two application partitions (A and B). In this example, Nodes which are only in Partition A, such as Node 0, cannot communicate with Nodes in Partition B, such as Node y.

MANAGEMENT SECURITY

Intel OPA fabric has been designed with fabric management security as a first-order consideration. As such, Intel OPA fabrics define partition `0x7fff` as the management partition. In a typical deployment, all fabric management nodes will be full members of this partition and all remaining nodes will be limited members. The security for this partition applies to all management protocols between the fabric manager and MAs as well as name services and multicast membership queries between the clients and the fabric manager.

Intel OPA fabric by default has this partition secured. As such, applications should not hardcode the `0x7fff/0xffff` P_Key for use in communications with

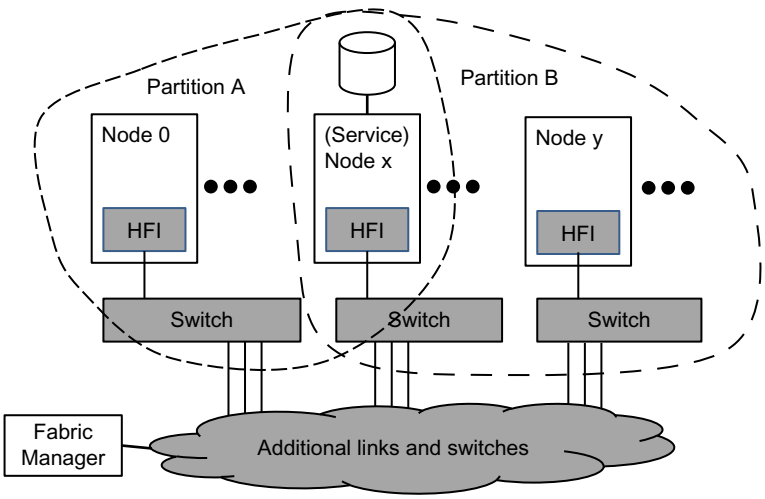


FIG. 5.2

Intel Omni-Path partitioning.

nonfabric management nodes. Instead, the appropriate `PathRecord` or multicast member record queries should be used to obtain the appropriate P_Key from the fabric manager.

Applications should not assume they can access management information from switches nor other servers. An application may access its own local management information (if nonroot umad access is enabled by the sysadmin). In general, use of such management information is discouraged.

Use `PathRecord` or multicast member record queries to obtain the appropriate P_Key from the fabric manager. Applications should not hardcode the `0x7fff/0xffff` P_Key for use in communications with nonfabric management nodes.

VIRTUAL FABRICS

Virtual Fabrics (vFabrics) provides a unifying concept by which Intel OPA Security and QoS features are configured and managed. It brings to the Intel OPA fabric many of the capabilities of Ethernet virtual LANs (VLANs). Using vFabrics, the administrator may slice up the physical fabric into many overlapping vFabrics. The administrator's selections determine how the configuration of the switches, HFIs, and links in the fabric is performed.

The goal of vFabrics is to permit multiple applications to be run on the same fabric at the same time with limited interference. The administrator can control the degree of isolation. As in the Ethernet VLANs, a given node may be in one or more vFabrics. vFabrics may have overlapping or completely independent membership. When IPoFabric (IPoIB) is in a fabric, typically each vFabric using IPoFabric represents a unique IP subnet, in the same way a unique subnet is assigned to different VLANs.

Each vFabric can be assigned QoS and security policies to control how common resources in the fabric are shared among vFabrics.

Additionally, as in the Fibre Channel Zoning, vFabrics can be used in an Intel OPA Fabric with shared I/O and storage. vFabrics help control and reduce the number of I/O devices visible to each host and prevent hosts from communicating with each other using storage vFabrics. This can further secure the fabric and in some cases, may make Plug and Play host software and storage-management tools easier to use.

Some typical usage models for vFabrics include the following:

- Separating a cluster into multiple vFabrics so independent applications run with minimal or no effect on each other.
- Separating classes of traffic. For example, putting a storage controller in one vFabric and a network controller in another enabling all networking to be secure and predictable.

---

vFabrics provide a unifying concept by which Security and QoS features are configured and managed.

A vFabric consists of a group of applications that run on a group of devices. For each vFabric the operational parameters of the vFabric can be selected.

As shown in Fig. 5.3, a set of applications, device groups, and finally vFabrics are defined. Each application is defined as a set of ServiceIDs and/or Multicast Group IDs (MGIDs). These are defined in terms of pattern match rules (or explicit lists), so that the wide range of 64 and 128 bit inputs can be easily specified. Each DeviceGroup consists of a list of device ports. Those device ports may be explicitly listed or matched via Node Description pattern matching. Finally, a VFabric is defined as a list of applications, a list of device groups, with a set of QoS and security policies, along with the P_Key and SL identifiers for the VirtualFabric.

Within the OFA APIs, packets for a given VFabric are identified by a P_Key (indicating security) and an SL (indicating QoS). During fabric initialization, the FM will have configured the devices to be aware of these two identifiers and assign the appropriate security (e.g., limited/full/none membership) and QoS (e.g., MTU, bandwidth, priority, preemption rank, static rate, flow control, and HoqLife) policies

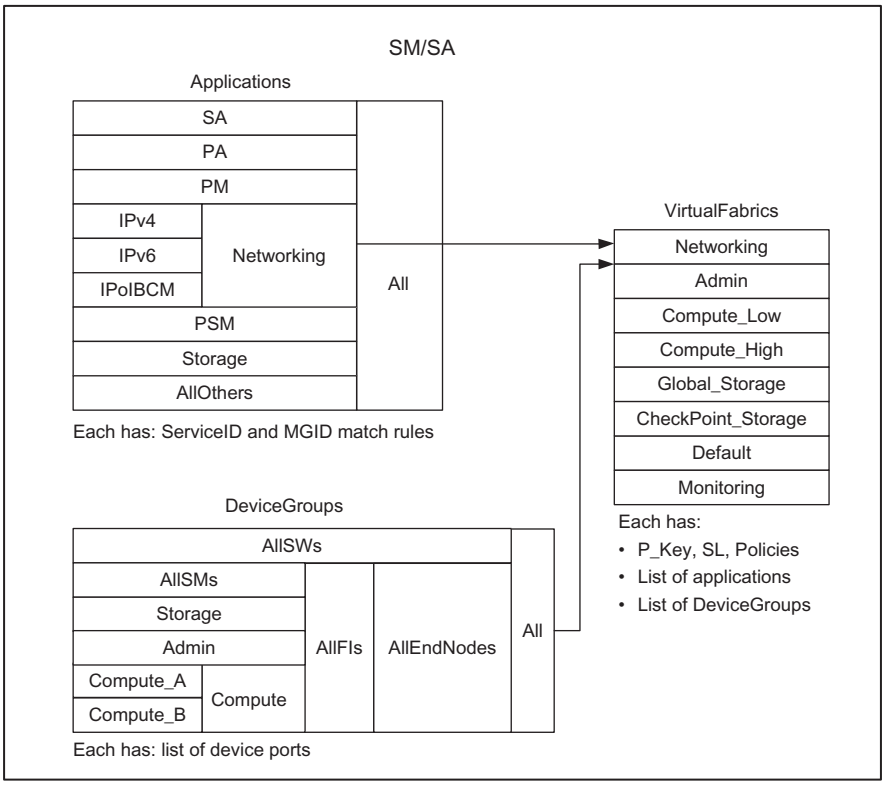


FIG. 5.3

Virtual Fabrics definition within FM.

to these identifiers at each port in the fabric. The settings for these policies may also influence other parameters for the port such as how buffers, VLs, and other resources are allocated to each VFabric.

The vFabrics mechanisms are closely tied to address resolution and multicast membership in the OFA APIs and applications.

When an application makes an address resolution or multicast membership query to the FM, the FM checks the application identifier (ServiceId or MGID) against the FM's list of applications to identify one or more potential matches. It also checks the source and destination nodes against the FM's list of DeviceGroups to identify one or more potential matching device groups. The FM then looks for vFabrics which include both a matching application and a matching device group and which also match any supplied SL, P_Key, and MTU. The one or more resulting vFabrics are used to compose address resolution responses which include the P_Key, SL, and other communications parameters (MTU, StaticRate, PktLifeTime) which the application should use to communicate through the fabric to the identified destinations.

Fig. 5.4 shows an example of this mechanism. In this example, a ServiceID (address resolution) or an MGID (multicast join/create) is supplied which matches the IPv4 Application. The IPv4 Application is part of the Networking and All applications, so they also are matched. Also a source and a destination are supplied explicitly or implicitly and a DeviceGroup is selected which matches both. The Compute_A group includes both the source and destination. The Compute_A group is also included in the Compute, AllFIs, AllEndNodes, and All groups, so they are also matched.

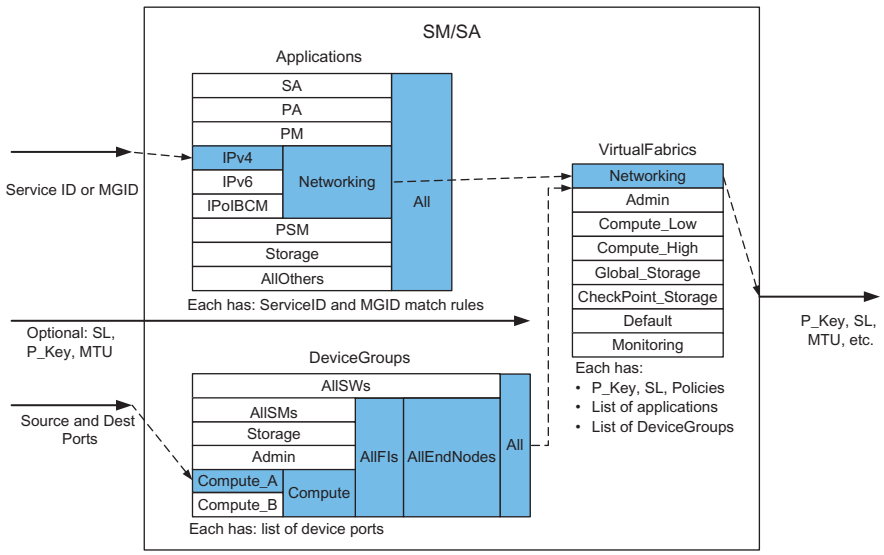


FIG. 5.4

Example of Virtual Fabrics resolution during address or multicast resolution by SM/SA.

Depending on the definition of Applications and DeviceGroups, it is possible for a given input to match two or more disjoint entries.

Given the list of matched Applications and matched DeviceGroups, the SM/SA searches the vFabrics list for an entry which includes both a matched Application and a matched DeviceGroup. In the example, the Networking VirtualFabric entry includes the Networking Application and the All DeviceGroup. If supplied, additional input parameters (SL, P_Key, MTU) will also be compared to the VirtualFabric to confirm a match. Finally, the response will include the P_Key, SL, MTU and other relevant polices from the matched VirtualFabric.

Some of the concepts of vFabrics are best explained via some simple sample configurations.

Fig. 5.5 shows the default FM configuration of vFabrics. In Fig. 5.5, a series of devices are shown, compute nodes A, compute nodes B, storage, switch Port 0s, FMs and other Admin nodes (such as job schedulers). In a given cluster there may be many of each type of device. Horizontally, two vFabrics are shown, Default VF and Admin VF. For each of these vFabrics the P_Key and SL are indicated. Parameters such as the P_Key and SL can be explicitly specified or dynamically assigned by the FM. Lastly on the right are shown the Applications which are associated with each vFabric.

In this example, the Default virtual fabric consists of all devices and "All Other" applications. This vFabric will be used by the majority of applications and is assigned a P_Key `0x0001` and SL 0. This example also shows the Admin virtual fabric. This fabric is used for secure fabric management of the fabric (SA, PA, PM, and implicitly the SM). As such it includes all the FM nodes and all the switch port 0's (which may also manage devices inside their given switch chassis). All other nodes are limited members. This vFabric is assigned P_Key `0x7fff` and SL 0. In this configuration, all traffic shares a single SL, so there is no fabric QoS. The Admin traffic is secured so that SA, PA, PM, and SM traffic is only permitted between the FM (and switch port 0's) and other devices. However other devices, such as Compute A devices, cannot attempt to manage other devices in the fabric.

Fig. 5.6 builds on Fig. 5.5 by providing an additional vFabric named Compute which includes all devices and the Compute applications. This vFabric uses

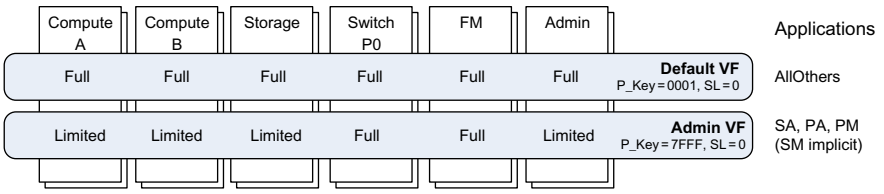


FIG. 5.5

Default Virtual Fabrics configuration.

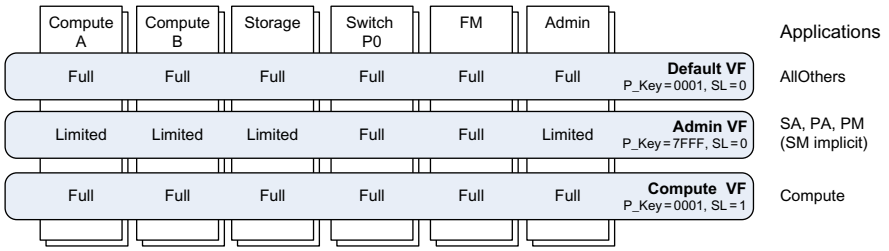


FIG. 5.6

Simple QoS Virtual Fabrics configuration.

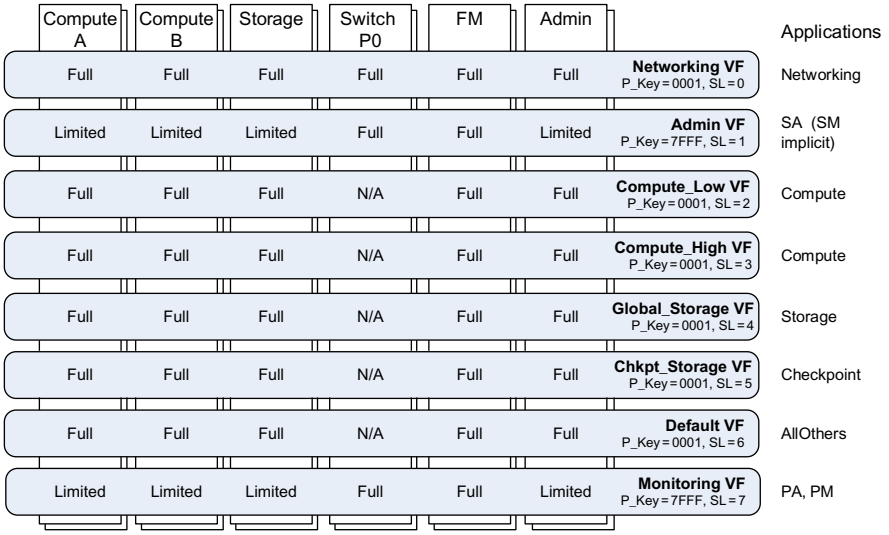


FIG. 5.7

Advanced QoS Virtual Fabrics configuration.

P_Key `0x0001` and SL 1. In this configuration, no new security is enabled, however by assigning Compute to a unique SL (and configuring the Compute virtual fabric's QoS policies such as bandwidth and preemption), the Compute fabric's traffic can be separated from other fabric traffic by using a unique SL and hence unique SC's and VL's throughout the fabric. This example also points out the flexibility and cross matching. The Default and Compute virtual fabrics share a common P_Key and hence have the same security rules. However, the Default and Admin virtual fabrics share a common SL and hence have the same QoS rules.

Fig. 5.7 builds further on Fig. 5.6. In this figure, eight unique vFabrics are defined and each is assigned a unique SL. This configuration has limited security as only management traffic is secured on P_Key `0x7fff`, while all other traffic is

unsecured and uses P_Key `0x0001`. However by assigning each vFabric a unique SL, they each have unique QoS characteristics and the performance impacts between vFabrics will be controlled by the QoS policies configured for each vFabric. In this configuration, due to separate vFabrics, the fabric performance statistics can also be monitored per vFabric. This occurs in the FM's Performance Manager and is based on analysis of the per VL statistics on each port. This example also shows that, if desired, the management traffic can also be split out; the Admin virtual fabric is used for SA traffic while the Monitoring virtual fabric is used for PM and PA traffic. This can permit the PM traffic to proceed at regular intervals with less impact from bursty name resolution traffic which may be occurring to the SA.

Fig. 5.8 is a security-focused example. In this example, each vFabric is assigned a unique P_Key. The Compute A virtual fabric is limited to the Compute A devices. Similarly the Compute B virtual fabric is limited to the Compute B devices. This means a Compute application can only be run within the Compute A or the Compute B devices, but may not use devices from both groups. Both Networking and Compute applications are permitted within those two vFabrics, so Compute A can be its own separate IP subnet from Compute B. This style of configuration may be useful when multiple departments or tenants of the cluster each need separate subsets of the cluster to run jobs and the sysadmin wants to secure the traffic so applications cannot mistakenly communicate with each other. This example also has an Admin virtual fabric, which is similar to Fig. 5.5 and secures the fabric management traffic from the other traffic. Finally there is a Services virtual fabric which includes the storage and admin nodes as full members with Storage and AllOthers applications. This permits the storage and admin nodes to communicate with any node in the fabric, however it prevents the nodes in Compute A from attempting to communicate with Compute B via this P_Key.

The concepts presented in these examples can easily be combined to create more advanced configurations with a mixture and security and QoS polices as required by the given fabric and its operations.

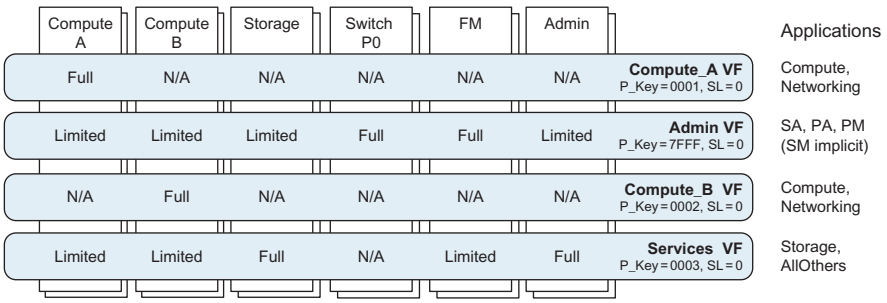


FIG. 5.8

Simple security Virtual Fabric configuration.

The Fabric Manager also has some capabilities to dynamically adjust vFabrics by making changes to the configuration in a live fabric. This can be used to facilitate occasional changes in the needs, such as adding or removing a department or tenant on the fabric. See the *For More Information* section for resources with more details on the OPA FM and vFabrics.

UNICAST ADDRESS RESOLUTION

vFabrics is designed to integrate automatically within the OFA stack. Applications which take advantage of standard connection establishment and address resolution (aka `PathRecord` resolution) mechanisms such as RDMA CM, IB CM, and ibacm will automatically make the necessary SA queries so that the fabric manager can provide `PathRecord`s with the appropriate settings for SL, Partition Key (P_Key), MTU and other parameters used for fabric communications.

TYPICAL FLOW FOR WELL BEHAVED APPLICATIONS

The standard flow for `PathRecord`s and vFabric address resolution is as shown in Fig. 5.9.

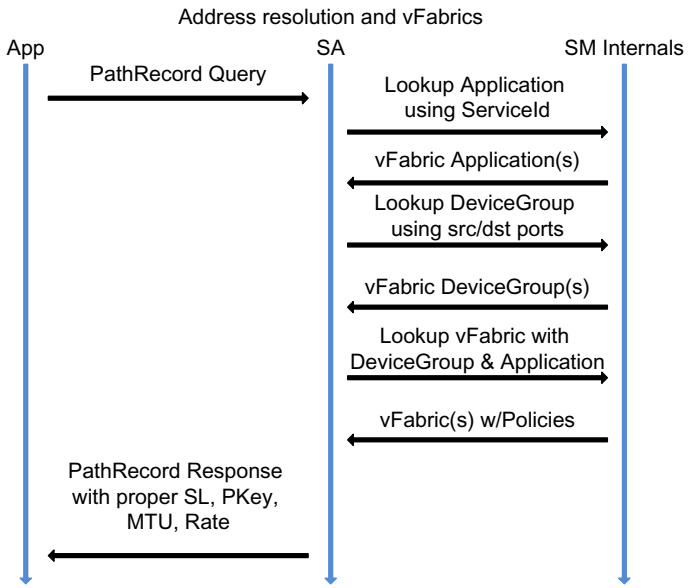


FIG. 5.9

Address resolution flow within SM/SA.

As show in the flow diagram in Fig. 5.9, the SA makes use of the Service Identifier (Service Id) supplied in the `PathRecord` query to identify the application making the request. The source and destination nodes specified in the query will imply the possible set of DeviceGroups which are relevant, and then the SA will find the virtual fabric(s) which represent the intersection of the given application with the possible device groups. To ensure this process works smoothly and correctly, it's important that the FM configuration of vFabrics specifies the appropriate set of Service Ids for the application. Some applications may provide direct configuration of the Service Id as a 64 bit number. This is especially true of those using the IB CM or ibacm directly. Applications using the RDMA CM will specify a protocol and port number and these will be used to compose a 64 bit service ID.

RDMA Service IDs take the form `0x0000000001`*`NNPPPP`* where *N* is the Internet Assigned Numbers Authority (IANA) protocol number and *P* is the port number the application needs to bind on, both in hexadecimal. For example, Lustre is known to use the TCP protocol (IANA `0x06`) on port 987 (`0x03db`) so its Service ID is expected to be `0x00000000010603db`.

OUT OF BAND MECHANISMS

In some cases, the application may use non-standard, out of band or ad-hoc mechanisms to establish connections. In which case, the key connection parameters of P_Key, SL, and MTU will need to be specified a priori. This approach is shown in Fig. 5.10.

Intel OPA fabric software provides assorted tools which may be used to identify the P_Key, SL, and MTU associated with a given vFabric. In some cases, those tools may be used to automate the discovery of the SL, P_Key, and MTU and then provide them directly to the application, hence reducing the risk of human mistakes and permitting future runs of the application to correctly obtain any changes to the vFabrics configuration.

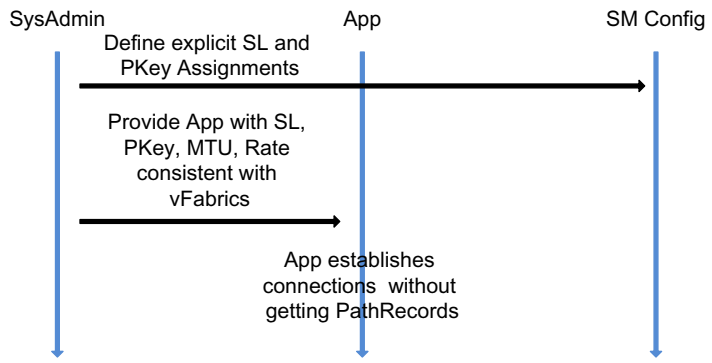


FIG. 5.10

Out of band mechanisms for QoS and security.

MULTICAST ADDRESS RESOLUTION

Applications which take advantage of standard multicast mechanisms (aka Multicast Member Records) either directly, via IPoFabric (aka IPoIB) or via ibacm, will automatically receive the proper SL, P_Key, and MTU as part of the multicast group parameters.

TYPICAL FLOW FOR WELL-BEHAVED APPLICATIONS

The standard flow for Multicast Member Records when creating a new multicast group is as shown in Fig. 5.11.

As shown in the flow diagram in Fig. 5.11, the SA makes use of the MGID supplied in the Multicast Member Record to identify the application making the request. The source nodes specified in the query will imply the possible set of DeviceGroups which are relevant, and then the SA will find the virtual fabric(s) which represent the intersection of the given application with the possible device groups. To ensure this process works smoothly and correctly, it's important that the FM configuration of vFabrics specifies the appropriate set of MGIDs for the application. Some applications may provide direct configuration of the MGID as a 128-bit number. This is especially true of those using the IB SA or ibacm directly. Applications using IPoFabric (aka IPoIB) will specify an IP multicast group and these will be used to compose a 128-bit MGID.

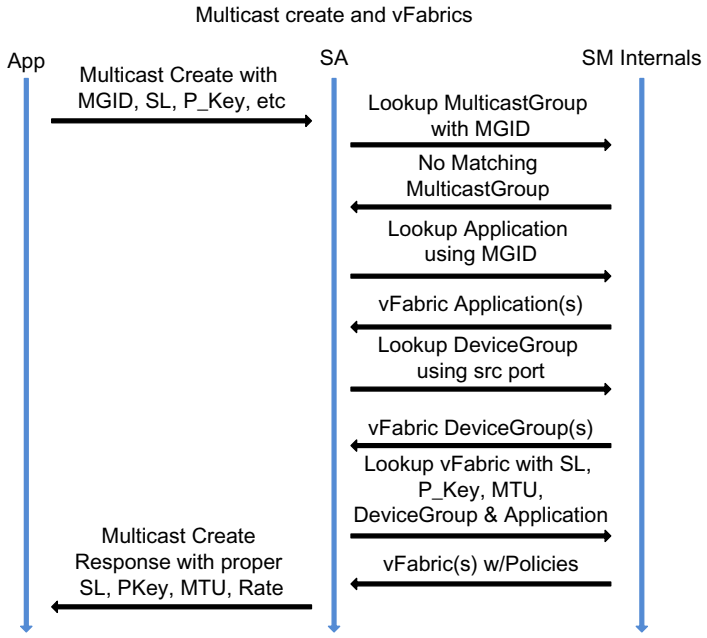


FIG. 5.11

Multicast create flow within SM/SA.

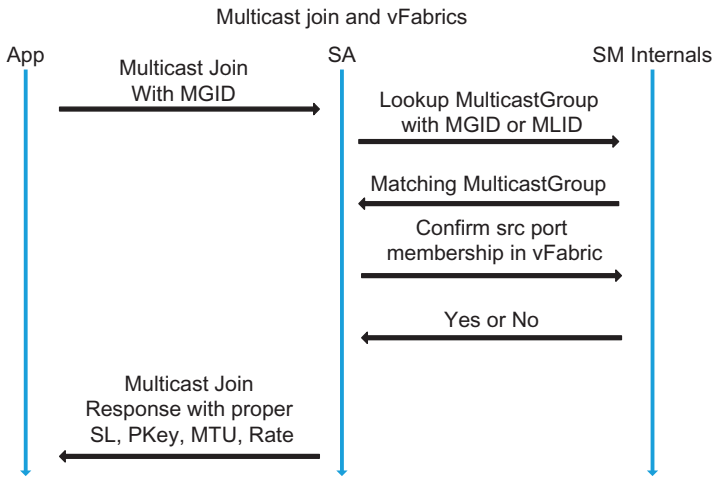


FIG. 5.12

Multicast join flow within SM/SA.

For IPv4, MGIDs are composed by IPoFabric as follows:

```
MGID=0xffFS401bPPPP0000:00000000GGGGGGGG
where F=flags, S=scope, P=PKey, and G=IP Multicast Group
```

For IPv6, MGIDs are composed by IPoFabric as follows:

```
MGID=0xffFS601bPPPPGGGG:GGGGGGGGGGGGGGGG
where F=flags, S=scope, P=PKey, and G=IP Multicast Group
```

When an application joins an existing multicast group via the Multicast Member Record mechanisms, the steps in Fig. 5.12 occur.

In this case the pre-existing multicast group will already have an assigned SL, P_Key, and MTU. The new join request will merely need to confirm the source nodes ability to communicate with that P_Key.

SUMMARY

The Intel Omni-Path Architecture introduces a multi-generation fabric architecture designed to meet the scalability needs of datacenters ranging from the high-end of HPC to the breadth of commercial datacenters. Link-level reliability features provide the reliability needed for systems at large scales. The QoS architecture is coupled with new packet preemption capabilities to enable both bandwidth fairness and low latency jitter for high-priority packets. In the first product generation, each link provides 100 Gb/s of bandwidth, and each HFI can achieve 160 million messages per second. Switch latency has been reduced to under 110 ns. These are substantial improvements relative to prior-generation products while preserving the existing

software ecosystem. In addition, a new community standard network API, i.e., OFA OFI (libfabric), is designed to match user-level semantic requirements to enable hardware innovation beneath the API.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this chapter.

- Further Omni-Path information, documentation and software can be found at www.intel.com/omnipath and http://lotsofcores.com/omnipath.
- Intel, "Intel True Scale Fabric Architecture: Enhanced HPC Architecture and Performance," Intel, Santa Clara, CA, USA, 2012.
- G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins and J. Reinhard, "Cray cascade: a scalable HPC system based on a Dragonfly network," in SC12: International Conference on High Performance Computing, Networking, Storage and Analysis, Los Alimitos, CA, USA, 2012.
- L. A. Barroso, J. Clidara and U. Hozle, The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition, California, USA: Morgan and Claypool Publishers, 2013.
- P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard and J. M. Squyres, "A Brief Introduction to the OpenFabrics Interfaces: A New Network API for Maximizing High Performance Application Efficiency," in IEEE Hot Interconnect 23, Santa Clara, CA, USA, 2015.
- ISO/IEC, 7498-1: Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model, Geneva, Switzerland: ISO, 1996.
- InfiniBand Trade Association and others, Infiniband Architecture Specification: Release 1.0, InfiniBand Trade Association, 2000.
- IEEE, IEEE Std 802.3bj(TM)-2012; Amendment 2: Physical Layer Specifications and Management Parameters for 100 Gb/s Operation Over Backplanes and Copper Cables, New York, NY, USA: IEEE Computer Society, 2014.
- W. J. Dally, "Virtual-channel Flow Control," Parallel and Distributes Systems, IEEE Transactions on, vol. 3, no. 2, pp. 194–205, 1992.
- M. Luo, K. Seager, K. Murthy, C. Archer, S. Sur and S. Hefty, "Early Evaluation of Scalable Fabric Interface for PGAS Programming Models," in PGAS 2014, 8th International Conference on Partitioned Global Address Space Programming Models, Eugene, OR, USA, 2014.
- www.top500.org — a web site tracking the current and historical top 500 performing HPC systems in the world. The Performance Development charts are especially interesting.
- http://gasnet.lbl.gov — a Web site describing GasNET and its use on assorted PGAS compilers and libraries including UPC and SHMEM.

CHAPTER

μarch optimization advice

6

This chapter offers advice on tuning specific to the Knights Landing design which we know as the microarchitecture, and we like to abbreviate as μarch but pronounce as micro-architecture (or micro-ark). In other words, we focus on tuning advice arising specifically from the Knights Landing μarch design when compared with the Knights Corner μarch (found in the first generation Intel Xeon Phi products) or the μarch of a recent Intel® Xeon® processor. While maximal performance tuning of any machine relies on detailed knowledge of the underlying μarch, code may remain portable and performance portable by parameterizing choices such as how many threads per core to use, which vector instruction set to ask for with compiler switches, and how to best utilize the available memory types. The optimal choices, for these settings, are generally rewarded with higher performance.

What is new with Knights Landing in this chapter?

This chapter is exclusively about Knights Landing, including advice on how to deal with differences from the prior Intel Xeon Phi product (Knights Corner).

This chapter offers a collection of microarchitectural details and discusses their implications on tuning. The section names have been chosen to reflect a very short version of the advice, and the sections themselves offer more detailed explanations. There are also a series of sections on differences between AVX-512 and IMCI (Intel® Initial Many-core Instructions—the 512-bit SIMD ISA on Knights Corner) specifically for those transitioning code.

If you prefer to not dive into such machine specific details, we still recommend flipping through the sections and reading the first paragraph of each. If you crave details, perhaps because you are a compiler writer or a very advanced application tuner, we trust you will find the details within this chapter to be very useful.

BEST PERFORMANCE FROM 1, 2, OR 4 THREADS PER CORE, RARELY 3

The choice of how many threads per core to utilize is highly application dependent, and it is good advice to parameterize this choice and run performance tests to choose the best number for an application on any given machine. One thread per core on the

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00006-5

Knights Corner was generally far from the peak performance for every application (the best was most often two or three per core); Knights Landing μarch, which may reach maximum performance with a single thread per core.

An individual thread has the highest performance when running as the only thread on a core. As thread count per core grows to two or four, most applications will have higher aggregate performance, but lower per thread performance. If an application can scale its performance to an arbitrary number of threads, four threads per core are likely to have the highest instruction throughput. Practical limitations on memory capacity or parallelism may limit the scaling effectiveness when increasing the threads used per core. However, memory *latency* sensitive applications usually benefit from using the maximum number of threads per core available, as it is more likely that one of the thread's data is available for processing at any given time.

One thread per core can be powerful enough for an application to get maximal performance. Running three threads per core may be the least likely to be the best choice, but it can be.

The Knights Landing μarch partitions "per core" resources in fourths when using three or four threads on a core. Because of this, a three thread configuration will have fewer total internal resources than any other configuration of threads per core. Placing three threads on a core is less likely to perform better than two or four threads per core, but we have seen examples of three being the optimal—obviously when the reduced resources on the core were not the key bottleneck.

Most highly tuned applications are likely to use one thread per core, with two thread per core being in a close second. Unlike Knights Corner, a single thread on Knights Landing does have the ability to use the full capabilities of the core and to saturate memory. There will be some applications (e.g., random access with memory latency sensitivity) which find four threads per core useful, so it is worth parameterizing this decision and doing speed comparisons.

If multiple threads per core are active, but only a subset are doing useful work (e.g., the other threads are in a barrier), performance will suffer. The threads doing nonuseful work will take up pipestages and resources from the threads that we want to make progress—limiting their potential performance. If threads are likely to be stalled for a long period, it might be better to suspend the stalled thread, and wake it up later. The overheads for an OS level thread switch are high, so this should be done carefully.

HYPERTHREADING: DO NOT TURN IT OFF

It is possible to turn off hyperthreading on Knights Landing via a BIOS boot time option. However, turning it off is unlikely to ever be useful to any application. Keep hyperthreading on, and if only one thread per core is used, it will give all the resources to that one thread.

MEMORY SUBSYSTEM

CACHES

Knights Landing has multiple levels of caches. Data and instruction sets that fit in caches are likely to perform better. Knights Landing has separate 32 KB caches for instructions and data. These are commonly referred to as the "level 1" (L1) caches. The two cores in a tile share a 1 MB cache that can hold a mix of instructions and data. This is commonly referred to as the "level 2" (L2) cache. When multiple tiles read the same cache line, each tile might have a copy of the cache line. If both cores in the same tile read a cache line, there will only be a single copy in the L2 cache of that tile. Cache lines found in the L1 cache can be accessed with high bandwidth and low latency. Cache lines found in the L2 cache have lower bandwidth and higher latency than the L1 cache, but higher bandwidth and lower latency than an access to memory (including MCDRAM).

If MCDRAM is configured as a cache, then it can hold data or instructions that are accessed by the cores in a single place. If multiple tiles request the same line, only one MCDRAM cache line will be used.

MCDRAM AND DDR

MCDRAM and DDR memory have different latency and throughput profiles. When using the various memory modes (cache/hybrid/flat) and deciding on where to allocate memory, this knowledge is important. In most memory configurations, the DDR capacity will be substantially larger than MCDRAM capacity. Likewise, MCDRAM capacity will be much larger than the combined L2 cache. Working sets that fit in MCDRAM capacity, but not in the L2 caches should be in MCDRAM. Large or rarely accessed structures should migrate to DDR. The processor will try to do this dynamically if MCDRAM is put in cache or hybrid mode. If memory is in "flat" mode, working sets are bound to one memory or the other at allocation time.

Fig. 6.1 conceptually depicts the relationship between memory bandwidth and memory latency for MCDRAM and DDR. In general, memory latency increases with

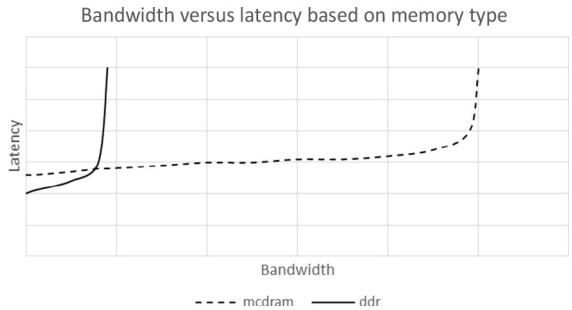


FIG. 6.1

Latency and bandwidth curves for MCDRAM and DDR.

memory bandwidth because as memory gets busier with increasing number of requests, each request waits longer in hardware queues for their turn to access memory. While this increase is gradual for most part, it becomes exponential as demand for memory bandwidth approaches memory's peak bandwidth. Although MCDRAM latency is higher than DDR when bandwidth demand is low, it does not rise as fast as DDR latency when bandwidth demand increases because MCDRAM sustains much higher bandwidth compared to DDR. The actual memory latency experienced by software will depend on the bandwidth demand it places on MCDRAM and DDR.

ADVICE: LARGE PAGES CAN BE GOOD (2M/1G)

Most programs default to using 4KB pages. There are many instances where using 2M or 1G pages can improve performance. For instance, when accessing a large data structure (64 MB) randomly, 2M pages will stay in the DTLB, while 4KB pages will constantly require the hardware to translate linear to physical addresses. For many applications with large datasets and nonsequential access patterns, larger pages can provide significant performance improvements.

μARCH NUANCES (TILE)

In this section, we review several suggestions on how to optimize the performance of the tile for the Knights Landing μarch.

INSTRUCTION CACHE, DECODE, AND BRANCH PREDICTORS

The front end unit (instruction cache, decoders, and branch prediction) handles most HPC code well. There are some restrictions that rarely impact performance of which optimizers and compiler writers should be aware. Code generation should avoid large instructions, or instructions with many prefixes. Branch targets should be within 4GB of the branch.

The instruction cache can fetch up to 16 bytes per cycle. Since the processor can process up to two instructions per cycle, this could limit performance when the average instruction length is greater than 8 bytes. The only common instructions that use more than 8 bytes are vector math instructions with a memory source that have a 4-byte displacement. If executing a long sequence of such operations, constraining the displacement size will provide an improvement in performance. Chapter 21 has an example of using this knowledge to increase FMA issue rate by encoding FMAs as 8-byte instructions, instead of longer instructions. Using multiple base or index registers is one option to reduce the size of the displacement.

When selecting instructions, try to avoid instructions with many prefixes. If an instruction has three or more prefixes, the hardware requires more time to determine instruction boundaries. The penalty will happen each time the instruction

is encountered. This is generally not an issue, but some instructions, like ADCX and ADOX, are affected. These instructions should not be used in performance critical code.

The indirect branch predictor assumes that the branch destination is in the same 4 GB chunk of the virtual address-space as the indirect branch instruction (note for these purposes a "ret" is an indirect branch instruction). Since indirect branches are used to enter dynamic shared libraries, which are normally scattered through the virtual address space, calls into DLLs incur two misprediction penalties (one on the call and one on the return). Similarly, system calls in Linux are made via a pseudodynamic library; therefore, they will also suffer from the misprediction. To mitigate this, try to limit the frequency of system calls and calls to dynamic libraries. Statically linking libraries (instead of using shared libraries) can help. This is especially important for low latency library calls that are accessed frequently—whether MPI, a math function, or a network driver. An updated version of the dynamic linker (ld.so) is shipped with some OS releases. If requested, it will try to place all shared libraries into the low 4 GB of the address space, to avoid the mispredictions on shared library calls. If this linker is installed, setting the environment variable LD_PREFER_MAP_32BIT_EXEC is likely to improve performance. Since it has no impact if the modified dynamic linker is not present, we set the environment variable on all of the machines where we run our program. Therefore, we like to put this in our ~/.bash_profile file:

```
export LD_PREFER_MAP_32BIT_EXEC=1
```

This works for 64-bit applications because branch prediction performance on Knights Landing cores can be negatively impacted when the target of a branch is more than 4 GB away from the branch. On Linux, adding the `LD_PREFER_MAP_32BIT_EXEC` bit will affect `mmap`, so it will map executable pages with `MAP_32BIT` first. `LD_PREFER_MAP_32BIT_EXEC` does reduce bits available for address space layout randomization and can only be enabled by creating the environment variable `LD_PREFER_MAP_32BIT_EXEC` (its value is unimportant, its existence creates the behavior change). Anecdotal feedback indicates this can often give a boost of several percent in performance. This is a feature of glibc 2.23 and later.

Having "`export LD_PREFER_MAP_32BIT_EXEC=1`" in your ~/.bash_profile file may be worth a few percent in performance.

INTEGER

Knights Landing integer instructions perform similar to the Intel® Atom™ processors. There is no optimization for instructions with partial integer writes, or partial flag writes. Instructions like PUSH, POP, and DIV should be avoided if simpler instructions can be used.

If a code sequence needs to PUSH or POP several values sequentially, it is better to issue multiple stores or loads (each using a unique RSP offset) and then update

RSP once. Each PUSH/POP instruction generates its own μop to update RSP. This puts more pressure on allocation and integer execution bandwidth. PUSH and POP instructions are commonly used as part of the calling convention, and eliminating redundant RSP updates can yield some benefit to code with many short functions.

The Knights Landing μarch does not optimize 8- and 16-bit integer partial registers (like AH, AL, and AX). Each reference to an integer register reads and writes the full 64-bit version. For (very old) code that was written when 32-bit CPUs were a new concept, this might be an issue, but is unlikely to be a problem for high performance applications. It is simplest, and not slower, to just refer to the 32- and 64-bit versions of the integer registers (like EAX and RAX).

The Knights Landing μarch does not optimize partial flag writes. Instructions that write EFLAGs status bits will write all the status bits. If a subset of the bits is unchanged by an instruction, then the EFLAGs writer cannot execute until the previous writer of EFLAGs has completed. This can cause unexpected performance drops relative to recent Core microprocessors. INC and DEC are the two most frequently encountered instructions that are affected by this. Unless your code is dependent on not writing all flags, it is usually better to replace DEC with "SUB 0x1," and INC with "ADD 0x1."

Integer division is a common mathematical operation, used whenever two integers are divided, or a modulus operation is specified. Unfortunately for those who care about performance, computing this operation is quite slow. If the integers are known to be relatively small (16 bits or less), there are fast software sequences to emulate the division (that the compiler might try to generate). If the divisor is known to be a power of 2, please use SHR (division) and/or AND (remainder) instead of DIV. If the divisor is a constant, multiplication by another constant will provide the correct answer. If the input values are highly constrained, a precomputed lookup table might provide better performance. Division instructions should be aggressively minimized by the compiler, either by using the techniques mentioned earlier or by hoisting redundant divisions out of inner loops.

VECTOR

Performance optimized code should not use x87 and SSE instructions. Instead, code should use the AVX (packed and scalar) equivalents (128-, 256-, or 512-bit width) as much as possible. SSE and x87 instructions are generally similar performance to AVX but are occasionally much slower. There are potential performance glass jaws when mixing SSE with 256- or 512-bit operations mentioned later on.

Do not use the COMIS* and UCOMIS* set of instructions. Replacing these instructions with the two instruction sequence of VCMPS* and KORTEST is almost always better for performance.

Some instructions, like VCOMPRESS*, are fast when writing a register, but much slower when storing to memory. Where possible, it is faster to do a VCOMPRESS to a register and then store it. However, potential setup (e.g., masking) may limit the benefit.

MASKMOVQ and similar instructions should be replaced by k-masked ZMM load/store instructions.

The Knights Landing μarch does not have the same restrictions on mixing SSE and AVX code that Core processors have. Knights Landing will suffer some performance loss if an AVX instruction that operates on 256- or 512-bits is allocated before all previous SSE instructions for that thread have retired. The performance loss is comparable to a mispredicted branch. When compiling for the Knights Landing μarch, it is best to not generate SSE instructions. If the binary has been freshly compiled, only legacy libraries should have SSE code in them. This limits any performance hiccups to a few instructions after returning from legacy library calls. Extremely bad code generation would alternate SSE instructions with wider AVX instructions. To achieve excellent performance using the Knights Landing μarch, avoid this.

The Knights Landing μarch does not optimize the performance of VZEROUPPER or VZEROALL. The VZEROALL instruction requires about 10 cycles of bandwidth in the allocation, execution, and retirement pipelines. The frontend in the Knights Landing μarch uses the microcode sequences (MSs) for VZEROUPPER and will need about 15 cycles. Core processors generally take a cycle of bandwidth at each point for VZEROUPPER. Code for Knights Landing should avoid both instructions as much as possible.

Experience has shown that intrinsic heavy code from IMCI (Knights Corner μarch) implementations will not necessarily generate optimal code for the Knights Landing μarch. IMCI had free simple permutes and a more limited Instruction Set Architecture (ISA) relative to AVX-512 in the Knights Landing μarch. An expert coder will want to scrub legacy 512-bit intrinsics tuned for IMCI to limit the number of MS flows generated, and to fix limitations imposed by the Knights Corner μarch (unaligned memory accesses). Large speedups are possible from even a quick scrub of the generated code.

Do not use the HADDP* instruction to do a horizontal addition with Knights Landing μarch. There are examples later in the document that list more efficient instruction sequences.

For vector code, it is generally best to keep consistent with the element types. For instance, if a vector compare on 32-bit integer elements writes a YMM register, performance is likely to be better if the consumer of the YMM register has a 32-bit integer data type (like PXOR).

There are a few instructions relevant to HPC that are long latency. The most commonly used are floating-point division, floating-point square root, and integer division. The programmer and/or compiler should carefully consider various tradeoffs for each of these instructions. The DIVPD, DIVPS, SQRTPD, and SQRTPS instructions are implemented as MSs that use Newton-Raphson to iterate to an IEEE correct answer. This gives better bandwidth and slightly worse latency than the same instructions on most recent Intel Xeon processor cores. The scalar versions of these instructions are shorter MSs, that use a hardware unit, with worse bandwidth and better latency than the packed versions of the instructions. The integer divide

performance is slow, similar to Intel Atom processor cores, and should be avoided in performance critical code.

Compiler options like "-no-prec-div" in the Intel compilers take advantage of loosening IEEE compatibility requirements and can generate faster instructions sequences with the AVX-512ER instructions (more on this later).

For floating point divisions and square roots, it is greatly beneficial to be able to vectorize. The vector versions are approximately as fast as the scalar versions, so able to do 8 or 16 operations simultaneously greatly improves performance.

Many complex math operations (with real and imaginary values) benefit from the VFMADDSUB* instructions. Please use them when appropriate.

MEMORY ACCESSES AND PREFETCH OPTIONS

The Knights Landing μarch performs well on accesses that split cache lines, but there is still a performance penalty relative to accesses that do not split cache lines. If an algorithm has an access pattern that streams through memory, it would be beneficial to align as many of the accesses as possible to a 64-byte boundary. Likewise, if we want a 32-byte value (YMM), do not access 64-byte (ZMM) in memory and then mask off the last 32-bytes. This creates cache line splits for no reason.

The Knights Landing μarch must take extra time when a load or store access crosses a 4KB boundary in order to test the permissions for each page involved. This happens regardless of the page size. This introduces a larger performance penalty for the access that crosses the boundary. If an algorithm is streaming through memory with unaligned 64-byte loads, every 64th load will cross a 4KB boundary. It is best to align the access pattern as much as possible. If it is not possible, consider inserting a few software prefetches to the L2 cache (PREFETCHT1 and similar instructions) several iterations ahead of the stream. This will start the page translation early and permit the L2 hardware prefetcher to start fetching the stream on the next page.

Some access patterns for gather and scatter will always have pairs of consecutive addresses. One common example are complex numbers, where the real and imaginary parts are laid out contiguously. It is also common when w, x, y, and z information is contiguous. If the values are 32-bit, it is faster to gather and scatter the elements as half as many 64-bit elements. If the numbers are 64-bit, then it is usually faster to load and insert 128-bit values instead of doing a gather.

There are multiple hardware prefetchers on each tile in the Knights Landing μarch. One analyzes all the accesses in the data cache and the instructions that generated them—the Instruction Pointer Prefetcher (IPP). This prefetcher attempts to insert hardware prefetches to the L1 if a strided access pattern is detected on a cacheable page. The IPP will not generate prefetches that cross a 4KB boundary.

The L2 hardware prefetcher tries to identify streaming access patterns (where consecutive cachelines are accessed) and can track up to 48 streams. The L2 hardware prefetcher only looks at the address of requests that make it to the L2 cache. It has no knowledge of the instruction, or any accesses that were satisfied inside the individual cores. If there are four threads enabled per core, this means that the L2

hardware prefetcher can track up to six streams per thread (two cores share an L2 cache). If a stream is detected, and there are few memory accesses pending, hardware prefetches for later elements of the stream will be send to the L2, and if they miss, those accesses go to memory. The hardware prefetcher will not stream across a 4KB address boundary.

The hardware prefetchers can be disabled within the BIOS menus at boot time or using BIOS tools, specific to your machine, similar to those used in Chapter 3 for memory and cluster modes.

Tuning tools can measure memory activity and can help understand the effectiveness of both hardware and software prefetches. The counters available to tuning tools, such as the Intel® VTune™ Amplifier, include only "demand" loads in the "miss" counters. This was not possible on Knights Corner. Effective prefetching should cause "miss" counts to be reduced. Performing comparisons by turning on or off different types of prefetching (software prefetching comes in different types and usages, there are two hardware prefetchers) and observing the "miss" rates can be very instructive. More information on the specific event counters are available in Chapter 14. We also have more pointers to additional prefetching advice on our website as mentioned in *For More Information* at the end of this chapter.

Experiments with prefetching will see "miss" rates go down when prefetching is effective because the "miss" counters include only "demand" loads, not prefetches. Chapter 14 discusses these counters.

AVX-512PF supports many flavors of software prefetch instructions. The Knights Landing μarch is more resilient to cache misses than Knights Corner, so programmers with experience from Knights Corner should not feel compelled to aggressively insert software prefetches. With the two hardware prefetchers described in previous paragraphs, most streaming and short strided access patterns should be detected by hardware prefetchers. If the access pattern is streaming, then a programmer will likely benefit from software prefetches beyond a 4KB boundary, since the hardware prefetchers will miss the first several accesses to a new 4KB region. If the access pattern is known, but nonstreaming, then software prefetches can be beneficial. This is especially true if the access pattern is a relatively large stride (>256 bytes), since the hardware prefetches will not fetch across a 4KB boundary. The software prefetch will do the PMH walk to fill the TLB and start the memory reference early. When accessing multidimensional arrays (like A[i][j][k]) sequentially in *i*, *j*, or *k*, remember that one access order has sequential addresses, and the others involve large strides in the address. The former is captured by hardware prefetchers, and the latter is generally best captured by software prefetches.

Generally, software prefetching into the L2 will show more benefit than L1 prefetches. L1 prefetches hold critical hardware resources (e.g., fill buffer) until the cache line fill completes. L2 prefetches do not hold those resources, so it is less likely that inserting an L2 software prefetch will negatively impact performance. When

using L1 software prefetches, it is strongly advised that the software prefetch hits in the L2 cache so that the length of time that the hardware resources are held is minimized. Common practice is for L1 software prefetches to be used well after an L2 prefetch has previously been executed for the same address.

Avoid inserting software prefetches too aggressively. It is always a good idea to try turning software prefetching on and off and compare results. Code tuned for Knights Corner generally needed prefetching more than Knights Landing. Chapter 26 describes an application which benefited strongly from software prefetching with Knights Corner, but runs best with no software prefetching on Knights Landing.

Additionally, if software prefetches reference addresses that are not valid, or are not mapped by the OS, an application may experience a significant slowdown from such software prefetches. The performance monitoring event NUKE.ALL provides an indication of when this might be affecting performance. Tools such as Intel® VTune™ Amplifier (see Chapter 14) can help.

The MEU is partially out of order. Memory accesses are dispatched from the scheduler in-order but can complete in any order. This impacts performance negatively when the second oldest memory μop is ready to dispatch (address is good), but the oldest memory μop has not yet dispatched (memory address is not good). This can be easily observed in algorithms that do pointer chasing. If code loads the pointer from memory (cycle 0), and then dereferences it in the next μop, the earliest point that μop can dispatch is in cycle 4. This means that no other MEC μops from that thread can dispatch in the second slot of cycle 0, or in any slot of cycles 1, 2, or 3. A more common example of pointer chasing relevant to supercomputing is when the base address of many arrays (&a[0]) is kept on the stack. The compiler and/or ninja programmer should try to maximize the number of memory operations between the load of the base address, and the instruction that dereferences it. A simple option is to load base pointers as consecutive memory references and then dereference them in pairs. For instance, if we want to access a[i] and b[i], we can write the code shown in Fig. 6.2. The naïve sequence, shown in Fig. 6.3, interchanges the second and third

```
movq    r15, [rsp+0x40]     ;;; cycle N (load &a[0])
movq    r14, [rsp+0x48]     ;;; cycle N+1 (load &b[0])
vmovups zmm1, [r15+rax*8]   ;;; executes in cycle N+4
vmovups zmm2, [r14+rax*8]   ;;; cycle N+5
```

FIG. 6.2

Interleaved pointer chasing.

```
movq    r15, [rsp+0x40]     ;;; cycle N (load &a[0])
vmovups zmm1, [r15+rax*8]   ;;; executes in cycle N+4
movq    r14, [rsp+0x48]     ;;; cycle N+4 (load &b[0])
vmovups zmm2, [r14+rax*8]   ;;; cycle N+8
```

FIG. 6.3

Naïve pointer chasing.

instructions and is three cycles slower. The advantage of the naïve sequence is that it only requires a single integer register to hold the base address. The faster sequence requires two integer registers for the base addresses, which can put more pressure on the register allocator. The benefit of the first sequence increases if the first pointer load misses the data cache.

If there are many loads in the machine, it might be possible to hoist up the pointer loads so that there are several memory references between the pointer load and dereference, without requiring more integer registers to be reserved.

Store to load forwarding is simplistic in the Knights Landing μarch. Integer loads and stores (RBX, EAX) can forward if the store and load have the same beginning memory address and the load is not larger than the store. Vector, x87, and MMX loads and stores can forward (ZMM0, YMM1, XMM2, MM3, and ST4) with the same conditions. Vector to Integer and vice-versa do not forward between themselves, and the load must wait until the store is postretirement. Vector stores that use a mask (a k-register other than k0) cannot be immediately forwarded from. If an algorithm requires such behavior, it may benefit from merging the value in a register and then storing to memory using k0. Later loads can then forward from the merged value.

The memory hierarchy that caches lines and determines forwarding uses the address of the access. The L1 data cache uses bits 11:6 to identify which cache set to use. Forwarding logic uses bits 11:0 and the size of the access to identify potential forwarding or conflicts between loads and stores. If there are many conflicts, performance could be degraded. Unfortunately, many dynamic memory allocation routines (dependent on OS and compiler) will start large memory regions with the same bottom 12 bits. If a program accesses many arrays with identical shapes (element size and dimensions) and similar indices, performance could be significantly degraded. It is beneficial for bits [11..6] of memory accesses to be different, as illustrated in Fig. 6.4.

There are multiple ways to offset dynamic arrays. If `free()` is not relevant, we can just allocate a few hundred bytes and manually offset the arrays: `b=(double*)((char*)b+192)`. We can also code our own versions of memory allocation routines to achieve this. Alternatively, we can use `memalign()` with different alignment

```
a = malloc(sizeof(double) * 10000);
b = malloc(sizeof(double) * 10000);
// very likely in most OSes that (a & 0xfff) == (b & 0xfff)
for (i=0; i < 10000; i++) {
      // a[i] and b[i] of iteration N collide
      // a[i] of iteration N-1 and
      //     b[i-1] of iteration N collide
      a[i] = b[i] + 0.5 * b[i-1]);
}
```

FIG. 6.4

Array access pattern that collides in the same L1 set.

directives for each dynamic allocation to induce the OS to provide different memory alignments.

Many scientific applications that use large arrays are vulnerable to this. The SPEC06 application leslie3d can be affected by this quite easily.

When using VGATHER and VSCATTER, we often need to set a mask to all ones. An efficient instruction to do this is KXNOR of a mask register with itself. Since VSCATTER and VGATHER clear their mask as the last thing they do, a loop carried dependence from the VGATHER to KXNOR can be generated. Because of this, it is wise to avoid using the same mask for source and destination in KXNOR. Since it is rare for the k0 mask to be used as a destination, it is likely that "`KXNORW k1, k0, k0`" will be faster than "`KXNOR k1, k1, k1`."

CODE EXAMPLES

In this section, we look at some common kernels and access patterns and how to best code for the Knights Landing µarch.

Fig. 6.5 shows macrocode for a horizontal reduction of 32-bit elements. This is for single-precision floating point but can be easily altered for a 32-bit integer reduction. Fig. 6.6 shows code for a 64-bit floating-point horizontal reduction. Fig. 6.7 has a simplified code fragment for the inner loop of DGEMM, trying to compute

```
;; vector to reduce is in zmm6
vextractf64X4     zmm1, zmm6, 0x1
vaddps       ymm1, ymm6, ymm1
vpermpd      ymm4, ymm1, 0xFF
vpermpd      ymm5, ymm1, 0xAA
vpermpd      ymm3, ymm1, 0x44
vaddps       xmm1, xmm1, xmm4
vaddps       xmm3, xmm5, xmm3
vaddps       xmm3, xmm1, xmm3
vpsrlq       xmm1, xmm3, 32
vaddss       xmm3, xmm3, xmm1
```

FIG. 6.5

Single-precision horizontal reduction from ZMM vector to scalar.

```
;; vector to reduce is in zmm6
vextractf64X4 zmm1, zmm6, 0x01
vaddpd        ymm1, ymm6, ymm1
valignq       ymm4, ymm1, 0x3
valignq       ymm5, ymm1, 0x2
valignq       ymm3, ymm1, 0x1
vaddsd        ymm1, ymm1, ymm4
vaddsd        ymm3, ymm5, ymm3
vaddsd        ymm3, ymm1, ymm3
```

FIG. 6.6

Double-precision horizontal reduction from ZMM vector to scalar.

```
;; matrix - matrix dense multiplication
prefetcht0  [rdi+0x400]   ;; get A matrix element into L1$
vmovapd     zmm30, [%rdi]
prefetcht0  [rsi+0x400]   ;; get B matrix element into L1$
vfmadd231pd zmm1, [rsi+r12]{b}, zmm30 ;; b-cast B element
vfmadd231pd zmm2, [rsi+r12+0x08]{b}, zmm30
vfmadd231pd zmm3, [rsi+r12+0x10]{b}, zmm30
vfmadd231pd zmm4, [rsi+r12+0x18]{b}, zmm30
vfmadd231pd zmm5, [rsi+r12+0x20]{b}, zmm30
vfmadd231pd zmm6, [rsi+r12+0x28]{b}, zmm30
vfmadd231pd zmm7, [rsi+r12+0x30]{b}, zmm30
vfmadd231pd zmm8, [rsi+r12+0x38]{b}, zmm30
prefetcht0  [rsi+0x440]         ;; pull line into the L1 $
vfmadd231pd zmm9, [rsi+r12+0x40]{b}, zmm30
vfmadd231pd zmm10, [rsi+r12+0x48]{b}, zmm30
vfmadd231pd zmm11, [rsi+r12+0x50]{b}, zmm30
vfmadd231pd zmm12, [rsi+r12+0x58]{b}, zmm30
vfmadd231pd zmm13, [rsi+r12+0x60]{b}, zmm30
vfmadd231pd zmm14, [rsi+r12+0x68]{b}, zmm30
vfmadd231pd zmm15, [rsi+r12+0x70]{b}, zmm30
vfmadd231pd zmm16, [rsi+r12+0x78]{b}, zmm30
```

FIG. 6.7

Fragment of DGEMM inner loop.

C = A * B. The code in Fig. 6.7 has 16 partial sums. There should always be FMA instructions ready to execute in the VFU (6 cycles per FMA, up to 2 FMAs per cycle). It is important to keep the average instruction length at 8 bytes or less, to enable maximum throughput. This is why the index register in these examples `(r12)` is used, so the displacement used in the FMAs can be kept small. At the end of the inner loop, the partial sums will need to be added to produce a single value to be stored out (to the C matrix).

DIRECT MAPPED MCDRAM CACHE

The MCDRAM cache is a convenient way to increase memory bandwidth. As a memory side cache, it can automatically cache recently used data and provide much higher bandwidth than what DDR memory can achieve. When MCDRAM is placed in cache mode, it is a direct mapped cache. This means that multiple memory locations map to a single place in the cache. Because of this, a simple first optimization for a program is to turn on the MCDRAM cache. Some applications that heavily utilize a few GB of memory could see performance improvements of up to 4×. Because of the simplicity of this—no source code changes, and the large possible performance benefits, moving from DDR only to MCDRAM cache mode should be one of the first performance optimizations to try.

There are a few scenarios where enabling the cache could reduce performance. One case is when the MCDRAM cache is not able to hold the accessed working set. If an application streams through 64 GB of memory without reuse, then checking the MCDRAM cache (and missing) will only increase latency.

Another thing to note is that the direct mapped cache uses the physical address, not the linear address. Even if an address is contiguous in the linear/virtual address, the physical addresses that the OS gives to the application memory are not required to be. This can cause cache contention when using a significant portion of the MCDRAM cache. This is likely to reduce the peak memory bandwidth achievable. This can vary from run to run, as how the OS allocates pages can change from run to run. Monitoring the cache hit rate events from `perfmon` can be instructive in diagnosing this.

If MCDRAM cache is enabled, every modified line in the tile caches (L1 or L2 cache) must have an entry in the MCDRAM cache. If it is not in the MCDRAM cache, then the line will be downgraded to "shared" from "modified" in the tile cache. There is a very small probability that a pair of lines that are frequently read and written will map to the same MCDRAM set. This could cause a pair of reads that would normally hit in the L1 caches to become reads that need to go to DDR. This would cause a pair of threads to become substantially slower than the other threads in the chip. Due to linear to physical mapping variation, this behavior could vary from run to run, making it difficult to diagnose.

This case is very hard to create, but the way we employed to forcibly create this case was with two threads read and write their local stacks. Conceptually, any data location that is commonly read and written would work, but register spills to the stack are the most frequent case. If the stacks are offset by a multiple of 16 GB in physical memory, they would collide into the same MCDRAM cache set. A run-time that forced all thread stacks to allocate into a contiguous hardware memory region would avoid this from occurring. There is hardware to reduce the frequency of set conflicts from occurring. The probability of hitting this scenario on a given node is extremely small. The best clue to detecting this is that a pair of threads on the same chip are significantly slower than all other threads during a program phase—the threads that collide should vary from run to run, happen rarely, and only when MCDRAM cache mode is enabled.

ADVICE: USE AVX-512

ADVICE: UPGRADE TO AVX-512 FROM AVX/AVX2 AND IMCI

The Knights Landing microprocessor fully supports the full ISA from Intel processors, including the recent Intel Xeon processor (codenamed Haswell and Broadwell) μarchs, with the exception of Transactional Synchronization Extensions. Knights Landing supports the new 512-bit Advanced Vector Extensions (AVX-512). Because of this, vector instruction support exists for MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AVX, and AVX2. For floating point instructions, SSE* operates on 128-bits at a time, AVX* on 128- or 256-bits and AVX-512 on 512-bits at a time. Roughly speaking, AVX-512 offers 4× performance over SSE* and 2× over AVX* for similar mathematical operations, purely because of the number

of mathematical operations done in parallel. Using the instructions with the widest support will offer the highest performance on any processor, and that holds true for Knights Landing as well.

Specifically, the additional AVX-512 support includes AVX-512F (foundation), AVX-512CD (conflict detection), AVX-512PF (prefetching), and AVX-512ER (exponential and reciprocal) ISA extensions. With these new instructions, AVX-512 introduces the ZMM register set (32 512b vector registers—ZMM0-31) and mask registers (eight 64-bit mask registers—k0–7). All devices supporting AVX-512 must support the foundational instructions (AVX-512F), while the AVX-512 extensions are dependent on the μarch. All Intel Xeon Phi processors from Knights Landing forward will support at least the AVX-512CD, AVX-512PF, and AVX-512ER extensions. All Intel Xeon processors that include AVX-512F will also include at least the AVX-512CD extension. In general, this level of detail is left to be dealt with by a compiler or library. It is a detail with which programmers using intrinsics will also need to deal with (see Chapter 12).

The Knights Corner coprocessor did not support the AVX-512 ISA but did support a similar 512-bit vector ISA referred to as IMCI. Many of the instructions in IMCI and AVX-512 are similar, with different bit encodings. However, some differences exist. AVX-512 offers superior scatter and gather support and is significantly different than IMCI. In addition, IMCI supported "free" swizzles inside 128-bit lanes, and several up-conversions and down-conversions to and from memory, which are not supported by AVX-512 on Knights Landing. There are no other announced devices from Intel that support the IMCI ISA extensions. IMCI will likely live up to its name as being only "initial" instruction set to support 512-bit vectors. Intel has announced future support for AVX-512 in its Intel Xeon processor line as well.

The bottom-line is clear—use the AVX-512 ISA to achieve the best performance.

SCALAR VERSUS VECTOR CODE

The compiler or programmer can determine that a loop can be vectorized, but there is a threshold of loop count below which vectorization may not be better than scalar code. A very simple guideline is that loops that iterate more than 16 times will be faster with vector code, and scalar code is faster for loops that iterate less than four times. In between those two values, vector and scalar code has about the same performance.

For the rest of this section, a slightly more sophisticated algorithm to choose between scalar and vector code is discussed. It assumes that the operations in the loop and the number of iterations the loop will execute are known.

A horizontal reduction operation transforms a vector into a scalar value. These generally occur at the end of a loop when a scalar value must be produced from the vector registers. The most common type of horizontal reduction sums the vector elements. The constants used in the sample cost model are listed in Fig. 6.8.

Operation	Cost	Example
Simple math	1	A*B+C
Load (with cache-line split)	1 (2)	A[i]
Store (with cache-line split)	1 (2)	A[i] = 2;
Gather (scatter) 8 element	15 (20)	A[column[i]]
Gather (scatter) 16 element	20 (25)	A[column[i]]
Horizontal reduction	30	sum += A[i]
Division or square root	15	A/B

FIG. 6.8

Vectorization cost model.

```
for (i=0; i<N; i++) { sum += a[i]*K + b[i]; }
```

FIG. 6.9

Horizontal reduction.

It is easiest to explain the algorithm via examples. Due to differences in various compilers' internal formats and programmers' knowledge, the algorithm has to be a bit flexible. Some of the corner cases are glossed over.

Fig. 6.9 shows a simple horizontal reduction loop.

There are two loads (a[i] & b[i]), an FMA, and a horizontal reduction in the vector version (to get sum). If compiled to scalar code, the horizontal reduction becomes a scalar add per loop iteration.

The heuristic costs are 4*N for scalar code, and 3*ceiling(N/8) +30 for double precision AVX-512 vector code. The vector version of the code assumes that the main loop and remainder loop are vectorized. The break-even point between scalar and vector code would be around $N=9$. If N is smaller, then scalar is better. If N is larger, then vectorized code will be faster.

The code in Fig. 6.10 utilizes gathers. There are two loads (indir[i] & b[i]), an FMA, a store (c[i] =), and a gather/load, depending on whether vector or scalar code is produced. The costs are 5*N for the scalar code and 19*ceiling(N/8) for the vector code. Scalar is better if $N<4$.

The code in Fig. 6.11 uses even more gathers. There is one load (ind[i]), an FMA, a store (c[i] =) and 2 gathers/loads. The scalar cost is 5*N, and the vector cost is 33*ceiling(N/8). Scalar is better for small values of N. For very large values of N, vector code is probably better, but the margin is likely small.

```
for (i=0; i<N; i++) {c[i] = a[indir[i]]*K + b[i]; }
```

FIG. 6.10

Gather computation.

```
for (i=0; i<N; i++) {c[i] = a[ind[i]]*K + b[ind[i]]; }
```

FIG. 6.11

Double gather.

The code in Fig. 6.12 has a gather and a horizontal reduction. There are two loads (ind[i] & b[i]), an FMA, a gather/load, and a horizontal reduction/sum. The scalar cost is 5*N, and the vector cost is 19*ceiling(N/8)+30. Scalar code is better for $N \leq 13$, and vector code is probably better for larger trip counts.

The code in Fig. 6.13 uses a scatter and a division. There are three loads (a[i], b[i], & ind[i]), a scatter/store, and a division (a[]/b[]). The scalar cost is 19*N, and the vector cost is 38*ceiling(N/8). Scalar code is better if $N \leq 1$. In other words, vectors are best if you have more than one iteration to do!

The code in Fig. 6.14 uses a gather and scatter operation. There is one load (ind[i]), a scatter/store (b[ind[i]]), and a gather/load (a[ind[i]]). The scalar cost is 3*N, and the vector cost is 36*ceiling(N/8). According to the heuristic, scalar code is always better. With the in-order MEC, it is likely that the scalar version of the loop would benefit from being unrolled a bit.

A code fragment from miniMD is shown in Fig. 6.15. Outside the IF clause, there is a load, three gathers, and six math operations. Inside the IF clause, there is a

```
for (i=0; i<N; i++) {sum += a[ind[i]]*K + b[i]; }
```

FIG. 6.12

Gather/horizontal reduction.

```
for (i=0; i<N; i++) {c[ind[i]] = a[i] / b[i]; }
```

FIG. 6.13

Division/scatter loop.

```
for (i=0; i<N; i++) {b[ind[i]] = a[ind[i]]; }
```

FIG. 6.14

Gather/scatter copy loop.

```
for (int k = 0; k < numneigh; k++) {
  int j = neighs[k];
  double rsq = (xtmp - x[3*j])^2 +
               (ytmp - x[3*j+1])^2 +
               (ztmp - x[3*j+2])^2;
  if (rsq < cutforcesq) {
     double sr2 = 1.0/rsq;
     double sr6 = sr2*sr2*sr2;
     double force = sr6*(sr6-0.5)*sr2;
     res1 += delx*force;
     res2 += dely*force;
     res3 += delz*force;
  }
}
```

FIG. 6.15

Code fragment from miniMD.

division, eight math operations, and three horizontal reductions. The scalar cost is 10*numneigh+23 * numneigh * percent_rsq_less_than_cutforcesq. The vector cost is (52+23) * ceiling(numneigh/8)+3 * 30. Scalar code makes sense if (numneigh < 6) or if the compiler is highly confident that the if clause is almost never taken.

For many compilers, a vectorized loop is generated, and a remainder loop is used to take care of the rest of the operations. In other words, the vectorized loop is executed floor(*N*/8) times, and the remainder loop is executed (*N* mod 8) times. In that case, modify the equations to use floor instead of ceiling to determine whether the primary loop should be vectorized. For the remainder loop, the maximum value of the loop trip count is known: one less than the vector width. If *N* is unknown, it is simplest to set *N* to half the vector width (four for a ZMM vector of doubles).

More sophisticated analysis can be done in this area. This is a starting point for optimizers.

INSTRUCTION LATENCY TABLES

Knowledge of the latency and bandwidths of instructions is important to achieve good performance. Care should be taken to select instructions that can be executed in the least amount of time. Vector math instructions are generally 2 or 6 cycle latency and have their information listed in Fig. 6.16. Scalar integer instructions are mostly 1 cycle latency and have their information listed in Fig. 6.17.

Some of the instructions have throughput lower than 1. They might be bottlenecked by the number of operations being done (gather/scatter and division) or by the time it takes to generate the μops in the frontend. The converts that are listed

Vector Instructions	Latency	Instructions/cycle
Simple integer	2	2
Most vector math (FMA)	6	2
Mask operations	2	2
X87 / MMX math	6	1
EMU (AVX-512ER)	7	0.5
Shuffle / permutes (1 src)	2	1
Shuffle / permutes (2 src)	3	0.5
Convert – same width	2	1
Convert – different width	6	0.2
Vector Loads	5	2
Store to load forwarding	2	2
Gather (8 elements)	15	0.2
Gather (16 elements)	19	0.1
Float to integer move	2	1
Integer to float move	4	1
DIVSS or SQRTSS	25	0.05
DIVSD or SQRTSD	40	0.03
Packed DIV or SQRT	38	0.1

FIG. 6.16

Vector latency and bandwidth.

Scalar Integer instructions	Latency	Instructions/cycle
Most math	1	2
Integer multiply	3 or 5	1
Store to load forwarding	2	1
Integer Loads	4	1
Integer division	Variable	0.05

FIG. 6.17

Scalar integer latency and bandwidth.

as "different width" are conversions where the vector width of the sources and destination are different.

ADVICE: USE AVX-512 EXTENSIONS FOR KNIGHTS LANDING

These relatively small sets of instructions (AVX-512ER, AVX-512PF, and AVX-512CD) are important for efficient vectorization of HPC programs.

ADVICE: USE AVX-512ER

The AVX-512ER instructions provide high precision approximations of exponential, reciprocal, and reciprocal square root functions. Approximations from earlier ISA extensions, like rcp11ps, are far less accurate. An accurate approximation can reduce execution time for iterative methods like Newton-Raphson. Fig. 6.18 shows code using the Newton-Raphson method to compute a single 32-bit float division with vrcp28ss. Both values are read off the stack. Note the use of rounding mode overrides on some of the math operations.

IMCI TO AVX-512: RECIPROCAL AND EXPONENTIALS

The 23-bit single-precision reciprocals and exponentials found in Knights Corner are replaced by AVX-512ER in Knights Landing as listed in Fig. 6.19. AVX-512ER brings Knights Landing a high accuracy implementation of base 2 exponential,

```
vgetmantss    xmm18, xmm18, [rsp+0x10], 0
vgetmantss    xmm20, xmm20, [rsp+0x8], 0
vrcp28ss      xmm19, xmm18, xmm18
vgetexpss     xmm16, xmm16, [rsp+0x8]
vgetexpss     xmm17, xmm17, [rsp+0x10]
vsubss        xmm22, xmm16, xmm17
vmulss        xmm21{rne-sae}, xmm19, xmm20
vfnmadd231ss  xmm20{rne-sae}, xmm21, xmm18
vfmadd231ss   xmm21, xmm19, xmm20
vscalefss     xmm0, xmm21, xmm22
```

FIG. 6.18

Division via Newton-Raphson algorithm using vrcp28ss.

Instruction	Description
VEXP2PD, VEXP2PS	Compute approximate exponential 2^x with a relative error of at most 2^{-23}.
VRCP28PD, VRCP28PS, VRCP28SD, VRCP28SS	Compute approximate reciprocals with a relative error is at most $2^{-28.}$
VRSQRT28PD, VRSQRT28PS, VRSQRT28SD, VRSQRT28SS	Compute approximate reciprocals of square roots with a relative error is at most $2^{-28.}$

FIG. 6.19

AVX-512ER instructions.

reciprocal, and reciprocal of square root with 28-bit mantissa bits. The two reciprocal instructions are much more accurate than the 14-bit mantissa version in AVX-512F. The single-precision version is also more accurate than Knights Corner's 23-bit version.

The reason for increased accuracy and double-precision support is to avoid expensive divide instruction and runtime calls to exponential function when the application accuracy requirement is relaxed.

ADVICE: USE AVX-512CD

The AVX-512CD instructions (see Fig. 6.20) allow for efficient vectorization of several access patterns. A common scheme can be characterized as the "histogram update." This is when a memory location is read, operated on, and then stored to. A sample piece of C code with this access pattern is shown in Fig. 6.21. This code can be incorrectly vectorized with a gather, vpadd, and a scatter. To get the correct answer with vectorization, we must worry about cases where key[n] and key[m] have the same value, and *n* and *m* are in the same vector chunk. The AVX-512CD

Instruction	Description
VPCONFLICTD, VPCONFLICTQ	Detect duplicate values within a vector and create conflict-free subsets
VPLZCNTD, VPLZCNTQ	Count the number of leading zero bits in each element
VPBROADCASTMB2Q, VPBROADCASTMW2D	Broadcast vector mask into vector elements

FIG. 6.20

AVX-512CD instructions.

```
for (i=0; i < 512; i++)
     histo[key[i]] += 1;
```

FIG. 6.21

Histogram update in C.

instructions detect these cases and permit correct vectorization of the loop. Disassembly from a compiler that vectorizes the histogram update code is shown in Fig. 6.20.

Another example of using AVX-512CD is found in Chapter 20 to create a function called `conflict_safe_accumulate` using a sequence of intrinsics for a conflict safe gather-modify-scatter force update. The use of AVX-512CD allows accumulation of forces from multiple lanes with the same neighbor index into a single-data lane so that the last value written to memory will contain the correct result.

ADVICE: USE AVX-512PF

The AVX-512PF instructions (see Fig. 6.22) are used to aid memory performance when the programmer or compiler knows with a high degree of certainty the set of cache lines that will be accessed in the near future. Their relationship to gather and scatter is similar to the relationship of prefetch(w)* to loads and stores. They should be used in equivalent scenarios, where the compiler or software writer is comfortable in providing hints to hardware on which lines should be in cache. Prefetch instructions do not impact architectural state.

Instruction	Description
VGATHERPF0DPS, VGATHERPF0QPS, VGATHERPF0DPD, VGATHERPF0QPD	Using signed dword/qword indices, prefetch sparse byte memory locations containing single/double floating-point data using opmask k1 and T0 hint.
VGATHERPF1DPS, VGATHERPF1QPS, VGATHERPF1DPD, VGATHERPF1QPD	Using signed dword/qword indices, prefetch sparse byte memory locations containing single/double floating-point data using opmask k1 and T1 hint.
VSCATTERPF0DPS, VSCATTERPF0QPS, VSCATTERPF0DPD, VSCATTERPF0QPD	Using signed dword/qword indices, prefetch sparse byte memory locations containing single/double floating-point data using writemask k1 and T0 hint with intent to write.
VSCATTERPF1DPS, VSCATTERPF1QPS, VSCATTERPF1DPD, VSCATTERPF1QPD	Using signed dword/qword indices, prefetch sparse byte memory locations containing single/double floating-point data using writemask k1 and T1 hint with intent to write.

FIG. 6.22

AVX-512PF instructions.

IMCI TO AVX-512: SOFTWARE PREFETCHING

Software Prefetching is mainly used to hide the memory latency for an application. For Knights Corner, software prefetching is essential. In addition to latency sensitive applications, software prefetching proved useful even on many streaming and/or memory bandwidth bound workloads. This was largely because of the hardware prefetcher Knights Corner was limited to only a streaming L2 Hardware prefetcher per core.

Knights Landing has multiple hardware prefetchers per core which greatly diminishes the need for aggressive software prefetching for many applications. We recommend reading about a methodology for tuning prefetching in Chapter 21, *Prefetch Tuning Optimizations*, of the Pearls Volume Two book. See reference in *For More Information* at the end of this chapter.

Since it may still be worthwhile to experiment with software prefetching in applications, we explain the different types of prefetch hints available (with some "intrinsics" example) and the differences that exist between Knights Corner and Knights Landing.

The table in Fig. 6.23 shows the different prefetch instructions available on Knights Corner with details on the level of the cache the instruction brings the cache line into and also the state of the cache line (which is controlled using "prefetch hints").

The table in Fig. 6.24 shows the different prefetch instructions available on Knights Landing with details on the level of the cache the instruction brings the cache

Instruction	Cache Level	Non-Temporal	Bring as exclusive	Prefetch Hint (for intrinsics)
vprefetch0	L1	No	No	_MM_HINT_T0
vprefetchnta	L1	Yes	No	_MM_HINT_NTA
vprefetch1	L2	No	No	_MM_HINT_T1
vprefetch2	L2	Yes	No	_MM_HINT_T2
vprefetche0	L1	No	Yes	_MM_HINT_ET0
vprefetchenta	L1	Yes	Yes	_MM_HINT_ENTA
vprefetche1	L2	No	Yes	_MM_HINT_ET1
vprefetche2	L2	Yes	Yes	_MM_HINT_ET2

FIG. 6.23

Knights Corner (IMCI) prefetch instructions.

Instruction	Cache Level	Non-Temporal	Bring as exclusive	Prefetch Hint (for intrinsics)
prefetcht0/	L1	No	No	_MM_HINT_T0/
prefetchnta	L2	No	No	_MM_HINT_NTA
prefetcht1/				_MM_HINT_T1/
prefetcht2				_MM_HINT_T2
prefetchw	L1	No	Yes	_MM_HINT_ET0
prefetchwt1	L2	No	Yes	_MM_HINT_ET1

FIG. 6.24

Knights Landing prefetch instructions.

```
int L2dist=64*8;
int L1dist=4*8;
#pragma omp parallel for
for (j=0; j<ARRAY_SIZE; j++) {
// Prefetch "a" from memory to L2 and bring line as "exclusive"
   _mm_prefetch ((const void*)&a[j+L2dist],_MM_HINT_ET1);
//Prefetch "a" from L2 to L1 and bring line as "exclusive"
   _mm_prefetch ((const void*)&a[j+L1dist],_MM_HINT_ET0);
//Prefetch "b" from memory to L2 and bring line as "exclusive"
   _mm_prefetch ((const void*)&b[j+L2dist],_MM_HINT_ET1);
//Prefetch "b" from L2 to L1 and bring line as "exclusive"
   _mm_prefetch ((const void*)&b[j+L1dist],_MM_HINT_ET0);
  c[j] = a[j] + b[j];
}
```

FIG. 6.25

Prefetch intrinsics suitable for both Knights Landing and Knights Corner.

line into and also the state of the cache line (which is controlled using "hints"). Note that Knights Landing does not have nontemporal software prefetching. If a cacheline is brought into a cache level, it is also allocated in the lower level caches (L2 and/or MCDRAM cache).

Fig. 6.25 is a simple example showing how to use the **prefetch intrinsics**.

ADVICE: GATHER AND SCATTER INSTRUCTIONS ONLY WHEN IRREGULAR

In order to increase the performance of algorithms with irregular data access patterns, Knights Landing supports scatter and gather instructions as part of AVX-512. Gather can be thought of as multiple element size loads, where the elements are merged into a single-vector register. A second vector register is used to specify multiple addresses. A scatter is comprised of multiple element-sized stores, where one vector register holds the data to be stored, and another vector register specifies the multiple addresses for the stores. AVX2 supports gathers for XMM and YMM vectors but does not support scatter.

AVX-512 gather and scatter operation should only be used when the data needed is truly scattered in memory (not contiguous). If we have data that is contiguous in memory, but needs to be in a different order in the registers before computations are done, we should not use gather/scatter instructions to achieve the rearranging. It is better to load data using regular AVX-512 instructions and permute the data once it is in the SIMD (ZMM) registers. Chapter 12 includes an excellent "load plus permute" sequence because such sequences are more efficient than using gather instructions to combine the loading and permuting. The example in Chapter 12 involves complex numbers which are layed out sequentially in memory but need to be operated on in a different pattern than they are stored. Our example shows how to load, and permute, the incoming data to maximize performance. Similarly, the example shows how to organize results so that a masked store can be used instead of resorting to a scatter.

To maximize performance, use gather/scatter instructions only when the data is truly sparse in memory. If some data is contiguous in memory, but needs rearranging before computation, we should do regular loads and then use permute instructions. Such "load plus permute" sequences are more efficient than using gather instructions to combine the loading and permuting. Similarly, permute plus store will be more efficient than a scatter.

AVX-512F versions of gather and scatter are quite different from the IMCI implementation, which would only process the elements in a single cache line for each instruction. The AVX-512F version of the instructions completes the entire gather or scatter in a single instruction which makes them much faster than the IMCI instructions. There are several microarchitectural improvements that allow Knights Landing µarch to be faster than the Broadwell µarch for AVX2 gathers.

Gather and scatter instructions support various index, element, and vector widths. The AVX-512 flavors of gather and scatter use the mask registers to identify the lanes that should be loaded to, or stored from, the vector registers. AVX2 does not support scatter instructions or mask registers. In addition, AVX2 is limited to at most a 256-bit vector width. Knights Corner supports scatter and gather instructions with mask registers and 512-bit vector registers. For correct completion of a Knights Corner gather or scatter, the programmer would need to wrap the instruction in a test of the mask for 0, and a conditional branch.

For instance, consider the C code fragment shown in Fig. 6.26. The optimal AVX2 code would be similar to what is shown in Fig. 6.27; the optimal IMCI coding would be similar to what is shown in Fig. 6.28; the optimal AVX-512F coding would be similar to what is shown in Fig. 6.29. The AVX-512F code is more compact that the other instruction formats. In addition to using fewer instructions, the code can be executed faster than the alternate ISA sequences (IMCI or AVX2).

```
for (uint32 i=0; i < 16; i++)
      b[i] = a[indirect[i]];
```

FIG. 6.26

Looping on `b[i] = a[indirect[i]]`.

```
vmovdqu     ymm0, [rsp+0x1000]  ;; load indirect[]
vmovdqu     ymm3, [rsp+0x1020]  ;;   part 2
vpcmpeqd    ymm4, ymm4, ymm4    ;; setup mask
vmovdqa     ymm1, ymm4          ;;   part 2
vpgatherdd  ymm2, [rax+ymm0*4], ymm1
vpgatherdd  ymm5, [rax+ymm3*4], ymm4
vmovdqu     [rsp], ymm2         ;; store b[]
vmovdqu     [rsp+0x20], ymm5    ;;   part 2
```

FIG. 6.27

AVX2 code for `b[i] = a[indirect[i]]`.

```
 vmovaps     zmm0,[rsp+0x1000]   ;; load indirect[]
 kxnor       k1,k1               ;; setup mask for gather
gather_loop:
 vgatherdps  zmm2{k1},[rax+zmm0*4]
 vgatherdps  zmm2{k1},[rax+zmm0*4]
 jknzd       k1,<gather_loop>    ;; see if gather is done
 vmovaps     [rsp],zmm2          ;; store b[]
```

FIG. 6.28

IMCI code for `b[i] = a[indirect[i]]`.

```
 vmovups     zmm0, [rsp+0x1040]     ;; load indirect[]
 kxnor       k1, k0, k0             ;; mask for gather
 vpgatherdd  zmm1{k1}, [rsi+zmm0*4] ;; gather a[]
 vmovdqu32   [rsp], zmm1            ;; store b[]
```

FIG. 6.29

AVX-512 code for `b[i] = a[indirect[i]]`.

IMCI TO AVX-512: GATHERS/SCATTERS

The hardware implementation and the corresponding software (compiler generated code/intrinsics) for gather-scatter are different between Knights Corner and Knights Landing. Consider a simple *Indirect Access* Kernel Loop as shown in Fig. 6.30. This code snippet will explain the gather-scatter differences between Knights Corner and Knights Landing in detail.

In Fig. 6.30, the load of "a [b[j]]" is a "gather" operation and store of "a [b[j]]" is a scatter operation. The corresponding compiler generated code for Knights Corner and Knights Landing is shown in Figs. 6.31 and 6.32.

On Knights Corner, the gather/scatter implementation is "load per cacheline" which means a single gather and scatter instruction can load or store multiple elements from the same cacheline, due to which we see a "gather (scatter) loop" with branch on mask test in the compiler-generated code.

For example, if we look at the Knights Corner gather code in Fig. 6.31, there are two gather instructions wrapped in a loop with jkz/jknz. If the first gather instruction loads all elements from the same cacheline, then the "jkz" branch will be true and will exit the "gather loop" (since internally the "k2" mask is unset (set to zero) for every loaded element). If the elements are in multiple cache-lines, then it will loop until all the elements are loaded (mask (here "k2") controls the gather completion).

On Knights Landing, the gather/scatter implementation is "load (store) per element" which means every element is treated as an individual load (store) whether in the same cache line or not. But from the instruction point of view, gather/scatter is a single instruction as shown in Knights Landing code in Fig. 6.32 and need not be

```
int main() {
/* Memory Allocation*/
  a = (double *)_mm_malloc(sizeof(double)*(ARRAY_SIZE),64);
  b = (int *)_mm_malloc(sizeof(int)*(ARRAY_SIZE),64);
  c = (double *)_mm_malloc(sizeof(double)*(ARRAY_SIZE),64);

/* Initializing  the Arrays*/
for (k = 0; k < ARRAY_SIZE; k++) {
  a[k] = rand() % 20;
  b[k] = ARRAY_SIZE;
  c[k] = rand() % 20;
}
/* Calling the Sparse Function (which does Gathers/Scatters)*/
for (k = 0; k < NTIMES; k++)
  Sparse();
/* Memory Free*/
  _mm_free(a);
  _mm_free(b);
  _mm_free(c);
}

void Sparse() {
  int j;
  #pragma simd
  #pragma vector aligned
  #pragma omp parallel for
  for (j = 0; j < ARRAY_SIZE; j++) {
    //Gather (Load) "a" with multiple index loaded from b[j]
    c[j] = a[b[j]]; -- > Gather
    //Scatter (Store)  "a" with multiple index loaded from b[j]
    a[b[j]] = c[j]; -- > Scatter
  }
}
```

FIG. 6.30

Indirect Access Kernel Loop.

wrapped in any loop. Gathers can process two "elements" per clock, and scatter can process one "element" per clock.

The implementation for Knights Corner provides better performance than the AVX-512 implementation in Knights Landing if the number of cache lines touched in the gather is small (0, 1, or 2). The implementation in Knights Landing is likely to perform better than Knights Corner if many cache lines are accessed per gather. Please refer to the Knights Landing Optimization Manual for some more details.

Figs. 6.33 and 6.34 are simple examples (for the Sparse loop) showing how to use the "gather/scatter intrinsics" on Knights Corner and Knights Landing. There is a slight variation in the syntax for gather/scatter between Knights Corner and Knights Landing. But for both Knights Corner and Knights Landing, the arguments of the gather/scatter intrinsic are the same.

```
Gather: (datatype vindex, void const* base_addr, int scale)
```

Gathers starting at *base_addr* and offsets by elements in *vindex* (index is scaled by factor specified in *scale*). Scale depends on the data type of the element loaded.

```
Scatter: (void* base_addr, datatype vindex, datatype a, const int scale)
```

```
..B1.17:
        vloadunpackld (%r11,%r13,4), %zmm0{%k1}
```
Loading the multiple index values "b[i]". Note that the index for gather has to "integral" type
```
        kxnor     %k2, %k2
```
Setting k2 mask of all "1's" for Gather Operation
```
..L32:
        vgatherdpd (%r10,%zmm0,8), %zmm2{%k2}
        jkzd      ..L31, %k2    # Prob 50%
        vgatherdpd (%r10,%zmm0,8), %zmm2{%k2}
        jknzd     ..L32, %k2    # Prob 50%
```
Load (gather) the multiple values for the different loaded index to zmm2 [a[b[i]] . Internally mask "k2" is cleared (unset) for every loaded value. Mask "k2" of all 0's indicates gather completion
```
..L31:
        vmovaps   %zmm2, (%rbx,%r13,8)
        kxnor     %k3, %k3
```
Setting k3 mask of all "1's" for Gather Operation
```
      vloadunpackld (%r11,%r13,4), %zmm1{%k1}
```
Loading the multiple index values "b[i]". Note that the index for gather has to "integral" type
```
        addq      $8, %r13
        nop
..L34:
        vscatterdpd %zmm2, (%r10,%zmm1,8){%k3}
        jkzd      ..L33, %k3
        vscatterdpd %zmm2, (%r10,%zmm1,8){%k3}
        jknzd     ..L34, %k3
```
Store (gather) the multiple values from the different loaded index (zmm1) to mem addr Internally mask "k3" is cleared (unset) for every stored value. Mask "k3" of all 0's indicates scatter completion
```
..L33:
        cmpq      %rax, %r13
        jb        ..B1.17
```

FIG. 6.31

IMCI (Knights Corner) code for Fig. 6.30 Indirect Access Kernel Loop.

```
        vpxord    %zmm1, %zmm1, %zmm1
```
Clearing the contents of the zmm1 registers for Gather/Scatter Operation
```
        kxnorw    %k1, %k0, %k0
```
Setting k1 mask of all "1's" for Gather Operation
```
        kxnorw    %k2, %k0, %k0
```
Setting k2 mask of all "1's" for Scatter Operation
```
        vmovdqu   (%r11,%r13,4), %ymm0
```
Loading the multiple index values "b[i]". Note that the index for gather has to "integral" type
```
        vgatherdpd (%r8,%ymm0,8), %zmm1{%k1}
```
Load (gather) the multiple values for the different loaded index to zmm1 [a[b[i]] . Internally mask "k1" is cleared (unset) for every loaded value. Mask "k1" of all 0's indicates gather completion
```
        vmovups   %zmm1, (%r10,%r13,8)
        addq      $8, %r13
        vscatterdpd %zmm1, (%r8,%ymm0,8){%k2}
```
Store (scatter) the multiple values from the different loaded index (zmm1) to mem addr Internally mask "k2" is cleared (unset) for every stored value. Mask "k2" of all 0's indicates scatter completion
```
        cmpq      %r9, %r13
        jb          ..B1.17
```

FIG. 6.32

AVX-512 (Knights Landing) code for Fig. 6.30 Indirect Access Kernel Loop.

```
void Sparse() {
  int j;
  #pragma nounroll
  #pragma omp parallel for
  for (j=0; j<ARRAY_SIZE; j+=8) {
    //LOAD "b"
    __m512i Vb = _mm512_loadunpacklo_epi32 (
                      _mm512_undefined(), &b[j]);
    //GATHER "a"
    __m512d Vc = _mm512_i32logather_pd (Vb, &a[0], 8);
    //Store "c"
    _mm512_store_pd (&c[j], Vc);

    //Scatter "c" to "a"
    _mm512_i32loscatter_epi64 (&a[0], Vb, Vc, 8);
  }
}
```

FIG. 6.33

IMCI (Knights Corner) intrinsics code for Fig. 6.30 Indirect Access Kernel Loop.

```
void Sparse() {
  int j;
  #pragma nounroll
  #pragma vector aligned
  #pragma omp parallel for
  for (j=0; j<ARRAY_SIZE; j+=8){
    //LOAD "a"
    __m256i Vb = _mm256_loadu_si256 ((void*)&b[j]);
    //GATHER "a"
    __m512d Vc = _mm512_i32gather_pd (Vb, &a[0], 8);
    //STORE "c"
    _mm512_store_pd (&c[j], Vc);
    //Scatter "c" to "a"
    _mm512_i32scatter_pd (&a[0], Vb, Vc, 8);
  }
}
```

FIG. 6.34

AVX-512 (Knights Landing) intrinsics code for Fig. 6.30 Indirect Access Kernel Loop.

Scatters elements from "a" into memory starting at *base_addr* and offsets using index values in "`vindex`" (index is scaled by factor specified in *scale*). Scale depends on the data type of the element stored.

There are many intrinsic flavors of the gather/scatter instruction supporting the different data types and required conversions for the same. Chapter 12 has an excellent introduction to intrinsics, and information about the online Intrinsics Guide.

IMCI TO AVX-512: SWIZZLE INSTRUCTIONS

Swizzle is an operation to perform data element rearrangement/permutations of the source operands before execution. A temporary copy of the source operand with the swizzle is created which is fed to the ALU as the source of the operation. The temporary copy is destroyed after the operation.

On Knights Corner, the register/memory swizzle is available on all implicit loads and could be used to perform data replication via broadcasts or data conversion or permutations. There are some restrictions/rules on the data type for conversions, granularity for broadcasts, and the number of swizzle primitives available.

AVX-512, and hence Knights Landing, has no direct support of swizzles. The example in Fig. 6.35 leads to us showing the difference in the generated code for a swizzle operation on Knights Corner (IMCI) in Fig. 6.36 and Knights Landing (AVX-512) in Fig. 6.37.

Fig. 6.35 is a simple addition kernel with the "swizzle" operation performed on "a" before the ADD operation. The swizzle transformation is normally done on four element boundary (16-byte/32-byte). The code uses single-precision (32-bit) floating-point elements; thus the transformation is done on each of the four groups of packed 4× single-precision floating-point elements with swizzle parameter "BADC." The compiler-generated code for the intrinsics is shown in Figs. 6.36 and 6.37.

```
void test() {
  for (j=0; j<ARRAY_SIZE; j+=16){
    //LOAD "a"
     __m512 Va = _mm512_load_ps (&a[j]);
    //LOAD "b"
     __m512 Vb = _mm512_load_ps (&b[j]);
    // SWIZZLE
     __m512 Va_swizz =
       _mm512_swizzle_ps (Va, _MM_SWIZ_REG_BADC);

        //ADD
          __m512 Vc = _mm512_add_ps (Va_swizz, Vb);
        //STORE "c"
          _mm512_store_ps (&c[j], Vc);
        }
}
```

FIG. 6.35

Swizzle Intrinsics Code Example.

```
..B1.14:
        vmovaps   (%rdi,%rdx,4), %zmm1
Load "a"
        incb      %cl
        vmovaps   (%r13,%rdx,4), %zmm0
Load "b"
        vaddps    %zmm1{badc}, %zmm0, %zmm2
Perform "swizzle" operation on "a" and ADD to "b". Store result in "zmm2"
        nop
        vmovaps   %zmm2, (%r14,%rdx,4)
Store result in "c"
        addq      $16, %rdx
        cmpb      $6, %cl
        jb        ..B1.14
```

FIG. 6.36

IMCI (Knights Corner) code for Fig. 6.35 for Swizzle Intrinsics Code.

```
..B1.9:
        addb      $1, %cl
        vpshufd   $78, (%rdi,%rdx,4), %zmm0
Perform "swizzle" operation using "shuffle" instruction on "a" from memory
        vaddps    (%r13,%rdx,4), %zmm0, %zmm1
ADD swizzled "a" and "b" from memory. Store result in "zmm1"
        vmovups   %zmm1, (%r14,%rdx,4)
Store result in "c"
        addq      $16, %rdx
        cmpb      $6, %cl
        jb        ..B1.9
```

FIG. 6.37

AVX-512 (Knights Landing) code for Fig. 6.35 for Swizzle Intrinsics Code.

On Knights Corner (IMCI), we get a single instruction for the swizzle + ADD operation, but on Knights Landing (AVX-512), there are two instructions for the swizzle and ADD. There is no in-lane swizzle support in Knights Landing; thus we use a sequence with a shuffle and then ADD.

The "`$78`" in "`vpshufd`" instruction on Knights Landing is basically used as a 2-bit shuffle order to indicate the swizzle parameter ("BADC"). "`78`" in binary is `01 00 11 00`, which is "`1 0 3 2`" or "`B A D C`." Please note that, as shown, the "intrinsics" code for the "swizzle" operation on both Knights Corner (IMCI) and Knights Landing (AVX-512) is the same, and the generated code differs due to architectural differences.

IMCI TO AVX-512: UNALIGNED LOADS/STORES

Knights Landing, like recent Intel Xeon processors, uses a single instruction for an unaligned load/store and has no penalty for using an unaligned load/store instead of an aligned load/store. Knights Corner required a two instruction sequence, loadunpack**ld** and loadunpack**hd**, for unaligned accesses and therefore had a small penalty when using an unaligned load/store instead of an aligned load/store.

In Fig. 6.38, the load of arrays "a" and "b" starting at index "j+3" and "j+9," respectively, is unaligned. The best intrinsic code is shown in Figs. 6.39 and 6.40. The corresponding compiler-generated code is shown in Figs. 6.41 and 6.42. Note that for Knights Landing, even though array "c" is aligned, the compiler generates unaligned stores "`vmovups`" because there is no reason not to do so. Consider a simple Add kernel as shown in Fig. 6.38.

It is generally best to align arrays and data structures to the size of AVX-512 (i.e., 64-byte cache line boundaries) for Knights Landing or any processor. Refer to the online article Data Alignment to Assist Vectorization on different ways to align arrays and data structures in C/C++ and Fortran that works with the Intel Compiler—see *For More Information* at the end of this chapter.

```
static double a[ARRAY_SIZE],
              b[ARRAY_SIZE],
              c[ARRAY_SIZE];
int main() {
  for (k=0; k<NTIMES; k++)
    Add ();
}
void Add () {
  int j;
  #pragma nounroll
  for (j=0; j<ARRAY_SIZE; j++)
    c[j] = a[j+3] + b[j+9];
}
```

FIG. 6.38

ADD kernel with unaligned access.

```
for (j=0; j<ARRAY_SIZE; j+=8)  {
  /*load "a" starting from unaligned address*/
  __m512d Va_lo = _mm512_loadunpacklo_pd(
                      _mm512_undefined_pd(), &a[j+3]);
  __m512d Va = _mm512_loadunpackhi_pd (Va_lo, &a[j+3]);
  /*load "b" starting from unaligned address*/
  __m512d Vb_lo = _mm512_loadunpacklo_pd (
                      _mm512_undefined_pd(), &b[j+9]);
  __m512d Vb = _mm512_loadunpackhi_pd (Vb_lo, &b[j+9]);
  __m512d Vc =  _mm512_add_pd (Va, Vb);
   _mm512_store_pd (&c[j], Vc);
}
```

FIG. 6.39

IMCI (Knights Corner) intrinsic version of Fig. 6.38 ADD kernel code.

```
for (j=0; j<ARRAY_SIZE; j+=8) {
  /* load "a" starting from unaligned address */
  __m512d Va = _mm512_loadu_pd (&a[j+3]);
  /* load "b" starting from unaligned address */
  __m512d Vb = _mm512_loadu_pd (&b[j+9]);
  __m512d Vc =  _mm512_add_pd (Va, Vb);
  /* store "c" */
  _mm512_store_pd (&c[j], Vc);
}
```

FIG. 6.40

AVX-512 (Knights Landing) intrinsic version of Fig. 6.38 ADD kernel code.

```
vloadunpackld 24+a(%rax,%r8,8), %zmm0
```
→ load lower portion of the elements of array "a" starting from "addr" upto the first 64byte-aligned address
```
vloadunpackld 72+b(%rax,%r8,8), %zmm1
```
→ load lower portion of the elements of array "b" starting from "addr" to the first 64byte-aligned address
```
vloadunpackhd 88+a(%rax,%r8,8), %zmm0
```
→ load upper portion of the elements of array "a" starting from the first 64byte-aligned address following "addr"
```
vloadunpackhd 136+b(%rax,%r8,8), %zmm1
```
→ load upper portion of the elements of array "b" starting from the first 64byte-aligned address following "addr"
```
vaddpd     %zmm1, %zmm0, %zmm2
nop
vmovaps    %zmm2, c(%rax,%r8,8)
addq       $8, %r8
cmpq       %rbx, %r8
jb         ..B2.27
```

FIG. 6.41

IMCI (Knights Corner) code for Fig. 6.38 ADD kernel code.

```
vmovups    24+a(%rax,%r9,8), %zmm0
```
→ single load of all elements of array "a" starting from unaligned "addr". The "u" in "vmov**u**ps" denotes unaligned loads
```
vaddpd     72+b(%rax,%r9,8), %zmm0, %zmm1
```
→ load-op (single load of all elements of array "b" starting from unaligned "addr" + add to the register with elements of array "a")
```
vmovups    %zmm1, c(%rax,%r9,8)
→ Store the addition results in "c" array
addq       $8, %r9
cmpq       %r8, %r9
jb         ..B2.27
```

FIG. 6.42

AVX-512 (Knights Landing) code for Fig. 6.38 ADD kernel code.

IMCI TO AVX-512: DATA CONVERSION INSTRUCTIONS

Similar to swizzles, there are some differences in data conversion instructions between Knights Corner (IMCI) and Knights Landing (AVX-512).

As opposed to Knights Corner, Knights Landing does not support "on the fly" data conversions of source operands from a register/memory in a Load-Op. Also some of the IMCI intrinsics (`_mm512_ext*`) which take up/down conversion parameters will not work on Knights Landing if a "non-zero" up/down conversion is specified.

The code example in Fig. 6.43 helps us illustrate some conversion changes a bit more in detail. The compiler generated code for Fig. 6.43 on Knights Corner and Knights Landing is shown in Figs. 6.44 and 6.45.

```
int main()
{
  a = (short int *)  _mm_malloc(
       sizeof(short int)*(ARRAY_SIZE),64);
  b = (short int *)  _mm_malloc(
       sizeof(short int)*(ARRAY_SIZE),64);
  c = (float *)      _mm_malloc(
       sizeof(float)*(ARRAY_SIZE),64);
  /* Initializing */
  for (k=0;k<ARRAY_SIZE;k++) {
    a[k] =  0;
    b[k] =  0 ;
    c[k] =  0 ;
  }
  for (k=0;k<NTIMES;k++) {
    test();
  }
  _mm_free(a);
  _mm_free(b);
  _mm_free(c);
}

void test() {
        for (j=0; j<ARRAY_SIZE; j++)
                c[j] = a[j] + b[j];
}
```

FIG. 6.43

A simple ADD kernel, where arrays "a" and "b" are "short int (uint16)" and array "c" is "float" data type, respectively.

```
..B1.24:
        vmovdqa32 (%rdx, %rdi){sint16}, %zmm0
```

Up-convert 16bit integer elements to 32bit integer elements from memory and load 16 elements to zmm0 (up-convert and load array "a")

```
        vpaddd (%rdx, %r13){sint16},%zmm0, %zmm1
```

Add packed 32 bit integers in zmm0 to the up-converted 32 bit integers from memory and store in zmm1 (up-convert array "b" and ADD to array "a")

```
        addq       $32, %rdx
        vcvtfxpntdq2ps $0, %zmm1, %zmm2
```

Perform element by element conversion of packed 32bit integer elements in zmm1 to packed 32bit floating point elements and store in zmm2

```
        nop
        vmovaps    %zmm2, (%r14, %rcx, 4)
```

Store packed 32 bitfloating point elements to memory (store result to array "c")

```
        addq       $16, %rcx
        cmpq       $96, %rcx
        jb         ..B1.24
```

FIG. 6.44

IMCI (Knights Corner) compiler generated code for Fig. 6.43.

```
..B1.21:
        vpmovsxwd (%rdi,%rdx,2), %zmm0
```
Sign extend packed 16bit integers in array "a" to packed 32bit integers and store in zmm0
```
        vpmovsxwd (%r15,%rdx,2), %zmm1
```
Sign extend packed 16bit integers in array "b" to packed 32bit integers and store in zmm1
```
        vpaddd    %zmm1, %zmm0, %zmm2
```
Add packed 32-bit integers in zmm1 and zmm0 and store results in zmm2 ("a" + "b")
```
        vcvtdq2ps %zmm2, %zmm3
```
Convert packed 32bit integers in zmm2 to packed 32bit floating point elements and store in zmm3
```
        vmovups   %zmm3, (%r14,%rdx,4)
```
Store results in array "c"
```
        addq      $16, %rdx
        cmpq      $96, %rdx
        jb        ..B1.21
```

FIG. 6.45

AVX-512 (Knights Landing) compiler-generated code for Fig. 6.43.

With IMCI, the vector load and load-op (add) performs data conversion (short int → int) on the data read from memory and stored to the vector register. With AVX-512, the data conversion or sign extension (short int → int) is done first before the add operation, since there is no support to do on the fly data conversion in a vector load-op.

The corresponding intrinsics code on Knights Corner and Knights Landing for data conversions are different. The sample intrinsics code for the example is shown in Figs. 6.46 and 6.47.

The important thing to note here is the change in the intrinsics code between Knights Corner (IMCI) and Knights Landing (AVX-512) to do the data conversions. This intrinsics will basically generate the same assembly code as shown in Fig. 6.45. Please refer to the intrinsics guide for the different available data conversion intrinsics.

IMCI TO AVX-512: NONTEMPORAL STORES/CACHE LINE EVICTS

Memory blocks that are not reused in the immediate future are nontemporal memory accesses. Caching a nontemporal memory access could lead to wasted bandwidth, since those cache lines can be used to cache more important data. The *streaming nontemporal store* instructions avoid caching the data on a store (also known as Read for Ownership) by eliminating any data read from memory to cache on a store and thus saving bandwidth.

```
for (j=0; j<ARRAY_SIZE; j+=16){
//LOAD "a" with up-conversion
  __m512i Va = _mm512_extload_epi32 (&a[j],
               _MM_UPCONV_EPI32_SINT16,
               _MM_BROADCAST32_NONE, 0);
//LOAD "b" with up-conversion
  __m512i Vb = _mm512_extload_epi32 (&b[j],
               _MM_UPCONV_EPI32_SINT16,
               _MM_BROADCAST32_NONE, 0);
//ADD
  __m512i Vc = _mm512_add_epi32 (Va, Vb);
//Convert int to float
  __m512 Vc_float =
  _mm512_cvtfxpnt_round_adjustepi32_ps (Vc,
               _MM_ROUND_MODE_DEFAULT,
               _MM_EXPADJ_NONE);
//STORE "c"
  _mm512_store_ps (&c[j], Vc_float);
}
```

FIG. 6.46

IMCI (Knights Corner) intrinsics code for Fig. 6.43.

```
for (j=0; j<ARRAY_SIZE; j+=16) {
      //LOAD "a"
  __m256i Va = _mm256_load_si256((__m256i*)&a[j]);
      //LOAD "b"
  __m256i Vb = _mm256_load_si256((__m256i*)&b[j]);
     //Up Convert "a" (short int -> int)
  __m512i Vaa =  _mm512_cvtepi16_epi32 (Va);
     //Up Convert "b" (short int -> int)
  __m512i Vbb =  _mm512_cvtepi16_epi32 (Vb);
     //ADD
  __m512i Vc = _mm512_add_epi32 (Vaa, Vbb);
     //Convert int to float
  __m512 Vc_float = _mm512_cvtepi32_ps (Vc);
     //STORE "c"
  _mm512_store_ps (&c[j], Vc_float);
}
```

FIG. 6.47

AVX-512 (Knights Landing) intrinsics code for Fig. 6.43.

The actual implementation of streaming nontemporal stores vary with different architectures. On Knights Corner, the streaming stores avoid reading the original contents of the cache line (no Read) since the cache line will be overwritten, but the stores happen to L2 cache. It did not bypass all the levels of cache.

On Knights Landing, the streaming stores behave similar to Intel Xeon processors, where the instruction avoids reading original contents of the cache line (no Read) and also bypasses all levels of caches (not MCDRAM if configured as cache).

The compiler generates streaming store instructions with compiler switch (-opt-streaming-stores always) or pragma (#pragma vector nontemporal) only when the loop can be vectorized and aligned unit-strided vector unmasked stores can be generated.

Consider a simple example in Fig. 6.48. Note the hint given to compiler to generate streaming stores using the "`#pragma vector nontemporal`." The compiler-generated code for code in Fig. 6.48 on Knights Corner and Knights Landing is shown in Figs. 6.49 and 6.50.

On Knights Corner, the compiler with the streaming store hint generates nontemporal streaming stores "`vmovnrngoaps`" which is a nonglobally ordered store of vector register contents into destination memory with No-Read hint.

Knights Corner also provides special instructions for evicting cache lines from different cache levels (L1 Cache—clevict0, L2 Cache—clevict1). The compiler

```
int main() {
  a = (float *)_mm_malloc(
        sizeof(float)*(ARRAY_SIZE),64);
  b = (float *) _mm_malloc(
        sizeof(float)*(ARRAY_SIZE),64);
  c = (float *) _mm_malloc(
        sizeof(float)*(ARRAY_SIZE),64);
/* Initializing */
  for (k=0;k<ARRAY_SIZE;k++) {
    a[k] =  0;
    b[k] =  0 ;
    c[k] =  0 ;
  }
  for (k=0;k<NTIMES;k++)
    test();
  _mm_free(a);
  _mm_free(b);
  _mm_free(c);
}

void test() {
#pragma vector nontemporal
#pragma vector aligned
for (j=0; j<ARRAY_SIZE; j++)
  c[j] = a[j] + b[j];
}
```

FIG. 6.48

Nontemporal data access code example.

```
..B1.36:
        vmovaps    (%rdi,%rax,4), %zmm0
Vector load of array "a"
        vaddps     (%r13,%rax,4), %zmm0, %zmm1
Vector load of array "b" and add to array "a" in zmm0
        vmovnrngoaps %zmm1, (%r14,%rax,4)
Store the Add result to "c". Non-temporal Streaming Store generated
        clevict1   (%r14,%rax,4)
Evict the non-temporal store cache line from L2
        addq       $16, %rax
        cmpq       $1600, %rax
        jb          ..B1.36
```

FIG. 6.49

IMCI (Knights Corner) compiler-generated code for Fig. 6.48.

```
..B1.32:
        vmovups    (%rdi,%rax,4), %zmm0
Vector load of array "a"
        vaddps     (%r14,%rax,4), %zmm0, %zmm1
Vector load of array "b" and add to array "a" in zmm0
        vmovntps   %zmm1, (%r15,%rax,4)
Store the Add result to "c". Non-temporal Streaming Store generated
        addq       $16, %rax
        cmpq       $1600, %rax
        jb         ..B1.32
```

FIG. 6.50

AVX-512 (Knights Landing) compiler-generated code for Fig. 6.48.

generates these instructions on loops that demonstrate nontemporal behavior (since on Knights Corner the nontemporal stores happen to memory through L2 cache), and it is possible to generate them because the vector length is equal to the cache line size.

On Knights Landing, the compiler, using a streaming store hint, generates nontemporal streaming stores similar to those on Intel Xeon processors. The nontemporal stores do not do an RFO and bypass all levels of cache (L1 and L2) on the store. There is no support for "cache line eviction" instructions on Knights Landing.

The corresponding intrinsics for the code in Fig. 6.48 on Knights Corner and Knights Landing are shown in Figs. 6.51 and 6.52. The main difference between the two is the "streaming store" intrinsics. The "`clevict`" instructions were Knights Corner specific and are not part of AVX-512.

```
for (j=0; j<ARRAY_SIZE; j+=16) {
  //LOAD "a"
  __m512 Va = _mm512_load_ps (&a[j]);
  //LOAD "b"
  __m512 Vb = _mm512_load_ps (&b[j]);
  //ADD
  __m512 Vc = _mm512_add_ps (Va, Vb);
  //Streaming Store
  _mm512_storenrngo_ps (&c[j], Vc);
  //Clevict
  _mm_clevict (&c[j], 2);
}
```

FIG. 6.51

IMCI (Knights Corner) intrinsics version for Fig. 6.48.

```
for (j=0; j<ARRAY_SIZE; j+=16){
  //LOAD "a"
  __m512 Va = _mm512_load_ps (&a[j]);
  //LOAD "b"
  __m512 Vb = _mm512_load_ps (&b[j]);
  //ADD
  __m512 Vc = _mm512_add_ps (Va, Vb) ;
  //Streaming Store
  _mm512_stream_ps (&c[j], Vc);
}
```

FIG. 6.52

AVX-512 (Knights Landing) intrinsics version for Fig. 6.48.

SUMMARY

As with every design, understanding the Knights Landing μarch offers insights into how to extract maximal performance. With proper parameters, applications can remain portable and performance portable by parameterizing around tuning knobs for areas of sensitivity. The most widely used will be knobs to affect the number of cores used, the number of threads per core used, working set sizes, data layout including page size options, numerical accuracy options, SIMD instruction sets to target, various forms of affinity, machine configuration, and compiler options (such as prefetching aggressiveness). Compiler writers use details on optimal instruction choices and generally shield most of us from needing to deal with such trade-offs directly. Yet, users of intrinsics (Chapter 12) will find such information valuable as well. Hopefully, this chapter, and more broadly this section of the book (Chapters 1–5), gave valuable insights into these choices as well as other sensistivities that may lead to effective detailed optimization work. Our next section of the book (Chapters 7–19) shifts to programming concerns directly, followed by the final section of the book which shares experiences from real applications.

FOR MORE INFORMATION

- Intel® Intrinsics Guide with Knights Landing specific intrinsics selected, http://lotsofcores.com/KNLintrinsics.
- Intel® 64 and IA-32 Architectures Software Developer Manuals, http://lotsofcores.com/Intel_ISA1.
- Intel® Architecture Instruction Set Extensions Programming Reference, http://lotsofcores.com/Intel_ISA2.
- Intel® Software Development Emulator (Intel® SDE), http://lotsofcores.com/Intel_SDE.
- Data Alignment to Assist Vectorization, https://software.intel.com/articles/data-alignment-to-assist-vectorization.
- A methodology for tuning prefetching: High-Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, Chapter 21—Prefetch Tuning Optimizations, ISBN 978-0-12-803819-2.
- How to detect Knights Landing AVX-512 support (Intel Xeon Phi processor), https://software.intel.com/articles/how-to-detect-knl-instruction-support.
- Glibc 2.23 patch to add Prefer_MAP_32BIT_EXEC to improve branch prediction on Silvermont, https://sourceware.org/bugzilla/show_bug.cgi?id=19367.
- Pointers to additional prefetching advice, http://lotsofcores.com/prefetch.

SECTION

Parallel programming II

This section focuses on application programming with consideration for the scale of many-core. Chapter 7 discusses when code changes for parallelism can be an evolution versus a revolution. Chapter 8 covers tasks and threads with OpenMP and TBB in particular. Chapters 9–12 include an overview of many data parallel vectorization tools and techniques. Chapter 13 covers libraries. Chapter 14 discusses profiling tools and techniques including roofline estimation techniques and ways to pick which data structures to place in the MCDRAM. Chapter 15 covers MPI. Chapter 16 takes a look at a collection of "rising stars" that provide PGAS style programming — including OpenSHMEM, Coarray Fortran, and UPC. While these will not generally be the most efficient parallel programming techniques for Knights Landing, there is a lot of merit in exploring this space during the upcoming years starting with Knights Landing. Chapter 17 discusses the emerging benefits of parallel graphics rendering using Software-Defined Visualization libraries. Chapter 18 (Offload to Knights Landing) discusses offload style programming including programming details on how to offload to a Knights Landing coprocessor, or across the fabric to a

Knights Landing processor. Chapter 18 is a "must read" if you have code that used Knights Corner offload methods, or if you will use Knights Landing in offload mode. We discuss "offload over fabric" which may be of interest for retaining offload code investments and maximizing application performance in clusters with a mix of multicore and many-core processors. Chapter 19 discusses Power Analysis on Knights Landing enabling insight into the growing emphasis on energy efficiency.

This book has three sections: I. Knights Landing, II. Parallel Programming, and III. Pearls. The book also has an extensive Glossary and Index to facilitate jumping around the book.

CHAPTER

Programming overview for Knights Landing

7

What is new with Knights Landing in this chapter?

Knights Landing is a processor fully able to benefit from all standards for exploiting the full capabilities of processors. More attention to optimizations, and tuning for parallelism, is warranted than for a less-parallel processor.

Programming Knights Landing should be approached as one would program any modern high-performance processor. Getting maximal performance from Knights Landing is largely the same challenge as with any processor, with added opportunity because the top performance is extremely high.

Breaking down the challenge of effective parallel programming can be done many ways. The National Energy Research Scientific Computing Center (NERSC), home to the Cori supercomputer based on Knights Landing, described the work as consisting of four elements:

- Manage Domain Parallelism
- Increase Thread Parallelism
- Exploit Data Parallelism
- Improve Data Locality

We could describe these casually as MPI, OpenMP/Threading Building Blocks (TBB), vectorization, and data layout and still capture the essence of the recommendations as it would be for most developers targeting Knights Landing. Of course, there are many variations on these. PGAS models (Chapter 16) may seem to blend elements. Keeping clear thinking about our approach for each element will help us manage the most expensive element of parallel computing: communication (moving data around).

We describe parallel programming, in terms of the four elements and their characteristics in Fig. 7.1. The first three elements address three key opportunities for parallelism in a modern machine: nodes, cores/threads, and vectors. The last element affects them all—attention to minimization of communication, that is, data movement.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00007-7

Category	Characteristics	Methods
Domain parallelism	Heavy computation in the domain, *limited* communication of data between domais	MPI (Chapter 15) TBB flow graph (Chapter 8) PGAS (Chapter 16) Frameworks (e.g., Hadoop, spark) - beyond the scope of this book
Thread parallelism	Data sharing between threads, high degree of (mostly) independent tasks. Potential for excessive synchronization going unnoticed	OpenMP (Chapter 8) TBB tasking (Chapter 8) PGAS (Chapter 16)
Data parallelism	Same algorithm/operation applied to multiple data items	Libraries (Chapters 13, 17) Auto vectorization (Chapter 9) OpenMP "simd" (Chapter 9) Code transformation (analysis tool to help: Chapter 10) AVX-512 intrinsics (Chapter 12)
Data locality	Arranging algorithms to minimize data movement Arranging data to reduce the need for data movement Arranging data for temporal reuse, to better utilize caches so as to reduce data movement all the way from memory	AOS to SOA, and related transforms; SDLT (Chapter 11) Code transformations (analysis tools to help: Chapter 14)

FIG. 7.1

A way to think about the elements of parallel programming.

TO REFACTOR, OR NOT TO REFACTOR, THAT IS THE QUESTION

"Code Modernization" is a term thrown around to encompass a wide range of activities we can do to update an application to make maximal use of today's machines. There is not a hard and fast definition, but there is one question that is on everyone's mind about "code modernization": How much of my application do I need to change? Of course, the answer is "it depends." Ultimately, the answer rests in whether an application needs to be refactored or not.

Factoring an application, and its algorithms, to decompose into three levels of parallelism is best done with data locality in mind. The proper factoring of an application is the very essence of parallel programming design. Poor factoring cannot be made up for by hard work within the confines of one level of parallelism.

Choosing a proper factoring of an application is fundamental for exploiting three levels of parallelism and minimizing data movement. Poor choices in factoring can easily be a major limiter of performance.

The three biggest drivers for refactoring programs coincide with the largest changes that have occurred in the past few decades in supercomputers: massive growth in all levels of parallelism, and radical increases in computational performance without a corresponding improvement in the performance of memory systems and other data movement capabilities. The growth in parallelism motivates us to cleanly separate work to be done into activities that can run independently. Barriers, or other synchronizations, are the enemy of scaling (using more core/threads/vectors). The growing gap between processor and memory performance means that higher performance can be driven more by reductions in communication (data movement) than in reduction of computations. This can be at odds with many design decisions, including data layout, made in an era when computations were more important to eliminate than memory accesses. That day is long gone!

EVOLUTIONARY OPTIMIZATION OF APPLICATIONS

A natural place to start is to examine if our program is open to optimization via an evolutionary approach, which does not require radical restructuring (refactoring). Fundamentally, we want to know whether this will reach an acceptable percentage of the capabilities of the machine. Fig. 7.2 illustrates the sort of iterative process we would employ. We are hoping to find a "hotspot" that we can tune without having to restructure substantial amounts of the application. Using tools to find MPI issues may have us optimize at the domain level of parallelism. Using tools to find node level issues may have us optimize at the thread or data parallelism level. Data locality challenges may surface doing either. Fig. 7.2 illustrates such a tool-driven approach for improving threading and vectorization to achieve better data parallelism.

This evolutionary approach can gain substantial speed-ups if your code has its parallelism locked up in a limited numbers of hotspots, each with substantial computations and few dependencies to limit parallelism. In fact, if our code does not have those qualities but could be rewritten to have such qualities—that would motivate a revolutionary approach with whatever refactoring is needed.

A key to deciding whether we can stop or not hinges on how close this technique will get us to the maximum we can expect for a particular machine. Chapter 10 has Section *Advisor Roofline Report* dedicated to thinking about this problem. We start by wondering if we are limited by computations or by memory. Sometimes it is obvious: European Options Pricing, for instance, may compute 2^{18} paths per option. That is overwhelmingly "CPU-bound" and not "Memory-bound." In such cases, most of our energy should go into reducing the time needed to do the computations. We can

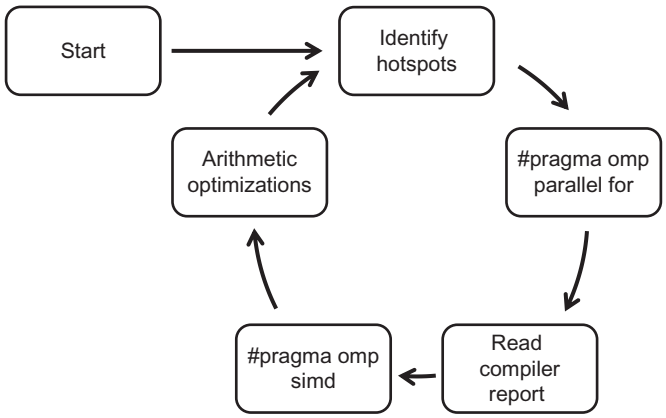


FIG. 7.2

Evolutionary optimization approach illustrated with OpenMP in mind.

compute the FLOPs needed, compare to the FLOP/s rating of our target machine, and determine the least time to compute the problem from an ideal standpoint. We know that:

$$\text{seconds} = \text{FLOPs needed by app.}/\text{machine's peak FLOP}/\text{s}$$

is ideal and not realizable. However, for a "CPU-bound" application, it is one possible "roofline" model.

A "roofline" model refers to an estimate of the highest obtainable performance given some "back of the envelope" estimation that includes some consideration for bottlenecks.

"Memory-bound" applications have a corresponding "roofline" model which involves comparing the amount of data needed to be processed versus the bandwidth of the machine. Poor data locality can easily force an application to need far more bandwidth if data has to be fetched from memory more than once. If evolutionary optimization cannot approach the appropriate "roofline" model, then revolutionary methods may be the key to higher performance. However, even if we reach our "roofline," there can always be new insights that yield a new algorithm. Finding a superior algorithm can easily trump evolutionary optimizations.

REVOLUTIONARY OPTIMIZATION OF APPLICATIONS

New algorithms can be revolutionary. For instance, if we can compute the same answer but with less algorithmic complexity, we can easily see gains in performance. Similarly, an algorithm with better data handling (e.g., more cache friendly, more locality of

reference in data accesses, less irregularity in data access) can yield substantial gains in performance. Fig. 7.3 illustrates an evolutionary approach leading to algorithm change.

KNOW WHEN TO HOLD'EM AND WHEN TO FOLD'EM

No one answer fits every application. Fig. 7.4 combines this into an approach that fully embraces our answer of "it depends." We offer a few closing thoughts:

- Adding pragmas and crossing fingers rarely solves the problem.

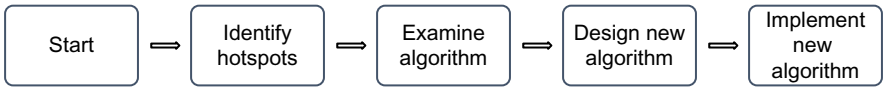


FIG. 7.3

Revolutionary optimization approach.

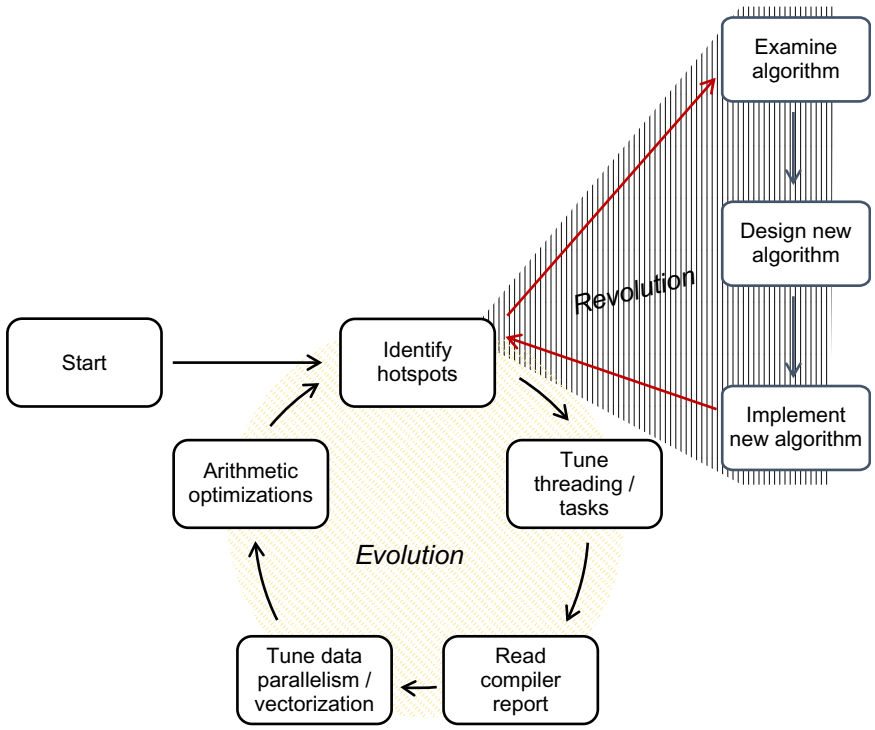


FIG. 7.4

Evolutionary optimization approach combined with revolution as needed.

- Building a "roofline" estimation of the best a given implementation can do on a particular machine can guide "evolutionary" optimization—especially to decide when to consider ourselves "good enough" (or in need of a revolution!).
- Consider how hardware should be used before worrying about coding.
- Choosing a programming model/methodology may restrict algorithmic choice.
- Revisit algorithms and throw out assumptions. A parallel implementation of a "slower" algorithm may be faster when run in parallel.
- Investment in "revolutionary" code today yields improved performance, better science, and a path to long-term "evolutionary" development.

Hopefully the remainder of the book will arm you with information that will guide your choice of evolution versus revolution. We have also invested in sharing proven real work examples in our prior two "Pearls" books, and in the "Pearls" section (Chapters 20-26) of this book. We believe in learning from experience, and from the experiences of others.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this article.

- Intel Modern Code website, https://software.intel.com/en-us/modern-code.
- *Getting ready for KNL? Take a Lesson from NERSC on Optimization*, HPCWire, Feb. 10, 2016, http://www.hpcwire.com/2016/02/10/nersc-getting-ready-knl.
- High-Performance Parallelism Pearls, Volume 1 and Volume 2: Multicore and Many-core Programming Approaches, Morgan Kauffman, 2015, edited by Jim Jeffers and James Reinders. Numerous examples of effective code optimization techniques in real world situations using multi- and many-core processors.
- Intel Xeon Phi Users Group, multiple working groups, http://www.ixpug.org with code repositories at https://github.com/IXPUG.

CHAPTER

Tasks and threads

8

It was tempting to not include this chapter. After all, Knights Landing is simply a high core count processor from Intel. That means it simply works with task/thread models like any other Intel processor. Learning these models in depth is already the subject of many books.

What is new with Knights Landing in this chapter?

Knights Landing is a processor fully able to benefit from all standards for exploiting the full capabilities of processors.

However, the unprecedented magnitude of parallelism on a Knights Landing processor means that this is a very important topic. Therefore, we have a short chapter to emphasize the key techniques we anticipate being most popular on Knights Landing. However, the compatibility of Knights Landing means that far more is possible than we can possibly mention in a short chapter.

MPI (Chapter 15) will certainly be the by far the most common programming model to connect multiple Knights Landing processors. While it is possible to employ one rank per core, most applications will employ an additional model for thread/task parallelism for the many-cores on a Knights Landing, and the most popular of these will be OpenMP for Fortran/C applications and threading building blocks (TBB) for C++ applications.

The advice in this chapter is this: you need lots of task-level parallelism and you should first consider using OpenMP, Fortran 2008 DO CONCURRENT, and TBB. Alternatives such as direct use of pthreads can deliver excellent performance results, but the limitations in terms of being portable and effort maintaining can be substantial. A problem, which is getting much attention in recent years, is the limitations of scaling caused by barrier synchronization. An emerging school of thought is to encourage programming that puts more focus on data flow instead of just control flow. Therefore, we mention the popular flow graph capability of TBB as well as the new hStreams programming capabilities. Altogether, the numerous methods to achieve scaling on Knights Landing are enough to help you have sufficiently high amounts of task-level parallelism, and the resulting code will work on all processors.

It is important to understand that these tasking/threading models dispatch only within a single shared memory space. The threads and tasks created on a Knights Landing thread will always stay somewhere within a single shared memory space, which for most machines will be a single Knights Landing processor.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00008-9

In the first chapter, we stressed the importance of many threads on Knights Landing in order to take advantage of its highly parallel nature. These threads are critical to keep Knight Landing busy and productive.

Loops are the first place most developers look to create tasks because considering every iteration of a loop to be a task can often keep a large number of threads active. Loops usually get special treatment to avoid creating a thread for every iteration since the number of iterations can be much larger than the number of threads on a machine. There is no sense in creating a million threads on Knight Landing. In fact, it would cause a problem in terms of overhead and memory usage. Instead, we like to think of creating the *opportunity* for a million threads if the loop has a million iterations. To express the opportunity for parallelism, OpenMP has a parallel loop directive (`PARALLEL DO`), Fortran has `DO CONCURRENT`, and TBB has a parallel for template (`parallel_for`) and flow graph support (`tbb::flow`). Loops are special and often the easiest place to find opportunities for parallelism, but in OpenMP, a common mistake is to enter and exit parallelism at loop boundaries by using "omp parallel do" or "omp parallel for" instead of using "omp parallel" to create a region with multiple "omp do" or "omp for" directives within the parallel region. Frequently exiting and restarting of parallelism can lead to scaling issues due to excessive implicit synchronization.

Other than parallelization of loops, explicitly creating parallel tasks requires more thinking and generally source code changes in diverse locations within an application. Both loop level parallelism and task parallelism are provided by the various models, but loop level parallelism is the most used, easiest to learn, and easiest to understand in a program.

If you choose to have multiple MPI ranks (see Chapter 15) on each Knights Landing, then the number of threads per rank will be large but need not be huge. The best option is likely involve creating fewer ranks each with enough threads to supply at least one thread per core. A typical usage with MPI might be 1-12 ranks on a Knights Landing to service about 72 cores (288 threads available) so that threading in each rank needs to provide 12-144 threads per rank. A program with a single rank per Knights Landing would likely need to have at least one thread per core created to best use the available processing capability. The importance of using some form of threading is very evident with and without use of MPI. More discussion of MPI programming can be found in Chapter 15.

As a programmer, you should program in tasks, not threads. What we mean by that is: the programmer should expose parallelism and share the opportunities for parallelism as tasks, but the work to map tasks to threads should not be encoded into an application. There are several reasons for this, including the difficulty of doing so, inflexibility for scaling on future hardware, and the fact that many good options now exist to program in tasks. It is very unwise to mix the concept of exposing tasks with the effort to map tasks to hardware threads. Separated, creation of tasks is the job of the application writer, and mapping those tasks onto hardware is the job of a tasking package such as OpenMP, or Intel TBB. Similarly, the Intel Math Kernel Library (MKL) hides pthreads by using OpenMP, which in turn is built using pthreads. Use of native threads (e.g., pthreads) directly by an application is the parallel

programming equivalent of writing in assembly language, complete with managing the contents of the machine registers. The advice to use tasks, not threads, is strictly an argument about how to be efficient at programming but still get great results that are maintainable and portable. The choice remains yours because Knights Landing supports all these choices. After all, OpenMP, TBB, and MKL are all just libraries that use pthreads on Linux (or Windows threads on Windows).

OpenMP, FORTRAN 2008, INTEL TBB, INTEL MKL

OpenMP, Fortran 2008, and TBB are standards to create task optimized programs. MKL is mentioned because it uses OpenMP internally and fills the same needs whenever one of its math algorithms is utilized by an application. MKL is covered in Chapter 13. We will discuss OpenMP, Fortran 2008, and TBB in this chapter. The table in Fig. 8.1 offers a brief comparison of the models.

Choosing between the models depends on your needs. OpenMP is well established, and caters to Fortran and C programming but does not cater to C++ programming and is not completely composable. Fortran 2008 is obviously Fortran only, with support appearing in most Fortran compilers, and is a full ISO standard. TBB caters to C++ programming, does not support Fortran or C, is composable, and is the only of the three that requires no compiler modifications.

IMPORTANCE OF THREAD POOLS

Applications should ultimately rest on top of thread pools so as to avoid the high overhead of creating and destroying threads repeatedly. If you are using pthreads, you should create all the threads you want only once and then reuse them through the life of the application. Fortunately, this is what OpenMP, TBB, and Fortran 2008 do automatically. Solutions for load balancing plus scheduling algorithms, to map work to threads, are details taken care of by well-engineered and tuned schedulers you can rely on from OpenMP, TBB, and Fortran 2008. In general, one software thread per hardware thread (four hardware threads per core) is the right upper bound for Knights Landing. This is a parameter worth tuning is the use of a dedicated core for operating system routines—this can be particularly useful when offload is being used to target a Knights Landing. The general rule of thumb is to use N cores natively, or $N-1$ cores if using them via offload because one core is busy doing data movement. Spreading the threads out evenly over the cores is done using affinity, which is important for OpenMP and TBB. All applications should use thread pools, either manually with pthreads or, preferably, automatically with OpenMP, TBB, or MKL.

OpenMP

OpenMP is a set of directives to a compiler that can be ignored and the program should simply work in a nonparallel mode (sequential). When a compiler recognizes OpenMP directives (requires the `-openmp` switch on the Intel compilers), then

	OpenMP	Fortran 2008	Intel® TBB
Website	openmp.org	fortranwiki.org	opentbb.org
First available	1997	2010	2006
Languages supported	Fortran, C, and C++	Fortran	C++
Summary	Directives for parallelism in Fortran and C	Language keywords/ syntax for Fortran	Template library for parallelism in C++
Approach	Standardization of compiler directives to add parallelism to Fortran and C, runtime library	Extends Fortran for task parallelism via language extensions, runtime library	Compiler independent C++ template library, runtime library
Specification	Open specification by OpenMP	ISO/IEC 1539-1:2010 standard	Open source project started by Intel
Features	Parallel loops, tasking model, portable locks, vectorization, target (offload)	Fortran language with extensions for concurrent programming	Parallel loops and algorithms, tasking model, portable locks, pipeline and flow graph models, scalable memory allocator
Unique features	Not completely composable	`DO CONCURRENT`, Coarrays	Can work with any compiler (does not require compiler changes), pipeline and flow graph models, scalable memory allocator

FIG. 8.1

OpenMP versus Fortran 2008 versus Intel® TBB.

the directives are interpreted to give direction on how to create parallel tasks in order to speed execution of a program through parallelism. OpenMP has vectorization (simd) directives and those will be covered separately in Chapter 9. OpenMP has also been extended with offload (target) directives and those will be covered separately in Chapter 18. For creating tasks, OpenMP offers a large variety of directives for invoking parallelism. The most widely used are the directives that simply distribute a loop in parallel. OpenMP offers enough options to have inspired quite a number of books dedicated to the topic as well as online training and reference materials. We will briefly cover only the essentials with an eye toward those that are most popular.

PARALLEL PROCESSING MODEL

A program containing OpenMP directives begins execution as a single thread known as the initial thread of execution. This initial thread executes sequentially until the first parallel construct is encountered. The `PARALLEL` (`!$OMP PARALLEL` or `#pragma omp parallel`) directive defines the extent of the parallel construct. When the initial thread encounters a parallel construct, it creates a team of threads (a thread pool), with the initial thread becoming the master of the team. All program statements enclosed by the parallel construct are executed in parallel by each thread in the team, including all routines called from within the enclosed statements.

The statements enclosed lexically within a construct define the static extent of the construct. The dynamic extent includes all statements encountered during the execution of a construct by a thread, including all called routines.

When a thread encounters the end of a structured block enclosed by a parallel construct, the thread waits until all threads in the team have arrived. When that happens, the team is dissolved, and only the master thread continues execution of the code following the parallel construct. The other threads in the team enter a wait state until they are needed to form another team. You can specify any number of parallel constructs in a single program. As a result, thread teams can be created and dissolved many times during program execution. Implementations will actually maintain a thread pool after the first creation so that creating and dissolving thread teams is not an expensive operation.

DIRECTIVES

OpenMP directives are disguised differently in C and C++ versus Fortran but both with the intent of being invisible to compilers that lack OpenMP support. For C and C++, an OpenMP directive will begin with `#pragma omp`. In Fortran, an OpenMP directive will begin with `!$OMP`. The most commonly used OpenMP directives are listed in Fig. 8.2.

SIGNIFICANT CONTROLS OVER OpenMP

One thing that characterizes OpenMP is the many options available to tune as part of performance-oriented work. Aside from the many clauses available to augment simple OpenMP directives, there are a number of OpenMP controls with wide-reaching implications. In general, OpenMP controls are available as function calls to be made within a program and via environment variables, which can be set and modified without changing the program. Function calls overrule environment variables. Use of environment variables is especially popular for describing items that we may wish to vary from machine to machine. The compiler documentation contains a long list of these controls with details on their function. Fig. 8.3 lists a few key controls. Those starting with "OMP_" are in the OpenMP standard; those starting with "KMP_" or "MIC_" are Intel specific controls not mandated by the standard.

Directive name (put after `#pragma omp` or `!$OMP`)	Description
`PARALLEL` and `END PARALLEL`	Defines a parallel region where all threads are executing everything, not just the "master" thread
`PARALLEL DO` and `END PARALLEL DO`	A shortcut for a PARALLEL region that contains a single DO directive
`MASTER` and `END MASTER`	Defines a serial region where only the master thread is executing
`CRITICAL` and `END CRITICAL`	Defines a serial region where only one thread can run at a time
`THREADPRIVATE`	Makes the named COMMON blocks or variables private to a thread. The list argument consists of a comma-separated list of COMMON blocks or variables
`TARGET`	Offload support useful for Knights Landing coprocessors, or using an offload model "over fabric" to Knights Landing processors, see Chapter 18

FIG. 8.2

Most used OpenMP directives ("END..." used with Fortran).

Controlling AFFINITY, NESTING (see hot teams feature), and BLOCKTIME can have a profound impact on performance. Serious performance tuning of OpenMP needs to comprehend and use these controls.

These controls also control parallelism generated by the Intel compiler, using the `-parallel` (`/Qparallel` on Windows) compile option, including parallel code generated from use of "do concurrent."

OpenMP NESTING—USE HOT TEAMS

OpenMP is unfortunately not fully composable, which can be a serious limitation when compared with the other abstract parallel programming models discussed in this chapter. Nesting of OpenMP can create explosive numbers of threads in recursive situations, which rapidly exhaust system resources, especially stack space, and require that the program be shut down. To avoid such issues, OpenMP defaults for the maximum number of levels of parallel nesting that will be activated are generally set to 1 (no nesting). This means, by default, an OpenMP thread encountering a nested OpenMP parallel region will serialize the section. It is possible, however, to override this behavior with an environment variable `OMP_NESTED=TRUE` or function call `omp_set_nested(true)`. Nesting OpenMP parallel regions in this manner is often not recommended, due to the repeated high overheads of creating and destroying the threads involved. These costs are prohibitively expensive when the nested

Environment variable name	Description
LD_LIBRARY_PATH and MIC_LD_LIBRARY_PATH	Specifies the path for shared (.so) library files. Needs to point to where the OpenMP library is located. (MIC_LD_LIBRARY_PATH is for offload – see Chapter 18)
OMP_NUM_THREADS	Sets the maximum number of threads to use for OpenMP parallel regions if no other value is specified in the application. Best values can range from 1–4 times the number of cores you wish to use. The number of cores to use should be the number of cores on Knights Landing unless we are using offload, in which case we decrease by one to allow the offload code in the OS a place to work, or if you choose to use less because you are running multiple MPI ranks on Knights Landing
KMP_AFFINITY	Enables runtime library to bind threads to physical processing units. A popular setting is export KMP_AFFINITY=SCATTER, which scatters across cores before using multiple threads on a given core. Another is COMPACT, which is usually not wanted since it favors using all threads on a core before using other cores. See the "Thread Affinity Interface" section of the compiler documentation to see the numerous controls available include exact enumeration of locations
KMP_STACKSIZE and MIC_STACKSIZE	Sets the number of bytes to allocate for each OpenMP thread to use as its private stack. Recommended size is 16 MB on any processor, including Knights landing, but the default is far smaller. The default on a Knights Landing coprocessor is a more reasonable 12 MB. This is often an important option to use. (MIC_STACKSIZE is for offload – see Chapter 18)
KMP_BLOCKTIME	Controls the amount of time that threads spend spinning at the end of an OpenMP region prior to going to sleep. 200ms is the default, a value of zero causing threads to sleep immediately and free up core resources, and infinite causes threads to spin forever, ensuring they are awake when the next parallel region is encountered. If we expect to encounter many parallel regions in quick succession, setting the blocking time to infinite can be very beneficial
OFFLOAD_REPORT	Enables printing diagnostics that show offload execution time, in seconds, and the amount of data transferred in bytes. Values should be 1 for a a small amount and 2 for more verbosity. (For offload – see Chapter 18)
OMP_NESTED KMP_HOT_TEAMS_MODE KMP_HOT_TEAMS_MAX_LEVEL	Three key environment variables for efficient use of nesting with OpenMP. OMP_NESTED enables nesting, the *hot teams* controls allow removal of some overhead from creating/destroy threads
OMP_PROC_BIND OMP_PLACES	OpenMP 4.x provides these new environment variables for handling the physical placement of threads, and these are compatible with nested parallel regions. To place team leaders on separate cores, and team members on the same core, set OMP_PROC_BIND=spread,close and OMP_PLACES=threads

FIG. 8.3

OpenMP controls.

regions are encountered often, such as when the threads are spawned for an innermost loop.

Avoiding the repeated overhead of creating and destroying the threads can lead to efficient nesting, which in turn can improve application performance. Therefore, there is support in the Intel OpenMP runtime (since Intel Compilers version 15, update 1, in 2014) known as "'hot teams"' that is able to reduce these overheads, by keeping a pool of threads alive (but idle) during the execution of the nonnested parallel code. The use of hot teams is controlled by two environment variables: `KMP_HOT_TEAMS_MODE` and `KMP_HOT_TEAMS_MAX_LEVEL`. To keep unused team members alive when team sizes change, we set `KMP_HOT_TEAMS_MODE=1`, and because we have two levels of parallelism, we set `KMP_HOT_TEAMS_MAX_LEVEL=2`.

Use of hot teams for nesting OpenMP solves a critical problem: nesting is important for effective parallelism for the large number of cores in processors today. Using hot teams for high core count nodes, including Knights Landing, is very useful for obtaining high performance.

The "hot teams" extensions are invoked purely from environment variables, code is not changed in a nonportable way. With use of the hot teams, nested parallelism is rewarded. We expect such solutions will make it into more OpenMP implementations in the future.

FORTRAN 2008

Fortran 2008 introduces two important features to support parallelism in the FORTRAN language: Coarrays and `DO CONCURRENT`.

We will focus on `DO CONCURRENT`; Coarrays, which offer a Partitioned Global Address Space (PGAS) model of programming, are covered in Chapter 16 (PGAS Programming Models).

When choosing `DO CONCURRENT`, it is important to note that the Intel compiler will not produce parallel code from `do concurrent` unless `-parallel` (`/Qparallel` on Windows) compile options are used. This is consistent with the Intel compiler avoiding creating concurrent code unless the compile line authorizes it. This helps avoid surprises when doing a simple recompile using an old Makefile. When parallelization is enabled, the Intel compiler maps "do concurrent" onto OpenMP automatically thereby making the APIs and environment variable controls of OpenMP apply equally to the use of "do concurrent."

DO CONCURRENT

The `DO CONCURRENT` construct specifies that there are no data dependencies between the iterations of a DO loop and it looks like this:

```
REAL :: SUM
   DO CONCURRENT (I = 1:N)
      SUM = BX(I) + CX(I)
      AX(I) = SUM + COS(SUM) + 3
   END DO
```

DO CONCURRENT AND DATA RACES

A confusing part of the standard is this: the use of `SUM` in this example is required to be defined before it is used in each iteration. This prevents a dependence on the ordering of the execution of iterations. However, the use of a single `SUM` in every iteration creates a conflict (a data race) if iterations are executed in parallel. The Fortran standard does not mandate that a compiler do anything to prevent the data race. The use of `SUM` in every iteration creates a data race if this loop is used concurrently (in parallel), so this is not advisable for a parallel loop. It is technically possible for a compiler to privatize the variable, since this would seem to be the intent, but even if one compiler did that, others may not. The "fix" for this is to use the `BLOCK` construct to explicitly create variables local to each iteration. The code example would change to:

```
DO CONCURRENT (I = 1:N)
   BLOCK
      REAL :: SUM
      SUM = BX(I) + CX(I)
      AX(I) = SUM + COS(SUM) + 3
   END BLOCK
END DO
```

DO CONCURRENT DEFINITION

The formal definition of `DO CONCURRENT` is shown in Fig. 8.4. The `DO CONCURRENT` range is executed for every active combination of the *index-name* values.

Each execution of the range is an iteration. The executions are free to occur in any order, so we need to be sure this is what we intend. A consequence of not knowing the

Part	Definition
type	An integer data type
forall-spec	*Index-variable-name* = *forall-limit* : *forall-limit* [: *forall-step*]
forall-limit	A scalar integer expression
forall-step	A scalar integer expression
mask-expr	A mask expression that is scalar and of type logical

FIG. 8.4

`DO [,] CONCURRENT ( [ type :: ] forall-spec [, mask-expr] ).`

order is that any variable modified in more than one iteration does not have a guaranteed outcome based on the iteration count. Branching is allowed within a given iteration but is not allowed to branch outside the DO CONCURRENT. Therefore branching cannot be used to terminate a DO CONCURRENT.

If *type* appears, the *index-name* has the specified type and type parameters. Otherwise, it has the type and type parameters that it would have if it were the name of a variable in the innermost executable construct or scoping unit.

If *type* does not appear, the *index-name* must not be the same as a local identifier, an accessible global identifier, or an identifier of an outer construct entity, except for a common block name or a scalar variable name.

The *index-name* of a contained FORALL or DO CONCURRENT construct must not be the same as an *index-name* of any of its containing FORALL or DO CONCURRENT constructs.

The following cannot appear in a DO CONCURRENT construct:

- A RETURN statement
- An image control statement
- A reference to a nonpure procedure
- A reference to module IEEE_EXCEPTIONS procedure IEEE_GET_FLAG, IEEE_SET_HALTING_MODE, or IEEE_GET_HALTING_MODE
- An EXIT statement must not appear within a DO CONCURRENT construct if it belongs to that construct or an outer construct.

The following are additional rules for DO CONCURRENT constructs:

- A variable that is referenced in an iteration must be previously defined during that iteration, or it must not be defined or become undefined during any other iteration.
- A variable that is defined or becomes undefined by more than one iteration becomes undefined when the loop terminates.
- An allocatable object that is allocated in more than one iteration must be subsequently deallocated during the same iteration in which it was allocated.
- An object that is allocated or deallocated in only one iteration must not be referenced, allocated, deallocated, defined, or become undefined in a different iteration.
- A pointer that is referenced in an iteration must have been pointer associated previously during that iteration, or it must not have its pointer association changed during any iteration.
- A pointer that has its pointer association changed in more than one iteration has an association status of undefined when the construct terminates.
- An input/output statement must not write data to a file record or position in one iteration and read from the same record or position in a different iteration.
- Records written by output statements in the range of the loop to a sequential-access file appear in the file in an indeterminate order.

The restrictions on referencing variables defined in an iteration of a DO CONCURRENT construct also apply to any procedure invoked within the loop.

These restrictions ensure no interdependencies occur that might affect code optimizations.

DO CONCURRENT VERSUS FOR ALL

DO CONCURRENT is sometimes referred to as "FORALL done right." FORALL is actually defined to be a series of array assignments that semantically must be executed one after the other. Unfortunately, it feels more natural to think of FORALL as a "parallel DO," which it is not, and it proved to be difficult to parallelize effectively. DO CONCURRENT is much simpler and specifies that each instance of the DO body can be executed in any order. The Intel compiler will try to parallelize DO CONCURRENT when the compiler "parallel" option is specified.

The statements within the DO CONCURRENT are executed in order, in each iteration, but there is no dependence on other iterations. FORALL could have array assignments only, but each assignment needed to be completed by all iterations before the next one executed, effectively creating a "wait for all" after each assignment. The initial idea of FORALL was to help create parallelization, but like a lot of High-Performance Fortran, the obsolete variant from which FORALL comes, it was not well thought through and creating parallelism was much more difficult than it seemed it would be at the time to many.

DO CONCURRENT VERSUS OpenMP "PARALLEL"

DO CONCURRENT is roughly equivalent to OMP PARALLEL used with a DO statement. The biggest difference in practice seems to be the lack of an explicit way to create private copies of variables within the loop, when using DO CONCURRENT, unless your Fortran compiler also supports the BLOCK construct from the Fortran 2008 standard. Since Fortran 2008 remains a standard with very uneven support in the industry, the use of OpenMP parallel instead is likely to be more portable in the short term. DO CONCURRENT offers a Fortran-specific language method to declare concurrency. Intel compilers make these methods highly compatible and consistently controlled by mapping "do concurrent" on top of OpenMP. This offers a consistent method to control affinity, number of threads used, and so on.

INTEL TBB

Intel® TBB is a widely used and highly portable template library that provides a comprehensive set of solutions to program using tasks in C++. It also provides a set of supporting functionality that can be used with or without the tasking infrastructure, such as concurrency-safe STL-compatible data structures, memory allocation, and portable atomics. We focus on tasks in this book due to their increased machine independence, safety, and scalability over threads. However, TBB also implements a significant subset of C++11 thread support, including platform-independent mutexes

and condition variables. See *For More Information* at the end of this chapter for more resources to learn more about TBB.

TBB is a collection of components that extends C++ for parallel programming. At the heart of TBB is a task scheduler that is most often used indirectly via the parallel algorithms in TBB, such as `tbb::parallel_for`. The rest of TBB provides thread-aware memory allocation, portable synchronization primitives, scalable containers (thread safe versions of key STL containers), and a variety of useful utilities. Each part is important for parallelism. Even the nontasking features are intended for use with other parallelism frameworks like OpenMP so that those frameworks do not have to duplicate key functionality.

WHY TBB?

Some key attributes of TBB are:

- TBB is designed to provide comprehensive support for C++ developers in one package.
- TBB is designed to work without any compiler changes, and thus be quickly portable to new platforms. As a result, TBB has been ported to a multitude of key operating systems and processors, and code written with TBB can likewise be easily ported.
- As a consequence of avoiding any need for compiler support, TBB does not have direct support for vector parallelism. However, TBB combined with #pragma omp simd from OpenMP or auto-vectorization can be an effective tool for exploiting both thread and vector parallelism.
- TBB is intended to provide low-level services such as memory allocation and atomic operations that can be used by programs with or without the parallel algorithm support in TBB.
- TBB is an active open source project. The project has enjoyed contributions from around the world. It is widely adopted and often cited in articles about parallelism in C++. It continues to grow as the parallel ecosystem evolves.

TBB was first available as a commercial library from Intel in the summer of 2006, not long after Intel shipped its first dual-core processors. It became an open source project in 2007 and has grown enormously in popularity to be the most used C++ abstraction for parallelism. TBB has been GPL v2 for nearly a decade but, in 2016, was moved to a nonviral open source license in response to numerous community requests.

TBB provides a much-needed comprehensive answer to the question "what must be fixed or added to C++ for parallel programming." TBB's key programming abstractions for parallelism focused on logical specification of parallelism via algorithm templates. By also including a taskstealing scheduler, a thread-aware memory allocator, portable mutexes, global timestamps, and concurrent containers, TBB provided what was needed to program for parallelism in C++. The first release was primarily focused on strict fork/join or loop-type data parallelism.

Through the involvement of customers and community, TBB has grown to be the most feature-rich and comprehensive solution for parallel application development available today. It has also become the most popular!

The TBB project was grown through a steady addition of ports to a wide variety of machines and operating systems, and the addition of numerous new features that have added to the applicability and power of TBB.

TBB 4.0 added a powerful capability for expressing parallelism as data flowing through a graph. Support for Intel Xeon Phi coprocessors started in 2010. Use of TBB continues to grow, and the open source project enjoys serious support from Intel and others.

USING TBB

Include the header `<tbb/tbb.h>` to use TBB in a source file. All public identifiers are in namespace `tbb` or `tbb::flow`. In the following descriptions, the phrase "in parallel" indicates that parallelism is permitted if resources allow but is not mandated. The license to ignore unnecessary parallelism enables the TBB task scheduler to use parallelism efficiently.

PARALLEL_FOR

The function template `parallel_for` maps a functor across arange of values. The template takes several forms. The simplest is:

```
tbb::parallel_for(first,last,f )
```

where *f* is a functor. It evaluates the expression *f* (*i*) in parallel for all *i* in the half-open interval (*first*,*last*), Both *first* and *last* must be of the same integral type. It is a parallel equivalent of:

```
for (auto i=first; i<last; ++i) f(i);
```

A slight variation specifies a stride:

```
tbb::parallel_for(first,last,stride,f )
```

It is like the previous version, except that the possible values of *i* step by *stride*, starting with *first*. This form is a parallel equivalent of:

```
for (auto i=first; i<last; i+=stride ) f(i);
```

Another form of `parallel_for` takes two arguments:

```
tbb::parallel_for(range,f )
```

It decomposes *range* into subranges and applies *f* to each subrange, in parallel. Hence the programmer has the opportunity to optimize *f* to operate on an entire subrange instead of a single index. This version in effect exposes the tiled implementation of the map pattern used by TBB.

This form of parallel for also generalizes the parallel map pattern beyond one-dimensional ranges. The argument *range* can be any *recursively splittable range* type.

BLOCKED_RANGE

The most commonly used recursive range is `tbb::blocked_range`. It is typically used with integral types or random-access iterator types. For example, `blocked_range<int>(0,8)` represents the index range {0, 1, 2, 3, 4, 5, 6, 7}. An optional third argument called the *grainsize* specifies the maximum size for splitting. It defaults to 1. For example, the following snippet splits a range of size 30 with grainsize 20 into two indivisible subranges of size 15:

```
//Construct half – open interval [0,30) with grainsize of 20
blocked_range<int> r(0,30,20); assert(r.is_divisible());

// Call splitting constructor
blocked_range<int> s(r);

// Now r=[0,15) and s=[15,30) and both have a grainsize 20
// inherited from the original value of r . assert(!r.is_divisible());
assert(!s.is_divisible());
```

There is a two-dimensional variant called `tbb::blocked_range2d`. It permits the use of a single `parallel_for` to iterate over two dimensions at once, which sometimes yields better cache behavior than nesting two one-dimensional instances of `parallel_for`.

PARTITIONERS

The range form of `parallel_for` takes an optional *partitioner* argument, which lets the programmer specify performance-related tactics. The argument can have one of three types:

- *auto partitioner*. The runtime will try to subdivide the range sufficiently to balance load, but no further. This behavior is the same as when no partitioner is specified.
- *simple partitioner*. The runtime must subdivide the range into subranges as finely as possible; that is method `is_divisible` will be false for the final subranges.
- *affinity partitioner*. Request that the assignment of subranges to underlying threads be similar to a previous invocation of `parallel_for` or `parallel_reduce` with the same `affinity_partitioner` object.

These partitioners also work with `parallel_reduce`. An invocation of `parallel_for` with a `simple_partitioner` looks like:

```
parallel_for(r,f ,simple_partitioner());
```

The simple_partitioner is useful in two scenarios:

- The functor *f* uses a fixed amount of memory for temporary storage and hence cannot deal with subranges of arbitrary size. For example, if *r* is a

`blocked_range`, the partitioner guarantees that *f* is invoked on subranges not exceeding the grainsize of *r*.

- The work for *f* (*r*) is highly unbalanced in a way that fools the `auto_partitioner` heuristic into not dividing work finely enough to balance load.

The ability to encourage affinity with TBB is a critical feature for performance, particularly with codes that do not benefit much from the dynamic scheduling.

An `affinity_partitioner` can be used for cache fusion. Unlike the other two partitioners, it carries state. The state holds information for replaying the assignment of subranges to threads. Fig. 8.5 shows an example of its use in a common pattern—serially iterating a map. In the listing, variable *ap* enables cache fusion of each map to the next map. Because it is carrying information between serial iterations, it must be declared outside the serial loop. TBB uses the variable *ap* to remember which threads

```
1    void relax(
2       double* a,   //  pointer to array of data
3       double* b,  //  pointer to temporary storage
4       size_t n,       //  number of data elements
5       int iterations // number of serial iterations
6    ) {
7       assert(iterations%2==0);
8       //   Declare partitioner prior to loop.
9       tbb::affinity_partitioner  ap;
10      //   Serial  loop around a  parallel  loop .
11      for( size_t t=0; t<iterations;  ++t ) {
12        tbb::parallel_for(
13          tbb::blocked_range<size_t>(1,n-1),
14            [=]( tbb::blocked_range<size_t>  r ) {
15              size_t  e =  r.end();
16     #pragma omp simd
17              for( size_t i=r.begin();  i<e; ++i )
18                b[i] =  (a[i-1]+a[i]+a[i+1])*(1/3.0);
19            },
20            ap);
21        std::swap(a,b);
22      }
23    }
```

FIG. 8.5

Example of `affinity_partitioner`.

ran which subranges of the previous invocation of `parallel_for` and biases execution toward replaying that assignment.

PARALLEL_REDUCE

Function template `parallel_reduce` performs a reduction over a recursive range. It has several forms.

PARALLEL_INVOKE

Template function `parallel_invoke` evaluates a fixed set of functors in parallel. For example:

```
tbb::parallel_invoke(f,g,h);
```

evaluates the expressions `f()`, `g()`, and `h()` in parallel and waits until they all complete. Anywhere from 2 to 10 functors are currently supported. The `task_group` class allows an arbitrary number of functors to be run in parallel.

TBB FLOW GRAPH

A fantastic way to avoid the evils of barrier synchronization is to program in a flow graph style, essentially describing dependencies as data flow instead of regions of code separated by barriers.

The Intel TBB library is well known for its support of loop parallelism; the flow graph interface extends the capabilities of TBB to allow fast, efficient implementations of dependency graph and data flow algorithms, enabling developers to exploit parallelism at higher levels in their application. The flow graph interface was introduced in Intel TBB 4.0 in 2011.

There is an excellent tutorial online for the TBB flow graph. See *For More Information* at the conclusion of this chapter for a link.

TBB MEMORY ALLOCATION, MEMKIND, AND MCDRAM

The memory allocator in TBB has been very popular. The other popular memory allocator is jemalloc. The initial version of memkind (see Chapter 3) is built upon jemalloc from which it grew. It is expected that memkind will eventually expand to offer the ability to interact with other memory allocators including TBB memory allocators. In the mean time, using both memory allocators is not necessarily a problem.

hStreams

hStreams is a library focused on making it easier to express and harvest task concurrency, and to manage memory on heterogeneous platforms. While hStreams is relatively new, it represents a growing diversity of solutions seeking to move away

from barrier-synchronous programming to favor data flow. The flow graph in TBB has been an available standard that we have recommended where it is suitable. Two differentiating features of hStreams that make it a compelling alternative to TBB flow graph are its C language interfaces, and its ability to infer dependence relationships among tasks rather than needing to explicitly express them.

hStreams emphasizes a "flow" style programming model built around sending both data and control in "streams" that flow through the system. hStreams was introduced in the second volume of the Pearls books. hStreams can be used either in an offload context, over PCIe or fabric, or in a standalone context, within a single processor (including Knights Landing). It is used to achieve concurrency by invoking tasks on multiple nodes (e.g., Intel Xeon processors, Knights Landing), on different partitions within a single node (e.g., each of four quadrants with SNC-4 cluster mode), and by overlapping computation and communication across the fabric, to I/O or to memory.

The hStreams library also makes memory management easier. Users do not need to become experts in the various memory interfaces that are available; they just specify the kind of memory, and let the runtime implement the underlying mechanism. Users can define memory buffers, which can be selectively instantiated in any subset of the nodes and partitions mentioned earlier. The kind of memory used for the buffer instances can be selected, for example, normal DDR, high-bandwidth MCDRAM, or nonvolatile. The allocation policy can also be controlled: the kind can be expressed as either a best-effort preference or a hard requirement that causes an allocation failure if that kind is not available. Finally, the affinity may also be controlled. Under load, the performance difference between accessing memory in a local memory controller and one from a different sub-NUMA clustering quadrant can be as high as 10%. hStreams does the hard work of assuring that the first touch of the physical memory happens from a thread affinitized to the desired quadrant, even if it needs to create a software thread to do so.

SUMMARY

Knights Landing has many cores, which in turn offer a high ability for parallel execution. Feeding them is important in order to reach the potential of Knights Landing. The rich environment of options available for programming in a consistent fashion offers the ability to program in an effective style that spans all processors, including Intel® Xeon® processors and Intel® Xeon Phi™ processors. Our recommendation is to utilize OpenMP, Fortran 2008, TBB, or MKL, but other viable options including pthreads exist and newer options such as hStreams are beginning to emerge. Even with new emerging entries, we expect OpenMP, Fortran 2008, TBB, and MKL to remain the best options for effective scaling in terms of giving both short-term benefits and holding up well over time.

There is a genuine need to abandon threads in programming in favor of tasks, "The Problem with Threads" is recommended reading. Think of it as a modern version of the classic "Go To Statement Considered Harmful," which is so commonly

accepted today that it is hard to recall the controversy it raised for more than a decade after its publication. Programming with threads has been rapidly fading away as did "Go To." These are the key papers that explain the need for change.

With the foundations of tasking to exploit the many cores in Knights Landing, the next four chapters discuss applying techniques for vectorization (Chapters 9-12). In Chapter 13, we discuss applying MKL. In Chapter 15, we discuss how MPI can combine with all the techniques from the prior chapters.

FOR MORE INFORMATION

Here are some additional reading materials we recommend on various threading models.

- Intel online product documentation: http://tinyurl.com/inteldocs.
- Intel Threading Building Blocks: http://opentbb.org.
- Intel Threading Building Blocks Flow Graph Tutorial, http://lotsofcores.com/tbbFG.
- Fortran 2008: http://fortranwiki.org.
- OpenMP: http://openmp.org.
- Thread Affinity Interface in the Intel Compilers documentation, search "Intel Thread Affinity Interface" for the latest online documentation.
- Supported Environment Variables in the Intel Compilers documentation, search "Intel Supported Environment Variables" for the latest online documentation.
- James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.
- "hStreams" information:
  - Chapter in Pearls volume 2: Gaurav Bansal, Chris J. Newburn and Paul Besl, Chapter 15—Fast Matrix Computations on Heterogeneous Streams, in High-Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, 2015, ISBN 978-0-12-803819-2.
  - http://lotsofcores.com/hstreams.
  - http://01.org/hetero-streams-library.
- Michael McCool, Arch Robison, James Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.
- Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.
- Edward A. Lee. *The problem with threads*. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, Jan. 2006. A published version of this paper is in IEEE Computer 39(5):33–42, May 2006.
- Edsger Dijkstra (Mar. 1968). "Go To Statement Considered Harmful" (PDF). Communications of the ACM 11 (3):147–148.

CHAPTER

Vectorization

9

What is new with Knights Landing in this chapter?
AVX-512

One way or another, using the Advanced Vector eXtensions 512-bit (AVX-512) operations, is critical to the performance of most applications that run on Knights Landing. Fortunately, there are many ways in which an application may use AVX-512, including libraries (Chapter 13), compilation with or without hints in the source code (covered in this chapter), or hand coded use of AVX-512 intrinsics (Chapter 12). These methods can be freely mixed within an application as there is no need to use one method exclusively. Data layout is critical for effective use of vector operations, so we discuss data layout in this chapter and present a template library solution in Chapter 11.

Two indispensable tools for helping achieve vectorization are the optimization reports produced by the Intel® compiler (-qopt-report=n) and the Vector Advisor (Chapter 10). The optimization reports give valuable information about an application and vectorization that the compiler is successfully able to generate; the Vector Advisor analyzes an application and gives specific advice on what to do to achieve more vectorization.

Knights Landing is designed with strong support for vector level parallelism with features such as the Advanced Vector eXtensions 512-bit (AVX-512) vector operations, hardware prefetching, software prefetch instructions, caches, and high aggregate memory bandwidth. This rich support in hardware coupled with numerous software options may seem overwhelming at first glance, but the richness available offers many solutions for diverse needs. Arranging algorithms to present opportunities to use vector instructions is the critical, but even then, the layout of data, alignment of data, and the effectiveness of prefetching into caches can affect vectorization. This chapter covers the rich set of options available for processors including Knights Landing without requiring detailed knowledge of AVX-512.

The best software options are using functions in the Intel® Math Kernel Library (MKL), or programming using the vector support from any of the SIMD directives/pragmas and prefetching. Alternatives such as using intrinsics and compiler autovectorization can deliver excellent performance results, but the limitations in portability and future maintenance can be substantial.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00009-0

In this chapter, we start by explaining why vectorization matters, and then we give an overview of three approaches to vectorization. We introduce a six-step methodology for looking for important and easy places where a few changes might improve vectorization. The rest of the chapter then walks through the key topics for writing vectorizable code:

- Streaming through caches: data layout, alignment, prefetching, streaming stores
- Compiler tips, options, and directives/pragmas
- Assembly code inspection
- Numerical result variations

WHY VECTORIZE?

Performance! Full use of the vector instructions (AVX-512) means being able to do 16 single-precision or 8 double-precision mathematical operations at once instead of just a single operation with a nonvector instruction. The performance boost from using vector instructions is substantial and is key to the performance we should expect from a many-core Intel Xeon Phi product such as Knights Corner (1st gen coprocessor) or Knights Landing. Until a couple of years ago, with SSE, processors could only offer four single-precision or two double-precision operations per instruction. The advent of AVX has made that eight and four, respectively, but still only half of what is found in Knights Landing.

HOW TO VECTORIZE

Effective vectorization comes from a combination of efficient data movement and recognition of vectorization opportunities in the program itself. Many things get in the way of vectorization: poor data layout, C/C++ language limitations, required compiler conservatism, and poor program structure. We will start with the three approaches to vectorization that should be of most interest, look at a six-step method that may help you, and finally, we will review the multiple obstacles and solutions such as data layout and compiler options, in order to vectorize your code! Fig. 9.1 summarizes techniques for achieving vectorization.

THREE APPROACHES TO ACHIEVING VECTORIZATION

Vectorization occurs when code makes use of vector instructions effectively. We encourage finding a method that has the compiler to generate the instructions as opposed to manually writing in assembly code or with explicit intrinsics. The compiler can do everything automatically for a very small number of examples but usually does much better with some help from programmers to overcome limitations in

	Libraries	SIMD directives	Auto-vectorization
Languages supported	Fortran, C, C++	Fortran, C, C++	Fortran, C, C++
First available	More than 30 years ago	2010	More than 30 years ago
Key benefit	Someone else did all the hard work! Reliable and predictable, no code changes to scale forward	Full programmer control, reliable and predictable. Now standard in OpenMP 4.0	Few code changes needed, but expect to babysit the compiler with options
Key disadvantage	Specific libraries only, not a completely general purpose solution	SIMD directives are powerful but need to be used carefully to avoid changing program meaning	Unreliable, limited by language and compiler technology, not reliably portable

FIG. 9.1

Techniques to achieve vectorization.

the programming language or algorithm implementation. As listed in Table 5.1, three approaches to consider are:

- *Libraries*. See Chapter 13. The Intel performance libraries have been tuned to use vectorization when possible within the library routines.
- *Auto vectorization*. Count on the compiler to figure it all out. This works for only the simplest loops, and usually only with a little help overcoming programming language limitations (for instance, using the `-ansi-alias` compiler option). Fig. 9.2 shows the C code for a simple add of two vectors. We have inserted an alignment directive, which is important. There are additional tips for compiler vectorization in an upcoming section, *General Compiler Tips and Comments for Vectorization*, in this chapter.
- *SIMD directives/pragmas to assist or require vectorization*. See Fig. 9.3. The SIMD directives (added to OpenMP 4.0 in 2013) give simple and effective controls to mandate vectorization. These are much more powerful than "ivdep" directives that did not mandate vectorization. The catch in using these directives is that the vectorization of arbitrary loops is generally unsafe or will change the meaning of the program. Use of these directives has been very popular but requires a firm understanding of their meaning and their dangers. SIMD directives are covered in an upcoming section, *SIMD Directives*, in this chapter.

```
__declspec(align(16)) float a[MAX], b[MAX], c[MAX];
for (i=0;i<MAX;i++)
   c[i]=a[i]+b[i];
```

FIG. 9.2

Vector addition using standard C.

```
!dir$ assume_aligned a:64,b:64,c:64
!$omp simd
   do i = 1,imax
      c(i) = a(i) + b(i)
   enddo
```

FIG. 9.3

Vector addition using standard Fortran.

SIX-STEP VECTORIZATION METHODOLOGY

Intel published an interesting article suggesting a six-step process for vectorizing an application. It is not specific to Knights Landing, but rather a general methodology designed for Intel Xeon processors that is equally appropriate for Knights Landing. It is neatly documented in an online *Vectorization Toolkit* that includes links to additional resources for each step. The URL is given in *For More Information* at the end of this chapter. This section is a brief overview suitable to give a taste of some interesting tools Intel has to help evaluate and guide vectorization work.

The six-step vectorization methodology approach is very useful in cases where incremental work can yield strong results. Scientific codes that have successfully used vector supercomputers previously, or made use of SIMD instructions such as SSE or AVX, are easily the best candidates for this approach. This six-step methodology is no panacea because it can completely miss a bigger picture of overall algorithm redesign when that might be most appropriate. Parallelism may not be easily exposed. An effective parallel program may require a more holistic approach involving restructuring (refactoring) of the algorithm and data layout to get significant gains. Whether parallelism is easily exposed, or with great difficulty, it is wise to start by looking for the easiest opportunities to expose. These six steps help explain how to look for the most accessible opportunities.

STEP 1. MEASURE BASELINE RELEASE BUILD PERFORMANCE

We should start with a baseline for performance, so we know if changes to introduce vectorization are effective. In addition, setting a goal can be useful at this point, so we can celebrate later when we achieve it. A release build should be used instead of a debug build. A release build will contain all the optimizations for our final application and may alter the hotspots or even the code that is executed. For instance, a release build may optimize away a loop in a hotspot that otherwise would be a candidate for vectorization. Using a debug build would be a mistake when working to optimize. A release build is the default in the Intel Compiler. We have to specifically turn off optimizations by doing a DEBUG build on Windows (or using the `-Zi`

switch) or using the `-Od` switch on Linux or OS X. If using the Intel Compiler, ensure you are using optimization levels 2 or 3 (-O2 or -O3) to enable the compiler to auto-vectorize.

STEP 2. DETERMINE HOTSPOTS USING INTEL VTune™ AMPLIFIER

We can use Intel® VTune™ Amplifier or Intel Advisor performance profilers, to find the most time-consuming functions in an application. The Hotspots Analysis and Advisor Surveys are recommended, although Lightweight Hotspots would work as well (it will profile the whole system as opposed to just your application).

Identifying which areas of the code are taking the most time will allow us to focus on optimization efforts in the areas where performance improvements will have the most effect. Generally, we want to focus on only the top few hotspots, or functions taking at least 10% of the application's total time. Make note of the hotspots we want to focus on for the next step.

STEP 3. DETERMINE LOOP CANDIDATES USING INTEL COMPILER VEC-REPORT

The optimization report produced by the Intel compiler is an invaluable tool for performance tuning and can be generated by using the compiler argument "-qopt-report=*n*" with *n* specifying verbosity. Chapter 23 discusses use of the report while optimizing an application (*n*-body).

The vectorization section of the optimization report can tell you whether or not each loop in your code was vectorized. Look at the output of the optimization report for the hotspots you determined in Step 2. If there are loops in your hotspots that did not vectorize, check whether they have math, data processing, or string calculations on data in parallel (for instance in an array). If they do, they might benefit from vectorization. Move to step 4 if any candidates are found.

To generate information about compiler vectorization from the optimization report, compile with the options `"-qopt-report -qop-report-phase=loop,vec"` on Linux or `"/Qopt-report/Qopt-report-phase:loop,vec"` on Windows.

STEP 4. GET ADVICE USING INTEL ADVISOR

Use the vectorization analysis capability of Intel Advisor to analyze your application's run time behavior and identify the components of the application that will benefit most from vectorization. The capabilities of the Vectorization Advisor, including the integration of compiler vec-reports, are covered in the next chapter (Chapter 10).

STEP 5. IMPLEMENT VECTORIZATION RECOMMENDATIONS

Make sure a recommended change does not affect the semantics or safety of your loop calculations. One way to ensure that the loop has no dependencies that may be affected is to see if executing the loop in backwards order would change the results. Another is to think about the calculations in your loop being in a scrambled order. If the results would be changed, your loop has dependencies and vectorization would not be safe. You may still be able to vectorize by eliminating dependencies in the loop. Modify your source code to give additional information to the compiler or optimize your loop for better vectorization.

STEP 6: REPEAT!

Iterate through the process as needed until performance is achieved or there are no good candidates left in the identified hotspots. Please bear in mind that this six-step process does not consider opportunities that may be possible if algorithm and data restructuring are considered (see Chapter 7).

STREAMING THROUGH CACHES: DATA LAYOUT, ALIGNMENT, PREFETCHING, AND SO ON

In order for vectorization to occur efficiently, data needs to flow to and from the vector instructions without excessive overhead. Efficiency in data movement depends on data layout, alignment, prefetching, and efficient store operations. Some investigation into the code, in order to look for inefficiencies, produced by the compiler can be done by anyone and is discussed in a later section in this chapter titled "Looking at What the Compiler Created."

WHY DATA LAYOUT AFFECTS VECTORIZATION PERFORMANCE

Vector parallelism comes from performing the same operation on multiple pairs (or triplets with FMA) of data elements simultaneously. Fig. 9.4 illustrates the concept using the AVX-512 vector registers known as ZMM registers. Fig. 9.4 is not particular to AVX-512, other than the exact width of the registers. In fact, the optimizations discussed in the chapter are essentially the same as we would do for any modern microprocessors. AVX-512 does offer more vector level parallelism by having wider registers and more aggregate memory bandwidth, but with the correct programming methods an application can be written to map to processors through use of Intel MKL or the compiler without requiring different approaches in programming.

Fig. 9.4 illustrates a vector addition of two registers, each holding 16 single-precision floating-point values yielding 16 single-precision sums. This math is done, for example, by loading the two input registers from data in memory and then

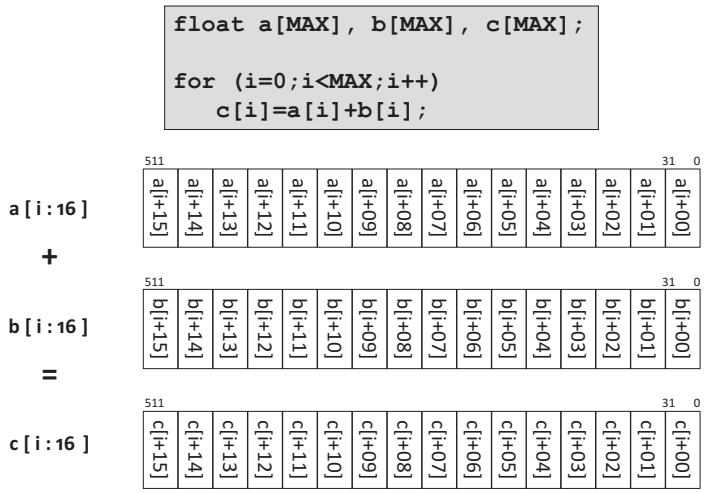


FIG. 9.4

Vector addition with AVX-512 (equation on left uses [start:length] notation).

performing the addition operation. There are several issues that need to be understood for optimal use:

- *Data layout in memory, aligned and packed*. Sixteen values have to be collected into each input register. A simple vector load from an aligned memory location will give optimal performance for that load. If not, the first consideration should be whether data could be laid out in order and aligned. Data that is not packed and aligned optimally will require more instructions and cache or memory accesses to collect and organize in registers in order to use the vector operations. As discussed sections regarding use of gather versus load/permute, and scatter versus permute/store in Chapter 6, optimal instruction sequences should be utilized—a task we hope we can entrust to the compiler but we may choose to control manually with intrinsics (Chapter 12).
- The extra instructions and data accesses reduce performance. This is especially true if gather/scatter instructions are issued for a particular data layout but could be avoided by a different data layout. The compilers offer alignment directives and command line options, covered later in this chapter, to force or specify alignment of variables. Chapter 11 discusses one option to address data layout issues for C++ programs.
- *Data locality*:
  - *Fetch data from cache not memory*. Data eventually comes from memory, but at the time a vector load instruction is issued it is much faster if the data has been fetched into the closest (level 1, known as L1 cache) prior to the load

instruction. On Knights Landing, data actually travels from a memory (MCDRAM or DDR) to L2 cache to L1 cache. Optimally this can be done by a prefetch of data from memory to L2, later followed by a prefetch from L2 to L1 cache, and later by the load instruction from L1 cache. The prefetches can be initiated by hardware or by software prefetches specified by the compiler automatically or manually by the programmer. L1 Prefetch instructions can ask for memory data to L1, but there are far fewer L1 prefetch operations allowed to be outstanding at the same time than the capacity for L2 prefetches to be outstanding. The compiler inserts prefetches automatically, and there are directives to help the compiler with hints and there are `mm_prefetch` intrinsics to do the prefetching manually.

Rearranging a program to increase temporal reuse is often a rewarding optimization.

- *Data reuse*. If a program is fetching data into the cache that will be accessed more than once, then those accesses should occur close together. This is called temporal locality of reference. Ensuring that data is reused quickly helps reduce the chance that data will be evicted before it is needed again—resulting in a new fetch. Rearranging a program to increase temporal reuse is often a rewarding optimization. Intel MKL (Chapter 13) uses blocking in the library routines, which is a big help. For code we write explicitly, the challenge of blocking generally falls to the programmer. The good news is that cache blocking has been studied for decades for processors, and Knights Landing relies on the same programming techniques. Before we decide to curse caches, it is worth noting that caches exist to lower power consumption and increase performance. The fact that our programming style can align with the cache designs to maximize performance is the price we pay to have more performance and power efficiency than would be possible without caches.
- *Streaming stores*. If a program is writing data out that will not be used again, and that data occupies memory linearly without gaps, then making sure streaming stores are being generated is important. Streaming stores write data directly to memory rather than storing in caches, this prevents needless evictions of data we are more likely to use in the near future.

DATA ALIGNMENT

If poor vectorization is detected via compiler vectorization report or assembler inspection, we may decide to improve data alignment to increase vectorization opportunities. The relative alignment of one array compared to another, or of one row of a matrix compared to the next, is often important. The compiler can often

compensate for an overall offset in absolute alignment. Though of course, absolute alignment of everything is one way of getting relative alignment. For matrices, we may want to pad the row length (or column length, for Fortran) to be a multiple of the alignment factor. As discussed in Chapter 6, alignment to 64-byte boundaries of all data (usually arrays) is not as important with Knights Landing as it was on Knights Corner but can help in our efforts to reach maximum performance. Aligned memory allocation can be achieved with calls to:

```
void* _mm_malloc (int size, int base)
```

or

```
int hbw_posix_memalign(void ** memptr,
                       size_t alignment,
                       size_t size);
```

or

```
int hbw_posix_memalign_psize(void ** memptr,
                 size_t alignment,
                 size_t size,
                 int pagesize);
```

The latter two (hbw) allocations are for MCDRAM as discussed in Chapter 3. Alignment assertion `align(base)` can be used on statically allocated variables:

```
/* array temp: 64byte aligned */
__attribute__((align(64))) float temp[SIZE];
```

or

```
/* array temp: 64byte aligned */
__declspec(align(64)) float temp[SIZE];
```

Note: The Intel compilers accept either `__attribute__()` or `__declspec()` spellings to contain the directives for alignment. The `__attribute__()` spelling is associated with gcc and Linux; the `__declspec()` spelling originated with Microsoft compilers and Windows. The Intel compiler for Linux accepts either spelling.

The equivalent Fortran directive is

```
!DIR$ ATTRIBUTES ALIGN: base :: variable
```

Alignment assertion `__assume_aligned(pointer, base)` can be used on pointers:

```
__assume_aligned(ptr, 64); /* ptr is 64byte aligned */
```

Fortran programmers would use the following to assert that the address of A(1) is 64-byte aligned:

```
!dir$ assume_aligned A(1): 64
```

We can use the ATTRIBUTES option ALIGN with FASTMEM for MCDRAM allocations (Chapter 3):

```
!DIR$ ATTRIBUTES FASTMEM, ALIGN:64 :: A
```

Additionally, there are compiler command line options (align) to specify alignment of data. The Fortran compiler supports variations to force alignment of arrays and `COMMON` blocks. C/C++ programs generally use only the `__attribute()` or `__declspec()` source code directives, whereas Fortran programmers often use the command line options as an alternative to the directives.

Some examples of alignment directives (shown in Fortran) are shown in Fig. 9.5

Directive/option	Descriptions
`!dir$ attributes align : 64 :: A, B`	Directs compiler to align the start of objects A, B to 64 bytes Does not work for objects inside common blocks or derived types with sequence attribute; can be used to align start of common block itself
`!dir$ assume_aligned : 64 :: A, B`	Tells the compiler that A and B have been aligned to 64-byte boundaries Useful if the compiler wouldn't otherwise know
`!dir$ vector aligned`	Invites the compiler to vectorize a loop using aligned loads for all arrays, ignoring efficiency heuristics, provided that it is safe to do so.
`-assume array64byte`	Roughly like `!dir$ attributes align:64` wherever possible

`!dir$ attributes align`
and
`!dir$ assume_aligned`
	Where arrays are declared, not where they are used
	Can appear together, but should not be necessary

`!dir$ assume_aligned`	Most obvious use is for an incoming subroutine argument Cannot be used for global objects declared elsewhere or sequential objects (risk of conflicts) such as within common blocks or within modules or within structures
`!dir$ vector aligned`	Applies to all the arrays in the loop that follows No impact on array attributes or other loops

FIG. 9.5

Direction/options for alignment, illustrated in Fortran.

Prefetching

The best performing applications will have well laid out data ready to be streamed to the math units to be processed. Moving the data from its tidy layout in memory to the math units is done using prefetching to avoid delays that occur if a mathematical operation has to wait for input data.

Prefetching is an important topic to consider regardless of what coding method we use to write an algorithm. Any time a load requests data not in the L1 cache, a delay occurs to fetch the data from an L2 cache. If data is not in any L2 cache, an even longer delay occurs to fetch data from memory. The lengths of these delays are non-trivial, and avoiding the delays can greatly enhance the performance of an application. Optimized libraries such as Intel MKL will already utilize prefetches, so we will focus on controlling prefetches for code we are writing.

The hardware prefetchers (discussed in Chapters 4 and 6) are very effective on Knights Landing and are very similar to the L2 prefetchers and L1 prefetchers on Intel Xeon processors.

Compiler prefetches

When using the Intel compiler with normal or elevated optimization levels (-O2 or greater), prefetching is automatically set to `opt-prefetch=3`. Compiler prefetching is enabled with the `-opt-prefetch=`*n* option, with *n* being a value 1–4 on Linux (Windows:`/Qopt-prefetch:`*n*). The higher the value, the more aggressive the compiler is with issuing prefetch instructions. If the algorithm is well blocked to fit in L2 cache, prefetching is less critical but generally still useful for Knights Landing.

In general, we recommend using the compiler to issue prefetches. If this is not perfect, there are a number of ways to give the compiler additional hints or refinements but still have the compiler generate the prefetches. We consider manual insertion of prefetch directives to be a last resort if compiler prefetching cannot be utilized for a particular need.

Compiler prefetch controls (prefetching via directives/pragmas)

If there is a need to manually inset directives, prefetch directive/pragma can be utilized in the source code. A prefetch directive alters default behavior in the compiler; the passing of addition information through directives can improve the performance of an application. The format for C/C++ and Fortran is, respectively:

```
#pragma prefetch var:hint:distance
!DIR$ prefetch var:hint:distance
```

The *hint* value can be 0 (L1 cache [and L2 due to inclusion]), 1 (L2 cache, not L1). A methodology for tuning prefetches is described in a chapter in a prior Pearls book, see *For More Information* at the end of this chapter. The methodology was to systematically vary prefetch distances, compile, run, and record the effects across a range of values to help automatically find better choices. The authors originally envisioned it to be useful only for in-order execution engines like those on Knights

Corner; however they were surprised to find that a systematic hunt can find better values even for out-of-order execution engines like those found on Knights Landing.

Manual prefetches

Figs. 9.6 and 9.7 show examples of using manual prefetch directives. We recommend only doing this if the compiler cannot generate prefetches sufficiently even with the hints covered in the prior section. These intrinsics work on both processors and coprocessors and are not specific to Knights Landing. The intrinsics for Fortran and C/C++, respectively, are:

```
MM_PREFETCH (address [, hint])
void _mm_prefetch(char const*address, int hint)
```

The intrinsic issues a software prefetch to load one cache line of data located at *address*. The value *hint* specifies the type of prefetch operation. Values of `FOR_K_PREFETCH_T0` (Fortran) or `_MM_HINT_T0 (C/C++)`, `for` L1 prefetches, and `FOR_K_PREFETCH_T1` (Fortran) or `_MM_HINT_T1 (C/C++)`, for L2 prefetches, are most common and are the interesting ones for use with Knights Landing.

If we do decide to insert prefetches manually, we will want to disable automatic compiler insertion of prefetch instructions. Too many prefetches lead to excessive

```
#include <stdio.h>
#include <immintrin.h>
#define N 1000
int main(int argc, char **argv) {
  int i, j, htab[N][2*N];
  for (i=0; i<N; i++) {
  #pragma noprefetch  // Turn off compiler prefetches for
this loop
    for (j=0; j<2*N; j++) {
      _mm_prefetch((const char *)&htab[i][j+20],
_MM_HINT_T1); // vprefetch1
      _mm_prefetch((const char *)&htab[i][j+2],
_MM_HINT_T0); // vprefetch0
      htab[i][j] = -1;
    }
  }
  printf("htab element is %d\n", htab[3][40]);  return 0;
}
/* constants to use with _mm_prefetch  (extracted from
*mmintrin.h) */
#define _MM_HINT_T0 1
#define _MM_HINT_T1 2
#define _MM_HINT_T2 3
#define _MM_HINT_NTA    0
#define _MM_HINT_ENTA   4
#define _MM_HINT_ET0    5
#define _MM_HINT_ET1    6
#define _MM_HINT_ET2    7
```

FIG. 9.6

C prefetch intrinsics example, manual prefetching.

```
subroutine spread_lf (a, b)
  PARAMETER (n = 1028)
  real*8 a(n,n), b(n,n), c(n)
  do j = 1,n
    do i = 1,n
      a(i, j) = b(i-1, j) + b(i+1, j)
      call mm_prefetch (a(i+2, j), 0)
      call mm_prefetch (a(i+20, j), 1)
      call mm_prefetch (b(i+21, j), 1)
    enddo
  enddo
  print *, a(2, 567)
  stop
end
```

FIG. 9.7

Fortran prefetch intrinsics example, manual prefetching.

memory traffic and cache conflicts that may reduce performance. If we are going through the effort to control prefetching manually, we should have already determined the compiler prefetching was not optimal. This generally happens for more complex data processing than the compiler can determine automatically.

Compiler prefetching is disabled with the `-opt-prefetch=0` or `-no-opt-prefetch` option on Linux (Windows:`/Qopt-prefetch:0` or `/Qopt-prefetch-`).

STREAMING STORES

Streaming stores are a special consideration in vectorization. Streaming stores are instructions especially designed for a continuous stream of output data that fills a section of memory with no gaps. They are supported by many processors, including Intel Xeon processors, as well as Knights Landing. An interesting property of an output stream is that the result in memory does not require knowledge of the prior memory content. This means that the original data does not need to be fetched from memory. This is the problem that streaming stores solve—the ability to output a data stream but not use memory bandwidth to read data needlessly.

Having the compiler generate streaming stores can improve performance by not having the processor or coprocessor fetch cache lines from memory that will be completely overwritten. This method stores data with instructions that use a nontemporal buffer, which minimizes memory hierarchy pollution. This optimization helps with memory bandwidth utilization and is present for all Intel processors (including Knights Landing) and may improve performance for both.

Use of the compiler option -opt-streaming-stores *keyword* (Linux and OS X) or/Qopt-streaming-stores:*keyword* (Windows) controls the compilers generation of streaming stores. The *keyword* can be *always*, *never* or *auto*. The default is *auto*. It is useful to know how to help the compiler understand an application enough to use streaming stores effectively.

When streaming stores will be generated for Knights Landing

The compiler is smart enough not to generate prefetches for lines for the streaming store, so as to avoid negating the value of the streaming stores. The Intel compiler generates streaming store instructions for Knights Landing only when:

(1) It is able to vectorize the loop and generate an aligned unit-strided vector unmasked store:

- Aligning stores, so they will be writing to a full cache line (vstore—64 bytes, no masks)

  Recommendation: Align the data appropriately at allocation time using align clauses, `aligned_malloc`, `-align array64byte` option on Fortran, and so on.
- Stores in the loop are aligned properly, and we can convey alignment information using directives/pragmas.

  Recommendation: Use a vector aligned directive before a loop to convey the alignment of all memory references inside loop including the stores.
- If the compiler aligns the store-access at runtime using a dynamic peel-loop based on its own heuristics (can occur in some cases, even when there is no directive to align the store-accesses).
- If the compiler could prove, based on its own alignment analysis, that the store accesses are aligned (at 64 bytes).

(2) Vector-stores are classified as nontemporal using one of:

- A nontemporal directive/pragma on the loop to mark the vector-stores as streaming

  Suggestion: Use a vector nontemporal directive/pragma before loop to mark aligned stores, or communicate the nontemporal-property of store using a vector nontemporal *A* directive where an assignment into array *A* is the store inside the loop.
- The compiler option `-opt-streaming-stores, which` forces the treatment of *all* aligned vector-stores as nontemporal.
  - This has the implicit effect of adding the nontemporal directive/pragma to all loops that are vectorized by the compiler in the compilation scope.
  - Unlike on Knights Corner, using this option has negative consequences if used incorrectly since the streaming-store instructions bypass the cache altogether.

Nontemporal: compiler generation of nontemporal stores

The Intel compiler has heuristics that determine whether a store is *streaming*. Usually this heuristic will kick in only when it can be determined at runtime that the loop has a large number of iterations. This is very limiting, so for applications where the calculation for the number of iterations is symbolic, there is a nontemporal directive/pragma that allows us to mark streaming stores:

```
#pragma vector nontemporal
```

For processors, including Knights Landing, this directive/pragma generates the nontemporal hints on stores. We shared an example of use of this directive/pragma and compared the code generated for Knights Corner and Knights Landing, in the Section *Nontemporal Stores/Cache Line Evicts* in Chapter 6. The compiler produces optimal code for either device from the same directive/pragma even though the instructions generated differ because Knights Landing resembles more recent Intel Xeon processors much more closely with its AVX-512 instructions.

COMPILER TIPS

In this section, we offer tips for using the Intel compiler which we offer that will help the achieve vectorization.

AVOID MANUAL LOOP UNROLLING

The Intel Compiler can typically generate efficient vectorized code if a loop structure is not manually unrolled. *Unrolling* means duplicating the loop body as many times as needed to operate on data using full vectors. For single precision on Knights Landing, this could mean unrolling 16 times. This means the loop body would do 16 iterations at once.

It is better to let the compiler do the unrolls, and we can control unrolling using `#pragma unroll N` (C/C++) or `!DIR$ UNROLL N` (Fortran). If *N* is specified, the optimizer unrolls the loop *N* times. If *N* is omitted, or if it is outside the allowed range, the optimizer picks the number of times to unroll the loop. Manual unrolling will generally interfere with compiler optimizations enough to hurt performance.

To add to this, manual loop unrolling tends to tune a loop for a particular processor or architecture, making it less than optimal for some future port of the application. Generally, it is good advice to write code in the most readable, straightforward manner. This gives the compiler the best chance of optimizing a given loop structure. Here is a Fortran example where manual unrolling is done in the source:

```
 m=MOD(N,4)
  if (m /=0) THEN
   do i=1, m
     Dy(i)=Dy(i)+Da*Dx(i)
  end do
if (N<4) RETURN
  end
if  mp1=m+1
 do i=mp1, N, 4
  Dy(i)=Dy(i)+Da*Dx(i)
  Dy(i+1)=Dy(i+1)+Da*Dx(i+1)
```

```
 Dy(i+2)=Dy(i+2)+Da*Dx(i+2)
 Dy(i+3)=Dy(i+3)+Da*Dx(i+3)
end do
```

It is better to express this in the simple form of:

```
do i=1,N
 Dy(i)=Dy(i)+Da*Dx(i)
end do
```

This allows the compiler to generate efficient vector-code for the entire computation and also improves code readability. Here is a C/C++ example where manual unrolling is done in the source:

```
double accu1=0, accu2=0, accu3=0, accu4=0;
double accu5=0, accu6=0, accu7=0, accu8=0;
for (i=0; i<NUM; i+=8) {
     accu1=src1[i+0]*src2+accu1;
     accu2=src1[i+1]*src2+accu2;
     accu3=src1[i+2]*src2+accu3;
     accu4=src1[i+3]*src2+accu4;
     accu5=src1[i+4]*src2+accu5;
     accu6=src1[i+5]*src2+accu6;
     accu7=src1[i+6]*src2+accu7;
     accu8=src1[i+7]*src2+accu8;
}
accu=accu1+accu2+accu3+accu4
    +accu5+accu6+accu7+accu8;
```

It is better to express this in the simple form of:

```
double accu=0;
for (i=0; i<NUM; i++) {
     accu=src1[i]*src2+accu;
}
```

REQUIREMENTS FOR A LOOP TO VECTORIZE (INTEL COMPILER)

Since a single iteration of a loop generally operates on one element of an array, but use of vector instructions depends on operating on multiple elements of an array at once, the "vectorization" of a loop essentially requires unrolling the loop so that it can take advantage of packed SIMD instructions to perform the same operation on multiple data elements in a single instruction.

The Intel compilers may be among the most capable and aggressive at vectorizing programs, but they still have limitations. In the most recent Intel compilers, vectorization is one of many optimizations enabled by default. Here is a list of requirements in order for a loop to potentially vectorize:

- If a loop is part of a loop nest, it must be the inner loop. Outer loops can be parallelized using OpenMP or autoparallelization (`-parallel`), but they cannot be vectorized unless the compiler is able either to fully unroll the inner loop or to interchange the inner and outer loops. Additional high-level loop transformations such as these may require use of the compiler option `-O3`.
- The loop must contain straight-line code (a single basic block). There should be no jumps or branches, but masked assignments are allowed. A "masked assignment" simply means an assignment controlled by a conditional (such as an IF statement).
- The loop must be countable, which means that the number of iterations must be known before the loop starts to execute although it need not be known at compile time. Consequently, there must be no data-dependent exit conditions.
- There should be no backward loop-carried dependencies (see Glossary for a definition). For example, the loop must not require statement 2 of iteration 1 to be executed before statement 1 of iteration 2 for correct results. This allows consecutive iterations of the original loop to be executed simultaneously in a single iteration of the unrolled, vectorized loop. Figs. 9.8 and 9.9 illustrate this requirement. The examples only make sense when thinking of doing multiple iterations at once due to vectorization. That is what would make Fig. 9.9 illustrate a barrier to vectorization.

There should be no special operators and no function or subroutine calls, unless these are inlined, either manually or automatically by the compiler. Intrinsic math functions such as `sin()`, `log()`, and `fmax()` are allowed, since the compiler runtime library contains vectorized versions of these functions. The following math functions may be vectorized by the Intel C/C++ compiler: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `log`, `log2`, `log10`, `exp`, `exp2`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`, `erf`, `erfc`, `erfinv`, `sqrt`, `cbrt`, `trunk`, `round`, `ceil`, `floor`, `fabs`, `fmin`, `fmax`, `pow`, and `atan2`. The list

```
for (i=1; i<MAX; i++) {
  a[i] = b[i] + c[i]
  d[i] = e[i] - a[i-1]
}
```

FIG. 9.8

Vectorizable: `a[i-1]` is always computed before it is used.

```
for (i=1; i<MAX; i++) {
  d[i] = e[i] - a[i-1]
  a[i] = b[i] + c[i]
}
```

FIG. 9.9

Not vectorizable: `a[i-1]` might be needed before it has been computed (if vectorized).

may not be exhaustive. Single-precision versions such as `sinf` may also vectorize. Both reductions and vector assignments to arrays are allowed, so feel free to use them. The Fortran equivalents, where available, should also vectorize.

Some additional advice on how to have a loop vectorize:

- Try to avoid mixing vectorizable data types in the same loop (except for integer arithmetic on array subscripts). Vectorization of type conversions may be either unsupported or inefficient. Support for the vectorization of loops containing mixed data types may be extended in a future version of the Intel compiler.
- Use code that accesses contiguous memory locations (loop over the first array index in Fortran, or the last array index in C, when using multidimensional arrays). The compiler may sometimes be able to vectorize loops with indirect or nonunit stride memory addressing; however the cost of gathering data from or scattering data back to memory is often too great to make vectorization worthwhile.
- Use the `ivdep` directive/pragma may be used to advise the compiler that there are no loop-carried dependencies that would make vectorization unsafe.
- Use the `vector always` directive/pragma to override the compiler's heuristics that determine whether vectorization of a loop is likely to yield a performance benefit.
- Refer to the vectorization section of an optimization report, to see whether a loop was or was not vectorized, and why.

IMPORTANCE OF INLINING, INTERFERENCE WITH SIMPLE PROFILING

Because vectorization happens on innermost loops consisting of straight-line code (see actual rules in the prior section), inlining of functions within the loop is critical. Turning off optimizations (the default is ON) will stop automatic inlining, and that will affect the ability of loops to vectorize if they contain function calls. Instrumenting for profiling using the `-pg` option will generally do the same, and stop optimization. Better profiling can be done using the less intrusive Intel VTune Amplifier (see Chapter 14). Because it requires no instrumentation that will defeat inlining, we can do profiling on the release version of an application.

COMPILER OPTIONS

There are many ways to share information that we know about our program that is not captured purely in the existing language. We selected a few of the most used and most effective compiler options to consider in order to boost the performance of an application:

- Use the ANSI aliasing option for C++ programs. This is not ON by default due to potential compatibility issues with older programs. It enables the compiler to do type-based disambiguation (asserts that pointers and floats do not overlap).

Without the option, the compiler may assume that the count for the number of iterations is changing inside the loop if the upper bound is an object-field access. Obeying strict ANSI aliasing rules provides more room for optimization. Hence, it is highly recommended to enable for ANSI-conforming code via `-ansi-alias` (Linux and OS X) or `/Qansi-alias` (Windows). This is already the default for the Intel Fortran Compiler. See an example coming up next in the section titled *Memory Disambiguation Inside Vector-Loops*.

Let the compiler if we do not alias arguments. On Linux and OS X, the option `-fargument-noalias` acts in the same way as applying the keyword `restrict` to all pointers of all function parameters throughout a compilation unit. This option is not available on Windows.

- Use loop count directives to give hints to compiler; this affects prefetch distance calculation, e.g., use "#pragma loop count (*number*)", before a loop where the number is the shortest distance (difference in iteration count) between a read and a write of the same memory location which does not occur entirely within the same iteration.
- The `restrict` keyword is a feature of the C99 standard that can be useful to C/C++ programmers. It can be attributed to pointers to guarantee that no other pointer overlaps the referenced memory location. Using the Intel C++ compiler does not only limit it to C99. It makes the keyword available for C89 and even for C++, simply by enabling a dedicated option: `-restrict` (Linux and OS X) or`/Qrestrict` (Windows).

MEMORY DISAMBIGUATION INSIDE VECTOR-LOOPS

Consider vectorization for a simple loop:

```
void vectorize(float *a, float *b, float *c, float *d, int n){
    int i;
    for (i=0; i<n; i++) {
        a[i]=c[i] * d[i];
        b[i]=a[i]+c[i] - d[i];
    }
}
```

Here, the compiler has no idea as to where those four pointers are pointing. As programmers, we may know they point to totally independent locations. The compiler thinks differently. Unless the programmer explicitly tells the compiler that they point to independent locations, the compiler has to assume the worst case aliasing is happening. For example, `c[1]` and `a[0]` may be at the same address and thus the loop cannot be vectorized at all.

When the number of unknown pointers is very small, the compiler can generate a runtime check and generate optimized and unoptimized versions of the loops (with overhead in compile time, code size, and also runtime testing). Since the overhead

grows quickly, that very small number has to be really small—like two—and even then we are paying the price for not telling the compiler that pointers are independent.

It is better to explicitly let the compiler known that pointers are independent. One way to do it is to use the C99 `restrict pointer` keyword. Even if we are not compiling with C99 standard, we can use the `-restrict` (Linux) and `-Qrestrict` (Windows) option to enable the Intel compilers to accept `restrict` as a keyword. In the following example, we tell the compiler that `a` is not aliased with anything else and `b` is not aliased with anything else:

```
void vectorize(float *restrict a, float *restrict b, float *c, float *d,
int n) {
    int i;
    for (i=0; i<n; i++) {
        a[i]=c[i] * d[i];
        b[i]=a[i]+c[i] - d[i];
    }
}
```

Another way is to use the `ivdep` directive/pragma. Semantics of `ivdep` is different from the restrict pointer, but it allows the compiler to eliminate some of the assumed dependencies—just enough to let the compiler think vectorization is safe.

```
void vectorize(float *a, float *b,
               float *c, float *d, int n) {
    int i;
#pragma ivdep
    for (i=0; i<n; i++) {
        a[i]=c[i] * d[i];
        b[i]=a[i]+c[i] - d[i];
    }
}
```

COMPILER DIRECTIVES

When we work to get the key part of an algorithm (usually a loop) to vectorize, we may quickly realize that there are two barriers: (1) getting the code to be vectorizable, (2) expressing the algorithm so the compiler accepts that it is vectorizable. The former is our responsibility as programmers. The latter is complicated by the nuances of programming languages. Even the most brilliantly conceived compiler cannot autovectorize many important loops without hints. In general, the presence of pointers that are too flexible in the C/C++ languages tends to make this problem most significant in C/C++ code (it is less so for Fortran code). Nevertheless, the challenge to

completely specify our intent so that a compiler will vectorize is why we have directives. In a later section, we will review array notation and elemental functions as more elegant coding styles that can do much the same thing if we rewrite our code using a better notation. The directive is a bit more like a hammer—we can use directives to force our current program to vectorize is we know it is safe. Code transformations to make the code vectorize is a prerequisite, and the directive simply finishes reassuring the compiler so that it can do the vectorization.

Traditionally, directives have been hints to the compiler to remove certain restrictions it may enforce. Assuming the restrictions that blocked vectorization were removed, and then the compiler will vectorize the code. These offer a way to work around the current safety heuristics the compiler is using. But, they are less portable and less predictable, as the heuristics in a compiler shift and change based on vendor and version. The vector and ivdep directives are examples—they give the compiler hints but still rely on the compiler checking and approving all concerns for which the hints do not apply.

The SIMD directives are a different approach; they force the compiler to vectorize and essentially place all the burden on the programmer to ensure correctness. Are the days of fighting the compiler over? Perhaps, but the programmer has to do the work and ensure correctness. For many, it is a dream come true: No compiler to fight. For many others, it is difficult without the compiler to help verify correctness.

SIMD DIRECTIVES

The SIMD directives enforce vectorization of loops. SIMD directives are an important capability to understand for vectorization work. SIMD directives come in two forms for C/C++ and Fortran, respectively:

```
#pragma omp simd [clause [[,] clause]...]
!$omp simd [clause[[,] clause]...]
```

Marking a `for` loop with `#pragma omp simd` similarly gives a compiler permission to execute a loop with vectorization. Note that `#pragma omp simd` is not restricted to inner loops. For example, the following code grants the compiler permission to vectorize the *outer* loop:

```
#pragma omp simd
for (int i=1; i<1000000; ++i) {
   while (a[i]>1)
   a[i] *= 0.5f;
}
```

In theory, a compiler can vectorize the outer loop by using masking to emulate the control flow of the inner `while` loop. Whether a compiler actually does so depends on the implementation. The Intel compiler detects and handles many forms of masking.

The simd directive/pragma can be combined with a parallel for directive/pragma. This asks the compiler to divide up a loop to run on multiple threads, and to vectorize the code for each thread. An example of this is shown:

```
char foo (char *a, int n) {
  int k;
  char c=0;
#pragma omp parallel for simd
  for (k=0; k<n; k++)
  c=c+a[n];
  return c;
}
```

Requirements to vectorize with SIMD directives

The SIMD directives ask the compiler to relax some of its requirements and to make every possible effort to vectorize a loop. A #pragma omp simd (C/C++), or !$omp simd (Fortran), behaves somewhat like a combination of #pragma vector always and #pragma ivdep but is more powerful. The compiler does not try to assess whether vectorization is likely to lead to performance gain, it does not check for aliasing or dependencies that might cause incorrect results after vectorization, and it does not protect against illegal memory references. Using #pragma ivdep overrides potential dependencies, but the compiler still performs a dependency analysis and will not vectorize if it finds a proven dependency that would affect results. With #pragma omp simd, the compiler does no such analysis and tries to vectorize regardless. It is the programmer's responsibility to ensure that there are no backward dependencies that might impact correctness. The semantics of #pragma omp simd are rather similar to those of the OpenMP #pragma omp parallel for. It accepts optional clauses such as COLLAPSE, REDUCTION, PRIVATE, and LASTPRIVATE. SIMD-specific clauses are aligned, and LINEAR, which can specify different strides for different variables. #pragma omp simd allows a wider variety of loops to be vectorized, including loops containing multiple branches or function calls (to functions using "omp declare simd").

Nevertheless, the technology underlying the SIMD directives is still that of the compiler, and some restrictions remain on what types of loop can be vectorized:

- The loop must be countable. This means that the number of iterations must be known before the loop starts to execute although it need not be known at compile time. Consequently, there must be no data-dependent exit conditions, such as break (C/C++) or EXIT (Fortran) statements. This also excludes most while loops. An example of a diagnostic message that occurs when this restriction is violated is:

  ```
  error: invalid simd pragma // warning #8410: Directive SIMD must be
  followed by counted DO loop.
  ```

- Certain special, nonmathematical operators and combinations of operators and of data types will result in diagnostic messages such as "operation not supported," "unsupported reduction," and "unsupported data type."
- Very complex array subscripts or pointer arithmetic. Typical diagnostic: "dereference too complex."
- Loops with very low number of iterations (also referred to as a *low trip count*) may not be vectorized. Typical diagnostic: "remark: loop was not vectorized: low trip count."
- Extremely large loop bodies (very many lines and symbols) may not be vectorized. The compiler has internal limits that prevent it from vectorizing loops that would require a very large number of vector registers, with many spills and restores to and from memory.
- SIMD directives may not be applied to Fortran array assignments.
- SIMD directives may not be applied to loops containing C++ exception handling code.

A number of the requirements detailed in the prior section "Requirements for a Loop to Vectorize (Intel Compiler)" are relaxed for SIMD directives, in addition to the above-mentioned ones relating to dependencies and performance estimates. Loops that are not the innermost loop may be vectorized in certain cases; more mixing of different data types is allowed; function calls are possible and more complex control flow is supported. Nevertheless, the advice in the prior section should be followed where possible, since it is likely to improve performance.

It is worth noting that with SIMD directives, loops are vectorized under the "fast" floating-point model, corresponding to `/fp:fast` (`-fp-model=fast`). The command line option `/fp:precise` (`-fp-model precise`) is not respected by a loop vectorized with a SIMD directive; such a loop might not give identical results to a loop without the directive. For further information about the floating-point model, see "Consistency of Floating-Point Results using the Intel Compiler" (listed in *For More Information* at the end of the chapter).

SIMD directive clauses

A SIMD directive can be modified by additional clauses, which control chunk size or allow for some C/C++ programmers' fondness for bumping pointers or indices inside the loop. The SIMD directives come in two forms for C/C++ and Fortran, respectively:

```
#pragma omp simd [clause [[,] clause]...]
!$omp simd [clause[[,] clause]...]
```

If we specify the SIMD directive with no clause, default rules will be used for variable attributes, vector length, and so forth. Misclassification of variables into `PRIVATE`, `LASTPRIVATE`, `LINEAR`, and `REDUCTION`, or the lack of appropriate classification of variables, may lead to unintended consequences such as runtime failures and/or incorrect results.

We can only specify a particular variable in at most one instance of a PRIVATE, LINEAR, or REDUCTION clause. If the compiler is unable to vectorize a loop, a warning occurs by default. However, if assert is specified, an error occurs instead. If the compiler has to stop vectorizing a loop for some reason, the fast floating-point model is used for the SIMD loop. A SIMD loop may contain one or more nested loops or be contained in a loop nest. Only the loop preceded by the SIMD directive is processed for SIMD vectorization.

The vectorization performed on a loop by the SIMD directive overrides any setting we may specify for options -fp-model (Linux OS and OS X) and /fp (Windows) for this loop. Valid clauses include:

```
LINEAR(var1:step1[, var2:step2]...)
```

var is a scalar variable; *step* is a compile-time positive, integer constant expression. For each iteration of a scalar loop, *var1* is incremented by *step1*, *var2* is incremented by *step2*, and so on. Therefore, every iteration of the vector loop increments the variables by VL*step1, VL*step2, …, to VL*stepN, respectively. If more than one step is specified for a var, a compile-time error occurs. Multiple LINEAR clauses are merged as a union. A variable in a LINEAR clause cannot appear in a REDUCTION, PRIVATE, or LASTPRIVATE clause.

```
REDUCTION(oper:var1[, var2]...)
```

oper is a reduction operator (+, *, -, .AND., .OR., .EQV., or .NEQV.); *var* is a scalar variable. Applies the vector reduction indicated by *oper* to *var1*, *var2*, …, *varN*. A SIMD directive can have multiple reduction clauses using the same or different operators. If more than one reduction operator is associated with a *var*, a compile-time error occurs. A variable in a REDUCTION clause cannot appear in a LINEAR, PRIVATE, FIRSTPRIVATE, or LASTPRIVATE clause.

```
PRIVATE(var1[, var2]...)
```

var is a scalar variable. Causes each variable to be private to each iteration of a loop. Its initial and last values are undefined upon entering and exiting the SIMD loop. Multiple PRIVATE clauses are merged as a union. A variable that is part of another variable (for example, as an array or structure element) cannot appear in a PRIVATE clause. A variable in a PRIVATE clause cannot appear in a LINEAR, REDUCTION, or LASTPRIVATE clause.

```
LASTPRIVATE(var1[, var2]...)
```

var is a scalar variable. Provides a superset of the functionality provided by the PRIVATE clause. Variables that appear in a LASTPRIVATE list are subject to PRIVATE clause semantics. In addition, when the SIMD loop is exited, each variable has the value that resulted from the sequentially last iteration of the SIMD loop (which may be undefined if the last iteration does not assign to the variable). A variable in a LASTPRIVATE clause cannot appear in a LINEAR, REDUCTION, or PRIVATE clause.

Use SIMD directives with care

The SIMD directives are the most powerful in forcing vectorization but come with the danger that the implications of vectorization need to be understood to avoid changing the program behavior in unexpected ways. They should be used with care. First-time users generally make mistakes and learn from them. We advise working on loops where the output values can be tested during development to provide rapid feedback. Some instructors advise "stick in SIMD directives and start debugging." We find that suggestion a little scary, but it seems to work with many developers quite well. Just be advised that it is hard to see the changes in a loop that are unexpected and problematic. On the other hand, SIMD directives free us from another evil: Everything looks good, but we cannot figure out how to get the compiler to do the vectorization. The SIMD directives force the compiler to vectorize, but we have to be careful because the burden is no longer on the compiler to keep things correct. Next, we discuss the vector and ivdep directives which give us some "in between" options, which share the burden by retaining some compiler checking but giving us more control.

THE VECTOR AND NOVECTOR DIRECTIVES

Unlike SIMD directives, vector and vector are generally hints to modify compiler heuristics. The vector and vector overrides the default heuristics for vectorization of FORTRAN `DO` loops and C/C++ `for` loops. They can also affect certain optimizations. Their format is:

```
!DIR$ vector [clause[[,] clause]...]
!DIR$ vector
#pragma vector [clause[[,] clause]...]
#pragma vector nontemporal[(var1[, var2, ...])]
```

clause is an optional vectorization or optimizer clause. It can be one or more of the following:

```
always [assert]
```

Enables or disables vectorization of a loop. The `ALWAYS` clause overrides efficiency heuristics of the compiler, but it only works if the loop can actually be vectorized. If the `assert` keyword is added, the compiler will generate an error-level assertion message saying that its efficiency heuristics indicate that the loop cannot be vectorized. We should use the ivdep directive to ignore assumed dependences or the SIMD directive to ignore virtually everything. This makes `vector ALWAYS` safer for the programmer in a way, but still leaves room for the compiler to be more conservative than might be strictly necessary. ivdep and SIMD both shift more burden to the programmer to check that the vectorization is safe (preserves the programmer's intent).

```
aligned | unaligned
```

Specifies that all data is aligned or no data is aligned in a loop. These clauses override efficiency heuristics in the optimizer. The clauses `aligned` and `unaligned` instruct the compiler to use, respectively, aligned and unaligned data movement instructions for all array references. These clauses disable all the advanced alignment optimizations of the compiler, such as determining alignment properties from the program context or using dynamic loop peeling to make references aligned.

Be careful when using the `aligned` clause. Instructing the compiler to implement all array references with aligned data movement instructions will cause a runtime exception if some of the access patterns are actually unaligned.

```
temporal | nontemporal [(var1 [, var2]...)]
```

var is an optional memory reference in the form of a variable name. This controls how the "stores" of register contents are performed (streaming versus nonstreaming). The `temporal` clause directs the compiler to use temporal (nonstreaming) stores. The `nontemporal` clause directs the compiler to use nontemporal (streaming) stores. For Knights Landing, the compiler generates clevict (cache-line-evict) instructions after the stores based on the nontemporal directive when the compiler knows that the store addresses are aligned.

By default, the compiler automatically determines whether a streaming store should be used for each variable. Streaming stores may enable significant performance improvements over nonstreaming stores. However, the misuse of streaming stores can significantly degrade performance.

```
vecremainder | novecremainder
```

Directs the compiler to vectorize (or not to vectorize) the remainder loop when the original loop is vectorized. If `!DIR$ vector ALWAYS` is specified, the following occurs: If neither the `vecremainder` or `novecremainder` clause is specified, the `ALWAYS` directive overrides efficiency heuristics of the compiler, and it determines whether the loop can be vectorized. If `vecremainder` is specified, the compiler vectorizes remainder loops when the original main loop is vectorized. If `novecremainder` is specified, the compiler does not vectorize the remainder loop when the original main loop is vectorized.

Use vector directives with care

The vector directive should be used with care. Overriding the efficiency heuristics of the compiler should only be done if we are absolutely sure the vectorization will improve performance.

For instance, the compiler normally does not vectorize loops that have a large number of nonunit stride references (compared to the number of unit stride references). In the following example, vectorization would be disabled by default, but the directive overrides this behavior:

```
!DIR$ vector ALWAYS
  do i=1, 100, 2
    ! two references with stride 2 follow
    a(i)=b(i)
  enddo
```

There may be cases where we want to explicitly avoid vectorization of a loop; for example, if vectorization would result in a performance regression rather than an improvement. In these cases, we can use the vector directive to disable vectorization of the loop.

In the following example, vectorization would be performed by default, but the directive overrides this behavior:

```
!DIR$ vector
   do i=1, 100
     a(i)=b(i)+c(i)
   enddo
```

THE IVDEP DIRECTIVE

Unlike SIMD directives, ivdep gives specific hints to modify compiler heuristics about dependencies. Specifically, the compiler will assume dependencies between loops when it cannot prove they do not exist. The ivdep directive/pragma instructs the compiler to ignore assumed vector dependencies. The format for C/C++ and Fortran, respectively, is:

```
#pragma ivdep

!DIR$ ivdep [: option]

!DIR$ ivdep with no option can also be specified as:
!DIR$ INIT_DEP_FWD (INITialize DEPendences ForWarD)
```

To ensure correct code, the compiler will prevent vectorization in the presence of assumed dependencies. The ivdep directive/pragma overrides that decision. Use ivdep only when we know that the assumed loop dependencies are safe to ignore.

In Fortran, the *option* LOOP implies no loop-carried dependencies and the option BACK implies no backward dependencies. When no option is specified, the compiler begins dependence analysis by assuming all dependences occur in the same forward direction as their appearance in the normal scalar execution order. This contrasts with normal compiler behavior, which is for the dependence analysis to make no initial assumptions about the direction of dependence.

ivdep example in fortran

In the following example, the ivdep directive provides more information about the dependences within the loop, which may enable loop transformations to occur:

```
!DIR$ ivdep
      DO I=1, N
         A(INDARR(I))=A(INDARR(I))+B(I)
      END DO
```

In this case, the scalar execution order follows:

(1) Retrieve `INDARR(I)`.
(2) Use the result from step 1 to retrieve `A(INDARR(I))`.
(3) Retrieve `B(I)`.
(4) Add the results from steps 2 and 3.
(5) Store the results from step 4 into the location indicated by `A(INDARR(I))` from step 1.

ivdep directs the compiler to initially assume that when steps 1 and 5 access a common memory location, step 1 always accesses the location first because step 1 occurs earlier in the execution sequence. This approach lets the compiler reorder instructions, as long as it chooses an instruction schedule that maintains the relative order of the array references.

ivdep examples in C

The loop in this example will not vectorize without the `ivdep` directive/pragma, since the value of k is not known; vectorization would be illegal if $k<0$:

```
void ignore_vec_dep(int *a, int k, int c, int m)
{
  #pragma ivdep
  for (int i=0; i<m; i++)
    a[i]=a[i+k] * c;
}
```

The directive/pragma binds only the `for` loop contained in current function. This includes a `for` loop contained in a subfunction called by the current function.

The following loop requires the parallel option in addition to the `ivdep` directive/pragma to indicate there are no loop-carried dependencies:

```
#pragma ivdep
for (i=1; i<n; i++)
{
  e[ix[2][i]]=e[ix[2][i]]+1.0;
  e[ix[3][i]]=e[ix[3][i]]+2.0;
}
```

The following loop requires the parallel option in addition to the `ivdep` pragma to ensure there is no loop-carried dependency for the store into `a()`:

```
#pragma ivdep
for (j=0; j<n; j++)
{
  a[b[j]]=a[b[j]]+1;
}
```

RANDOM NUMBER FUNCTION VECTORIZATION

The Intel Compiler supports a vectorized version of the random number function. This is the `drand48` family of random number functions in C/C++ and `RANF` and `Random_Number` functions (single and double precision) in Fortran. Fig. 9.10 shows the list of supported C/C++ functions and Figs. 9.11–9.16 show examples using them.

DATA ALIGNMENT TO ASSIST VECTORIZATION

Data alignment is a method to force the compiler to create data objects in memory on specific byte boundaries. This is done to increase efficiency of data loads and stores to and from the processor. Without going into great detail, processors are designed to efficiently move data when that data can be moved to and from memory addresses that are on specific byte boundaries. For Knights Landing, memory movement is optimal when the data starting address lies on 64-byte boundaries. Thus, it is desired

```
double drand48(void);
double erand48(unsigned short xsubi[3]);
long int lrand48(void);
long int nrand48(unsigned short xsubi[3]);
long int mrand48(void);
long int jrand48(unsigned short xsubi[3]);
```

FIG. 9.10

Supported C/C++ functions.

```
#include <stdlib.h>
#include <stdio.h>
#define ASIZE 1024
int main(int argc, char *argv[])
{
     int i;
     double rand_number[ASIZE] = {0};
     unsigned short seed[3] = {155,0,155};
     // Initialize Seed Value For Random Number
     seed48(&seed[0]);
     for (i = 0; i < ASIZE; i++) {
         rand_number[i] = drand48();
     }
     // Print Sample Array Element
     printf("%f\n", rand_number[ASIZE-1]);
     return 0;
}
```

FIG. 9.11

Example of drand48 vectorization.

```
#include <stdlib.h>
#include <stdio.h>
#define ASIZE 1024
int main(int argc,  char *argv[])
{
     int i;
     double rand_number [ASIZE] = {0};
     unsigned short seed[3] = {155,0,155};
     #pragma ivdep
     for (i = 0; i < ASIZE; i++) {
         rand_number[i] = erand48(&seed[0]);
     }
     // Print Sample Array Element
     printf("%f\n", rand_number[ASIZE-1]);
     return 0;
}
```

FIG. 9.12

erand38 vectorization; seed value is passed as an argument.

```
#include <stdlib.h>
#include <stdio.h>
#define ASIZE 1024
int main(int argc, char *argv[]) {
     int i;
     long rand_number[ASIZE] = {0};
     unsigned short seed[3] = {155,0,155};
     // Initialize Seed Value For Random Number
     seed48(&seed[0]);
     for (i = 0; i < ASIZE; i++) {
         rand_number[i] = lrand48();
     }
     printf("%ld\n", rand_number[ASIZE-1]);
     return 0;
}
```

FIG. 9.13

lrand38 vectorization.

```
#include <stdlib.h>
#include <stdio.h>
#define ASIZE 1024
int main(int argc,  char *argv[]) {
  int i;     long rand_number[ASIZE] = {0};
  unsigned short seed[3] = {155,0,155};
  #pragma ivdep
  for (i = 0; i < ASIZE; i++){
    rand_number[i] = nrand48(&seed[0]);
  }
  // Sample Array Element
  printf("%ld\n", rand_number[ASIZE-1]);
  return 0;
}
```

FIG. 9.14

nrand48 vectorization; seed value ID passed as an argument.

```
#include <stdlib.h>
#include <stdio.h>
#define ASIZE 1024
int main(int argc,  char *argv[]) {
  int i;
  long rand_number[ASIZE] = {0};
  unsigned short seed[3] = {155,0,155};
  // Initialize Seed Value For Random Number
  seed48(&seed[0]);
  for (i = 0; i < ASIZE; i++) {
    rand_number[i] = mrand48();
  }
  printf("%ld\n", rand_number[ASIZE-1]);
  return 0;
}
```

FIG. 9.15

mrand48 vectorization.

```
#include <stdlib.h>
#include <stdio.h>
#define ASIZE 1024
int main(int argc,  char *argv[]) {
  int i;
  long rand_number[ASIZE] = {0};
  unsigned short seed[3] = {155,0,155};
  #pragma ivdep
  for (i = 0; i < ASIZE; i++) {
    rand_number[i] = jrand48(&seed[0]);
  }
  printf("%ld\n", rand_number[ASIZE-1]);
  return 0;
}
```

FIG. 9.16

jrand48 vectorization; seed value is passed as an argument.

to force the compiler to create data objects with starting addresses that are modulo 64 bytes.

As programmers, we end up with two jobs: (1) align our data and (2) make sure the compiler knows it is aligned. Two problem areas for a compiler "knowing" about alignment are parameters into a subroutine/function and pointers. We can inform the compiler of alignments via pragmas (C/C++) or directives (Fortran) so that the compiler can generate optimal code. The one exception is that Fortran module data receives alignment information at USE sites. To summarize, two steps are needed:

1. Align the data
2. Use directives/pragmas in performance critical regions to tell the compiler that the data is aligned

Step 1: aligning the data

It is important for performance to align data. It is also important for optimization to inform the compiler of the alignment information in critical regions of the code. If we align data but do not tell the compiler, we can end up getting less optimal code and/or longer compilation time.

How to define aligned STATIC arrays

Here is an example for statically declaring a 1000-element single-precision floating-point array A on a 64-byte boundary, optimal on Windows C/C++:

```
__declspec(align(64)) float A[1000];
```

on Linux or OS X C/C++:

```
float A[1000] __attribute__((aligned(64)));
```

For Fortran simple arrays, the easiest way to get array data aligned is to use compiler option - `align array64byte` to get all arrays, static or dynamic, aligned on a 64-byte boundary. This option does not align data in COMMON blocks, nor elements within derived types. We can also align arrays with a directive. Directives in our code remove the need to use the -`align array64byte` compiler option by explicitly calling out the alignment in our code where the variable is declared. The compiler option applies to everything, whereas the directives can be selective. Using the compiler option is easier if you do not need to be selective. Here is an example of using the directive:

```
real :: A(1000)
!dir$ attributes align: 64:: A
```

For Fortran COMMON data, use -`align zcommons` to align all common block entities on 32-byte boundaries by adding padding bytes as needed. This is *not* the ideal data alignment for Knights Landing but is ideal for AVX. For Knights Landing, we have a 50/50 chance of full 64-byte alignment. Note: padding bytes may break many legacy applications that assume COMMON entities are packed and have no padding. So be sure to check results from the application to insure correctness with this compiler option.

For alignment of dynamic data in C/C++ we replace `malloc()` and `free()` with alignment-specified replacements `_mm_malloc()` and `_mm_free()`. The arguments are identical. These `_mm_` replacements provided by the Intel C++ Composer use the same argument and return types as `malloc()` and `free()`. The returned data from `_mm_malloc()` will be 64-byte aligned.

For alignment of dynamic data in Fortran we use the `-align arraynbyte` and `-align recnbyte` compiler options as discussed previously.

Step 2: inform the compiler of the alignment

Now that we have aligned our data, it is necessary to inform the compiler that the data is aligned. For example, when we pass data as arguments to a performance-critical function or subroutine, how does the compiler know if the arguments are aligned or

unaligned? The information must be provided by directives/pragmas since the compiler has no information on the arguments.

Here's an example in C/C++: For a specific variable, use the `__assume_aligned` macro to inform the compiler that a particular variable or argument is aligned. For example, to inform the compiler that an array passed as an argument or declared as global data is aligned we would do the following:

```
void myfunc(double p[]) {
  __assume_aligned(p, 64);
  for (int i=0; i<n; i++){
     p[i]++;
  }
}
void myfunc2(double *p2, double *p3, double *p4, int n) {
   for (int j=0; j<n; j+=8) {
    __assume_aligned(p2, 64);
    __assume_aligned(p3, 64);
    __assume_aligned(p4, 64);
    p2[j:8]=p3[j:8] * p4[j:8];
  }
}
```

Here's an example in Fortran: Use the directive `ASSUME_aligned`. The general syntax is:

```
cDEC$ ASSUME_aligned address1:n1 [, address2:n2]...
```

If we specify more than one `address:n` item, they must be separated by a comma. If the `address` is a Cray `POINTER` or it has the `POINTER` attribute, it is the `POINTER` and not the target of the pointer that is assumed aligned. If we specify an invalid value for *n*, an error message is displayed.

```
!dir$ assume_aligned A: 64
do i=1, N
A(I)=A(I)+1
end do

!dir$ assume_aligned A: 64
A=A+1
```

How to tell the compiler all memory references are nicely aligned for the target

A more general directive/pragma can be put in front of a loop to tell the compiler that *all* data in the loop is aligned. In this way, we do not have to specify alignment individually for every data element being accessed.

Example in C/C++

```
#pragma vector aligned
for (i=0; i<n; i++){
   A[i]=B[i] * C[i]+D[i];
}
//Add pragma just before an array-notation stmt to
//specify alignment for arrays used
#pragma vector aligned
A[0:n]=B[0:n] * C[0:n]+D[0:n];
```

Examples in Fortran:

```
!DIR$ vector aligned
do I=1, N
  A(I)=B(I) * C(I)+D(I)
end do

!DIR$ vector aligned
A=B * C+D
```

Some notes: These clauses override the efficiency heuristics in the compiler. These clauses cause the compiler to use aligned data movement instructions for all array references. These clauses disable all the advanced alignment optimizations of the compiler, such as determining alignment properties from the program context or using dynamic loop peeling to make references aligned. Be careful when using these clauses. Instructing the compiler to implement all array references with aligned data movement instructions will cause a runtime exception if some of the access patterns are actually unaligned.

USE ARRAY SECTIONS TO ENCOURAGE VECTORIZATION

Using array sections in Fortran makes our intentions very clear, which in turn helps the compiler have the information it needs to vectorize the loop because it is has less of the implied dependencies that creep up from the use of loops.

FORTRAN ARRAY SECTIONS

An array section is a portion of an array that is an array itself. It is an array sub-object. A section subscript list (appended to the array or array component) determines which portion is being referred to. A reference to an array section takes the following form `array(sect-subscript-list)`, where `array` is the name of the array, *`sect-subscript-list`* is a list of one or more section subscripts (subscripts, subscript triplets, or vector subscripts) indicating a set of elements along a particular dimension. At least one of the items in the section subscript list must be a subscript triplet or vector subscript. A subscript triplet specifies array elements in increasing or decreasing

order at a given stride. A vector subscript specifies elements in any order. Each subscript and subscript triplet must be a scalar integer (or other numeric) expression. Each vector subscript must be a rank-one integer expression.

If no section subscript list is specified, the rank and shape of the array section is the same as the parent array. Otherwise, the rank of the array section is the number of vector subscripts and subscript triplets that appear in the list. Its shape is a rank-one array where each element is the number of integer values in the sequence indicated by the corresponding subscript triplet or vector subscript.

If any of the sequences are empty, the array section has a size of zero. The subscript order of the elements of an array section is that of the array object that the array section represents.

Each array section inherits the type, kind type parameter, and certain attributes (`INTENT`, `PARAMETER`, and `TARGET`) of the parent array. An array section cannot inherit the `POINTER` attribute. If an array (or array component) is of type character, it can be followed by a substring range in parentheses.

Subscript triplets

A subscript triplet is a set of three values representing the lower bound of the array section, the upper bound of the array section, and the increment (stride) between them. It takes the following form

```
[first-bound] : [last-bound] [:stride],
```

where `first-bound` is a scalar integer (or other numeric) expression representing the first value in the subscript sequence. If omitted, the declared lower bound of the dimension is used, `last-bound` is a scalar integer (or other numeric) expression representing the last value in the subscript sequence. If omitted, the declared upper bound of the dimension is used. When indicating sections of an assumed-size array, this subscript must be specified.

The `stride` is a scalar integer (or other numeric) expression representing the increment between successive subscripts in the sequence. It must have a nonzero value. If it is omitted, it is assumed to be 1. If the stride is positive, the subscript range starts with the first subscript and is incremented by the value of the stride, until the largest value less than or equal to the second subscript is attained. If the first subscript is greater than the second subscript, the range is empty. If the stride is negative, the subscript range starts with the value of the first subscript and is decremented by the absolute value of the stride, until the smallest value greater than or equal to the second subscript is attained. If the second subscript is greater than the first subscript, the range is empty. If a range specified by the stride is empty, the array section has a size of zero.

A subscript in a subscript triplet need not be within the declared bounds for that dimension if all values used to select the array elements are within the declared bounds. For example, if an array has been declared as A(15), the array section specified as A(4:16:10) is valid. The section is a rank-one array with shape (2) and size 2. It consists of elements A(4) and A(14).

If the subscript triplet does not specify bounds or stride, but only a colon (:), the entire declared range for the dimension is used. If all subscripts are omitted, the section defaults to the entire extent in that dimension.

Vector subscripts

A vector subscript is a one-dimensional (rank one) array of integer values (within the declared bounds for the dimension) that selects a section of a whole (parent) array. The elements in the section do not have to be in order, and the section can contain duplicate values. An array section with a vector subscript that has two or more elements with the same value is called a many-one array section. A many-one section must not appear on the left of the equal sign in an assignment statement, or as an input item in a `READ` statement.

Implications for array copies, efficiency issues

The Fortran language semantics sometimes require the compiler to make a temporary copy of an array or array slice. Situations where this can occur include:

- Passing a noncontiguous array to a procedure that does not declare it as assumed-shape
- Array expressions, especially those involving `RESHAPE`, `PACK`, and `MERGE`
- Assignments of arrays where the array appears on both the left- and right-hand sides of the assignment
- Assignments of `POINTER` arrays

By default, these temporary values are created on the stack and, if large, may result in a "stack overflow" error at runtime. The size of the stack can be increased, but with limitations dependent on the operating system. Use of the `/heap-arrays` (Windows) or `-heap-arrays` (Linux) compiler option tells the compiler to use heap allocation, rather than the stack, for such temporary copies. Heap allocation adds a small amount of overhead when creating and removing the temporary values, but this is usually inconsequential in comparison to the rest of the code.

Performance can be further improved by eliminating the need for a temporary copy entirely. For the first case earlier, passing a noncontiguous array to a procedure expecting a contiguous array, enabling the `/check: arg_temp_created` (Windows) or `-check arg_temp_created` (Linux) compiler option, will produce a runtime informational message when the compiler determines that the argument being passed is not contiguous. A runtime test is made and, if the argument is contiguous, no copy is made. However, this option will not issue a diagnostic for other uses of temporary copies.

One way to avoid temporary copies for array arguments is to change the so-called procedure to declare the array as assumed-shape, with the `DIMENSION(:)` attribute. Such procedures require an explicit interface to be visible to the caller. This is best provided by placing the called procedure in a module or a `CONTAINS` section. As an alternative, an `INTERFACE` block can be declared.

Use of POINTER arrays makes it difficult for the compiler to know if a temporary value can be avoided. Where possible, replace POINTER with ALLOCATABLE, especially as components of derived types. The language definition allows the compiler to assume that ALLOCATABLE arrays are contiguous and that they do not overlap other variables, unlike POINTERs.

Another situation where temporary values can be created is for automatic arrays, where an array's bounds are dependent on a routine argument, or COMMON variable, and the array is a local variable in the procedure. As earlier, these automatic arrays are created on the stack by default; the/heap-arrays (Windows) or -heap-arrays (Linux) compiler option will create them on the heap. Consider making such arrays ALLOCATABLE instead; local ALLOCATABLE variables that do not have the SAVE attribute are automatically deallocated when the routine exits. For example, replace:

```
SUBROUTINE SUB (N)
INTEGER, INTENT(IN) :: N
REAL :: A(N)
```

with:

```
SUBROUTINE SUB(N)
INTEGER, INTENT(IN) :: N
REAL, ALLOCATABLE :: A(:)
ALLOCATE (A(N))
```

LOOK AT WHAT THE COMPILER CREATED: ASSEMBLY CODE INSPECTION

There are several ways to gain insight into how well applications were vectorized for Knights Landing. The Vectorization Intensity performance metric, discussed in Chapter 14, quantifies the efficiency of an application's vectorization in terms of how many SIMD operations are run. Unfortunately, as our Chapter 22 example will illustrate, Knights Landing counters (used by VTune) do not account for masking. Therefore, a vector instruction that only does one operation (7 double, or 15 single, operations being masked off) will count as fully vectorized. The Vector Advisor (Chapter 10) has a solution for this.

The vectorization section of the optimization report, introduced in this chapter, gives detailed information on which loops vectorized. Another handy tool for judging vectorization is assembly code inspection.

Visual inspection of assembly code can help identify performance problems that may merit further investigation. Understanding generated assembly code and its impact on application execution is a complex subject that can require years of study. For discussion of optimal code sequences, we recommend reading Chapter 6. To help, we will explain how to find the assembly code we may wish to study.

HOW TO FIND THE ASSEMBLY CODE

There are two methods for obtaining the assembly code for your Knights Landing application. The first is by using the `-S` option (Linux) for the Intel Compiler. This will produce an assembly file instead of a typical executable, and it will end in.s unless otherwise specified. If an offload application (see Chapter 18) is being compiled to assembly, two files will be generated: one with a `.s`, and the other ending in `MIC.s`, which will contain the specific binary to be run on the target of the offload. When using the `-S` option, we recommend removing `-g` (debug information) as it will remove a great deal of extra symbolic labeling in the assembly file and make it easier to read. Looking at an assembly listing presents several challenges though. The sheer length of most assembly listings can be intimidating. A trick for finding relevant code regions in an Intel Compiler assembly file is to search for "`#linenumber`" in the file (for example, "`#206`" for line 206). Even after doing this, however, you will often see hundreds of lines of assembly corresponding to one or two lines of source. For example, the compiler will typically try to align one or more memory references in a `for` loop dynamically. If, for example, a loop is accessing floats in an array and the first reference happens to be to a piece of data that is the 11th element in a cacheline, the compiler may generate three loops. With AVX-512 predication this is less frequent than it was previously. The first is a "peel" loop, which will perform the first five iterations of the loop (accessing the last five elements in the first cacheline). Then will be the main loop, accessing elements aligned to cachelines (16 elements per line). Then will be a remainder loop, to access any remaining elements in the final cacheline. Many other types of performance enhancements the compiler performs have the effect of making the assembly language very different from a straight-line interpretation of the original source. Another source of confusion in assembly files can come from parallelism libraries like OpenMP, which create a function for the parallel region and result in loop code being found in a different location than expected.

Given the challenges outlined earlier, many people prefer to use the second method for viewing assembly, which is the source/assembly viewer in Intel® VTune™ Amplifier. VTune Amplifier can display the assembly for an application you are analyzing without needing any special compilation options. If symbolic information is available, the assembly code can be displayed side-by-side with the original source code. When a line of source code is selected, the corresponding assembly lines are displayed. This makes it easier to locate the assembly code of interest, but does not completely solve the issue of sorting out things like where the body of a loop that has been parallelized will appear in the assembly. Another way that the Intel VTune Amplifier can make things easier: Loop analysis. Using loop analysis mode, VTune Amplifier will display hotspot information for the loops in your code instead of the functions.

Chapter 6 gives more insight into optimal instructions sequences which can help understand what to look for in the assembly, for example, gather/scatter usage.

NUMERICAL RESULT VARIATIONS WITH VECTORIZATION

Generally, vectorized reductions do not produce numerically identical results to scalar loops. Additionally, sometimes they may not be reproducible from one run to the next due to concurrency, even for identical executables running on identical data on the identical processor or coprocessor. If this is an issue, we can try the `-fp-model precise`, which, among other things, disables vectorization of reductions. Performance maybe impacted by such an option.

SUMMARY

The vector parallel capabilities of Knights Landing are utilized in the same manner as vectorization for other Intel processors. The big difference being that the level of performance possible due to the extra wide vectors makes it more important than for processors with smaller vector capabilities. Best use of the vector capability requires vectorization to create the vector instructions and good data layout and streaming to enable efficient movement of data to and from the vector instructions. This gives rise to many models to expose the vector parallelism, so vectorization occurs, as well as influencing prefetching of data into levels of the data cache. This chapter gave the fundamentals, which are the same techniques we would find in most any tutorial or reference on vectorization for processors. The Intel compiler documentation has more examples and options for advanced usage.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this chapter.

- OpenMP standard (for information on the SIMD directives), http://openmp.org.
- Intel documentation including compiler reference manuals: https://software.intel.com/en-us/intel-parallel-studio-xe-support/documentation.
- *Vectorization Toolkit* also includes links to additional resources for a step-by-step vectorization methodology. http://lotsofcores.com/intelveckit.
- "Cache Blocking Techniques" provides more background on this general purpose technique that helps processor and coprocessor performance http://lotsofcores.com/intelcacheblock.
- "Memory Layout Transformations" provides more background the "array of structures" vs "structures of array" topic. Understanding this general-purpose transformation can motivate changes to benefit processor and coprocessor performance. http://lotsofcores.com/intelmemlayout.
- "Fortran Array Data and Arguments and Vectorization," various Fortran array data types and arguments are vectorized by the Intel compiler. This information

may be helpful in generating effective vectorized programs. http://lotsofcores.com/intelfortranarrays.

- "Consistency of Floating-Point Results using the Intel® Compiler," http://lotsofcores.com/intelfpconsist.
- "Quick-Reference Guide to Optimization with Intel® Compilers," http://software.intel.com/en-us/intel-composer-xe/.
- A methodology for tuning prefetching: High-Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, Chapter 21—Prefetch Tuning Optimizations, ISBN 978-0-12-803819-2.

CHAPTER

10 Vectorization advisor

What is new with Knights Landing in this chapter?
AVX-512 and the Vectorization Advisor within the Intel® Advisor tool.

Many factors impact achieving good vectorization for our applications. The Vectorization Advisor directly analyzes an application and provides feedback on the extent of current vectorization and on possible steps to achieve more effective vectorization. Vectorization Advisor works with any compiler although some features in the Intel® compilers will increase the effectiveness of advice from the Vectorization Advisor tool. It is like having an expert sitting next to us who never tires of digging into an application to analyze what is really happening.

The Vectorization Advisor is one of the two major *workflows* (feature sets) available in the Intel® Advisor "2016" and later versions. The Intel Advisor also includes a thread prototyping feature set which can be useful for analysis of scaling for threads. In this chapter, we focus on using the Vectorization Advisor to help us maximize our vectorization performance.

How close is my application to maximum performance? Insight into this is helped by a "roofline model" analysis, in the Advisor Roofline Report section.

Vectorization Advisor lets us:

- For unvectorized loops, discover **what prevents code from being vectorized** and **get tips** on how to vectorize it.
- For vectorized loops that use modern SIMD instructions, **measure** their **performance efficiency** and **get tips** on how to increase it.
- For both vectorized and unvectorized loops, explore how the **memory layout and data structures** can be made more **vector friendly**.

Much of the functionality discussed in this chapter relies on the "2017" version of Intel® Advisor, scheduled for release in Q3 2016 and available as beta prior to that. We advise using the latest version possible to get the most benefits, as the product is constantly refined based on customer feedback.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00010-7

After an introduction, this chapter focuses on analysis of AVX-512 vectorization, and then an impressive collection of critical items related to effective vectorization: Memory Access Patterns, Gather/Scatter Analysis, Mask Utilization and FLOPs counting, Roofline analysis, and testing AVX-512 with a simulator. Finally, we walk through an example of usage on a chemistry code.

Counting FLOPs accurately: Because of masking registers, the most accurate way to count floating point operations on Knights Landing is with the FLOPs profiler, and is discussed in the Mask Utilization and FLOPs Profiler section.

GETTING STARTED WITH INTEL ADVISOR FOR KNIGHTS LANDING

Using Vectorization Advisor follows an iterative cycle as shown in Fig. 10.1. (The steps in dotted line boxes or in brackets are optional.)

- Build an optimized version of the application. Remember to generate debug information using the `-g` (Linux OS) or `/Zi` (Windows OS) compiler option. Use the same binary in all four analysis steps.

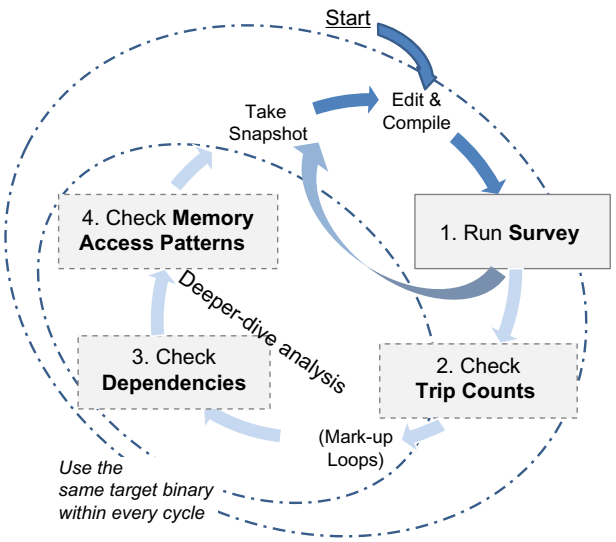


FIG. 10.1

Vectorization analysis lifecycle for the Intel (Vectorization) Advisor.

- Run a **Survey** analysis to look for hotspots data (how long each loop runs), compiler optimization messages, the type of vectorization used, and detailed static analysis of code generation and instruction set architecture (ISA) usage. The **Survey Report** elegantly combines the results of the compile-time analysis, static analysis of contributing binaries, and runtime workload metrics. The **Survey Report** also offers **Recommendations** we can use in optimization efforts.
- (Optional) Run a **Trip Counts** analysis to see how many times loops run.
- (Optional) Mark up loops of interest for deeper-dive **Refinement** analyses. For selected loops:
  - (Optional) Run a **Dependencies** analysis to see if there are loop-carried dependencies.
  - (Optional) Run a **Memory Access Patterns (MAP)** analysis to see how loops iterate through memory.
- (Optional) Before experimenting with various optimization techniques, take a **Snapshot** of your current data to preserve **Survey**, **Trip Counts**, and **Refinement** reports data for future reference.

We can use all parts of the Vectorization Advisor with any compiler. Also, most AVX-512-specific features in the Vectorization Advisor are compiler independent. However, the **Survey** analysis excels when coupled with Intel compilers, because in addition to static and dynamic analysis data it gives a more user-friendly and interactive view of various Intel compiler-generated reports. Nevertheless, a solid subset of metrics is available for binaries built with other compilers.

The following Vectorization Advisor capabilities have been introduced or specifically optimized to help with AVX-512 analysis:

- *Survey Report — AVX-512-aware Recommendations*. Detects performance issues or unnecessary performance overhead, then provides user-friendly suggestions on how to fix the issue and improve vector code quality and performance. Many Advisor **Recommendations**, such as *Ineffective peeled/remainder loop(s) present*, are specifically tuned to help developers enable effective AVX-512 usage.
- *Survey Report — AVX-512 Traits*. This expert-oriented feature highlights AVX-512 instructions whose presence or absence could significantly affect vector code performance.
- *Mask utilization and FLOPs profiler*. The best way to count FLOPs and estimate floating-point operations per second (FLOP/s) for Knights Landing.
- *MAP Report with new Gather/Scatter analysis*. It is important to understand performance trade-offs and potential tuning strategies for code using gather/scatter instructions.
- *MAP Report — Recommendations*. In certain cases, it is possible to replace gather/scatter instructions with faster sequences of instructions by adjusting an algorithm, using certain language keywords, or falling back to special kinds of intrinsics. These new **Recommendations** guide the replacement process.

ENABLING AND IMPROVING AVX-512 CODE WITH THE SURVEY REPORT

In this section, we discuss capabilities of Advisor Survey Report populated by lightweight **Survey** and **Trip Count** Analysis types.

PREPARING YOUR APPLICATION

As shown in Fig. 10.1, before running a **Survey** analysis, prepare an optimized version of the application to run on Knights Landing.

To generate an executable that leverages AVX-512 instructions and runs on Knights Landing, use one of these options when compiling on Intel C, C++, or Fortran compilers:

- `-xCOMMON-AVX512`— Intel AVX-512 is *common* for Intel Xeon processors and Knights Landing. It enables usage of AVX-512 Foundation (AVX-512F) and AVX-512 Conflict (Free) Detection Instructions (AVX-512CD).
- `-xMIC-AVX512`— Intel AVX-512 *includes* instructions supported currently only by Knights Landing. In addition to AVX-512F and AVX-512CD, it enables usage of AVX-512 Exponential and Reciprocal instructions of high accuracy (AVX-512ER) and AVX-512 new Prefetch Instructions (AVX-512PF).

Also enable at least a second level of optimization (`-O2`) and debug info generation (`-g`).

This chapter uses Linux OS-specific syntax for compiler options. In order to compile code for Windows OS targets, change the `-` to `/Q`, for example, change `-xCOMMON-AVX512` to `/QxCOMMON-AVX512`.

RUNNING A SURVEY ANALYSIS WITH TRIP COUNTS

After preparation steps, run **Survey** analysis on a Knights Landing processor machine in exactly the same way as on other Intel-based processor platforms. For advanced users, there are three additional ways to run and view **Survey** data on Knights Landing.

1. Run a **Survey** analysis and **Trip Counts** analysis on Knights Landing, then view the resulting **Survey Report** directly on a Knights Landing system. Use the combination of Intel Advisor command line interface (CLI) and GUI. This is the simplest usage scenario. For example:
   - Set the environment variables
     ```
     $ source/opt/intel/advisor_xe_2016/advixe-vars.sh
     ```
   - Collect the **Survey** data with automatic finalization
     ```
     $ advixe-cl --collect=survey --
     projectdir=<project_directory> -- <target_application>
     ```

where `<project_directory>` is the full path to the target folder where we want to keep Intel Advisor data

- Launch the Intel Advisor GUI and open the collected data

```
$ advixe-gui <project_directory>
```

2. Run a **Survey** analysis and **Trip Counts** analysis on a "target" Knights Landing system using the Intel Advisor CLI, then view the resulting **Survey Report** on your own ("host") system using the Intel Advisor GUI. The target and host systems can even have different operating systems. We can run a survey on the command line on a Linux system, and then view the results on a Windows desktop — either by using a mapped drive, or by copying the results directory (usually named e000) from a Linux target system.

   This approach is also beneficial when we want to profile an MPI application running on multiple cluster nodes that do not have a GUI subsystem available. In such cases, collect all the data on cluster nodes using the Intel Advisor CLI (wrapped by an MPI launcher). Then view the data by copying it to a head node or laptop and running the Intel Advisor GUI there.
3. Run a **Survey** analysis and **Trip Counts** analysis on a target Knights Landing system using `a -no-finalize` option. Specify a path to the application source/binaries, then finalize and view the resulting **Survey Report** on the host system.

   While this last option is the most sophisticated, we still might consider it to reduce the time consumed by result *finalization* process. While Intel Advisor finalization is partially parallelized, it still has serial-by-nature phases that may work slightly faster on Intel Xeon processors (well optimized for executing sequential, non-parallelized code).

ONE-STOP-SHOP PERFORMANCE OVERVIEW IN THE SURVEY REPORT

The Vectorization Advisor user interface is designed to bring together all the important vectorization features of a code into one place — almost like a one-stop shop. Consider the **Survey** and **Trip Counts Report** example shown in Fig. 10.2.

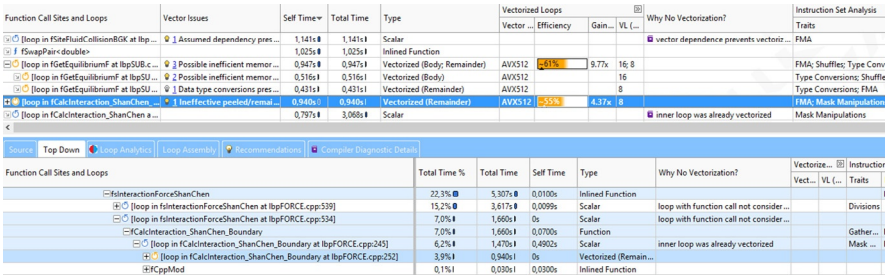


FIG. 10.2

Survey Report (data obtained on Knights Landing after profiling AVX-512-compiled DL_MESO application discussed in a later section).

The top part of Fig. 10.2 shows a list of time-consuming loops and (optionally) functions. The bottom part shows navigable tabs that provide additional details about source code, program tree, assembly, etc., for a loop or function selected in the top part.

The **Type** column shows that two loops are already vectorized. The **Vectorized Loops** columns show these loops are vectorized using Intel AVX-512 and have the potential to run 4.37× and 9.77× faster than their original scalar (non-vectorized) versions.

The other two loops are scalar. The **Why No Vectorization?** column explains what prevents the compiler from vectorizing these scalar loops.

In order to understand SIMD and Threading parallelism potential, it is often crucial to get knowledge about loop *call counts* and *trip counts* (the number of times loop is invoked and number of times loop bodies execute). Intel Advisor technology makes it possible to obtain given information using the **Trip Counts** analysis.

The **Survey Report** (especially when it contains **Trip Counts** data) is normally sufficient to effectively categorize all time-consuming loops (hotspots) as follows:

- **Non-vectorizable** kernels
- **Vectorizable**, but not vectorized loops that require minimal program changes (often with the help of OpenMP 4.x) to enable compiler-driven SIMD parallelism
- **Inefficiently vectorized** loops whose performance could be improved using low-hanging optimization techniques
- **Inefficiently vectorized** loops whose performance is limited by data layout or other fundamental limitations (and thus require code refactoring or algorithmic change to further speed up execution)
- **Vectorized** loops that perform well

Even when code can be vectorized, simply enabling vectorization does not always lead to a performance improvement. That's why it is important to examine the vectorized loops to confirm they are performing well and, if not, find out how to change the situation.

In an ideal world, the goal is to transform most vectorizable loops into efficiently vectorized loops. The **Vector Issues** and **Why Not Vectorized?** columns data can help achieve that goal. The **Vector Issues** column highlights vectorization ineffectiveness reasons and provides suggestions on how to improve performance. The **Vector Issues** column may also highlight some inefficiencies for scalar loops, especially in cases where it is clear that straightforward vectorization of a loop will not result in solid speedups.

Use the **Recommendations** tab to learn more about specific vector issues and find out how to fix them. The **Compiled Diagnostic Details** tab provides more context for the **Why No Vectorization?** column, normally giving an example and guidance of how to enable loop vectorization.

Vectorization Advisor **Recommendations** make it possible to figure out optimization steps and required source modifications without digging into architectural or assembly language details. This greatly benefits developers who want to focus on

writing source code in high-level programming languages using programming standards like OpenMP 4.x. For performance experts or *ninjas* who prefer to jump directly to the assembly level, the **Survey Report** offers low-level, detailed **Assembly** views.

Also, in practice, there is often a need for data representation *in between* programmer-oriented (source code) and *ninja*-oriented (assembly) levels. For developers who prefer something between high-level and low-level data representations, the **Survey Report** introduces **Traits**. Each trait indicates the presence of a specific subset of instructions, eliminating the need to read assembly. Intel Advisor AVX-512 **Recommendations** and **Traits** are described in the next sections.

ENABLING AVX-512 SPEEDUPS VIA RECOMMENDATIONS

This subsection discusses the three most-frequent, AVX-512-related recommendations and associated optimization techniques. This will help better understand typical vectorization trade-offs on Knights Landing and will give additional guidance on how to most effectively apply Advisor Recommendations or how to *manually* detect performance penalty cases using *raw* Advisor data.

Fixing ineffective AVX-512 peeled/remainder loop issues

Let's start by discussing a low-hanging optimization technique that makes it possible to transform an improvable vectorized loop into a highly efficient vectorized loop. Check the second loop in the **Survey Report** (hot loop at `fGetEquilibriumF` function) in Fig. 10.2. Notice the compiler generated two versions for it:

- A vectorized loop body, accounting for ~60% of total loop time. The Vector Length (VL) is 16; that is, 16 integers (or a single-precision floating-point) held in the 512-bit-wide AVX-512 registers.
- A vectorized remainder loop that consumes almost 40% of total loop time. The VL for the remainder loop is only 8. (In cases of unaligned data processing, a scalar or vectorized peeled loop could also be generated.)

The concept of peeled/remainder/body loops is illustrated in Fig. 10.3, where a typical breakdown of loop iterations is shown for an AVX2 loop processing double-precision floating-point computations.

The remainder loop exists because the loop (trip) count is not a multiple of the VL. The wider the vector register is, the greater the chance the compiler will have to generate a remainder loop. AVX-512 remainder loops occur more frequently than AVX2 or AVX remainder loops.

The presence of remainder loops for loops with a low total trip count may have a negative impact on the maximum achievable speedup. For an AVX-512 platform, the negative impact of remainder loops is caused by a smaller VL in the remainder loop and by the need to use masking. For SSE/AVX/AVX2 platforms, the negative impact is amplified by the need to use a scalar form of the remainder loop.

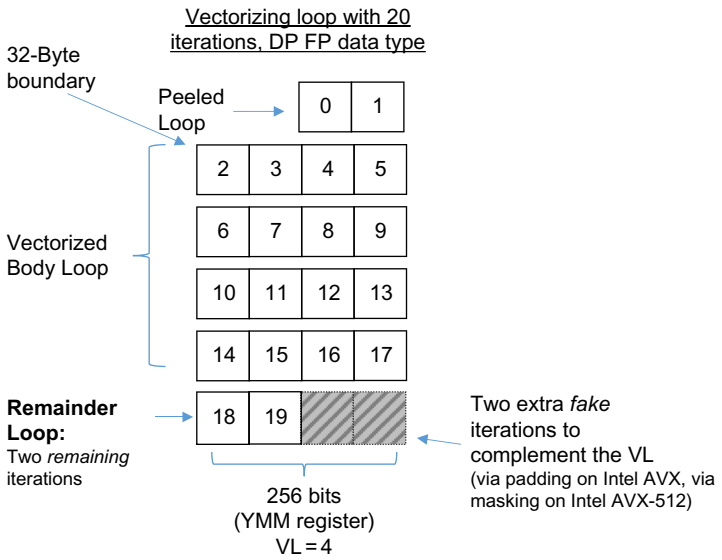


FIG. 10.3

Typical AVX2 iterations breakdown of peeled/remainder/body loops for double-precision, floating-point computations.

Notice the two *fake* or *wasted* iterations in the bottom right corner of Fig. 10.3. One technique we can apply to this code: increase the loop iterations count to become a multiple of VL. This is called *data padding* and is the first optimization technique the Vectorization Advisor suggests for SSE/AVX code in the **Recommendations** tab for this loop (as seen in Fig. 10.4).

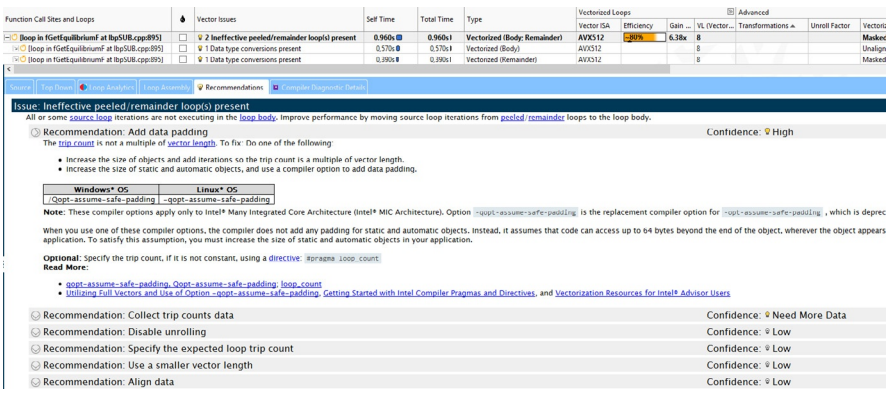


FIG. 10.4

Ineffective peeled/remainder loop issue and associated Vectorization Advisor Recommendations.

To pad the data, we normally increase the size of the arrays processed in the loop, so that accessing the (unused/fake) locations does not cause a segmentation violation or similar problem. Loop padding and more generally peeled/remainder loop optimization are good examples of relatively low-hanging-fruit optimization techniques applicable in a wide class of real-world HPC applications.

To guide developers, the Vectorization Advisor **Recommendations** feature performs a complex analysis of the following:

- Body loop versus remainder loop CPU time breakdown, available thanks to the synergy between static Intel compiler diagnostics and dynamic hotspots analysis
- Dynamic trip counts and static VL knowledge
- Unroll factor
- Internal compiler cost/benefit model

Putting all these data together makes it possible for the Vectorization Advisor to evaluate potential performance impact on a case-by-case basis, and to identify the most impactful optimization techniques for a loop. The analysis is performed automatically by the Vectorization Advisor **Recommendations** engine, which chooses loops with maximum return on investment and automatically suggests the next optimization steps. (Or experienced developers can explore the metrics in the **Survey Report** or the **Loop Analytics** tab to decide on next optimization steps themselves.)

Ineffective peeled/remainder loops are a good example of a performance issue that should be mitigated differently on AVX-512 than on AVX2. The reason is the introduction of eight dedicated 64-bit mask registers (k0-7) in AVX-512. Mask registers are normally used to identify the SIMD lanes to be processed by AVX-512 computational instructions and/or loaded/stored from vector registers/memory.

From the peeled/remainder loop perspective, mask registers make it possible to complement remaining iterations to VL without using loop padding, avoiding scalar remainder loop usage. The AVX-512 variation in Fig. 10.3 will always mask the fake *gray* iterations using the zero mask bit, so the corresponding data is never loaded/stored, but the maximum register width is still utilized. Masking is more beneficial than manual padding because it minimizes the number of wasted computations using hardware-supported capabilities. Wide masking usage was inapplicable before Knights Landing because of the absence of instruction set and corresponding micro-architecture support in other AVX-compatible platforms.

Wide masking usage improves average code performance but imposes new optimization trade-offs. For example, aggressive mask usage for codes with statically unknown low trip counts may lead to a situation when up to 90% of SIMD lanes are masked out. Assisting the compiler by providing additional information about real trip counts and optimal VL becomes even more important on AVX-512.

For ineffective peeled/remainder loop performance issues on Knights Landing and other AVX-512 platforms, the Vectorization Advisor tends to display a *Specify Loop Count* or *Specify Vector Length* recommendation instead of a manual *Add data*

padding recommendation. To better estimate the performance impact, the AVX-512 **Recommendations** variant additionally takes into account:

- Dynamic mask utilization ratio (can be obtained in a special mode of the **Trip Counts** collector)
- Number of masked load/store instructions versus the number of computational instructions
- AVX-512-specific alignment analysis (out of scope for this chapter)

Speedups with approximate reciprocal, reciprocal square root, and exponent/mantissa extraction

Square root and division operations are very common in HPC workloads. However, even with the presence of hardware-level support, division (DIV) and square root (SQRT) operations remain a performance bottleneck, and can be several orders of magnitude slower (in terms of throughput and latency) than multiplication operations. Knights Landing attacks the problem from two perspectives:

- Improve the characteristics of packed division and square root instructions.
- Introduce higher-accuracy approximate reciprocal and reciprocal square root instructions (often used in conjunction with Newton-Raphson method).

The table in Fig. 10.5 shows vector instructions available on Knights Landing, in order from higher accuracy to lower accuracy, for math algorithms with double-precision divisions and square root computations.

Scalar forms of divisions and square roots are also available in AVX-512. However, we recommend vectorizing codes with divide and/or square root operations, because vector variants of DIV/SQRT are roughly as fast (in terms of latency/throughput) as their scalar variants, while being capable of processing $8\times/16\times$ more data at the same time.

The Vectorization Advisor **Recommendations** feature detects the following subcases and provides the following advice:

- Scalar divisions and square root are detected in potentially vectorizable loops.

  Advisor Recommendation: *Scalar SQRT/DIV instructions detected. AVX-512 vectorization may result in speedups. Consider enabling explicit vectorization.* Case-by-case, the Vectorization Advisor may refer to specific steps required to safely vectorize the code.
- `-xCOMMON-AVX512` compiler option used when compiling reciprocal-intensive codes run on Knights Landing.

  Advisor Recommendation: Consider recompiling your application using –xMIC-AVX512 or –axMIC-AVX512 to enable higher-accuracy reciprocal and reciprocal square root instructions. Case-by-case, the Vectorization Advisor may additionally analyze the floating-point-model accuracy setting and `-no-prec-div`, `–fp-model` or `-fimf-precision` compiler option usage.

ISA Subset	Vector Instructions	Precision in bits (PD variant)	Performance Notes	Description
AVX-512 Foundation	VDIVPS/D, VSQRTPS/D	53	10×+ smaller throughput (i.e., 10×+ "slower") than AVX-512 multiplication	Accurate divisions/sqrts
AVX-512ER (Xeon Phi-only AVX-512 extension)	VRCP28PS/D, VRSQRT28PS/D	28	~2× smaller throughput than AVX-512 multiplication (i.e., 5×+ faster than corresponding division/sqrt instructions)	High accuracy approximate reciprocals Available in single-precision and double-precision forms
AVX-512 Foundation	VRCP14PS/D, VRSQRT14PS/D	14		Approximate reciprocals Available in single-precision and double-precision forms
AVX, AVX2	VDIVPS/D, VSQRTPS/D	53		Provided on older AVX/AVX2 platforms
AVX, AVX2	VRCP, VRSQRT	12		No double-precision variants available

FIG. 10.5

Performance and precision of square root, division, approximate reciprocal, and approximate reciprocal square root instructions on AVX, AVX2, and AVX-512.

- Double-precision AVX code demanded reciprocal usage (depending on the floating-point-model compiler option and DIV/SQRT instructions usage) while running on AVX-512 and especially on AVX-512 platform.

  Advisor Recommendation: Your code may benefit from double-precision approximate reciprocal instructions replacing slower divisions or square root computations. Double-precision reciprocals are available only on the AVX-512 platform used to run your non-AVX-512 application. Consider recompiling your application for AVX-512 using –(a)xCOMMON-AVX512 or –(a)xMIC-AVX512 to enable double-precision, reciprocal, and reciprocal square root instructions. While this sub-case should be rare, the resulting theoretical speedups could be as great as 20× per loop.

Inefficient memory access in assumed-shape array and AVX-512 gather/scatter

Inefficient memory access in assumed-shape array is a rare example of a Fortran-only Vectorization Advisor issue/recommendation.

The issue is normally detected in vectorized loops that use Fortran arrays with unknown offset size between elements (i.e., stride) that could be passed in runtime. These assumed-shape arrays were introduced in the Fortran 90 standard and have become increasingly popular.

Consider the example in Fig. 10.6. The actual parameters passed to the `assumed_shape_test` sub-routine could be either contiguous (unit-stride, see commented-out `a_unit` and `b_unit`) or non-contiguous (in our example actual `a` and `b` are non-contiguous sections with stride-4). The `assumed_shape_test` vector code generation can be optimized more effectively if it is known in advance that A and B are accessing memory in a linear manner; otherwise AVX-512 vectorization must be performed using expensive gather/scatter instructions.

For code similar to that in Fig. 10.6, the speedup of the linear access/optimized version over a gather/scatter-based variant could be as great as 3×–6× for AVX-512 and up to 8× for AVX2. For SSE/AVX/AVX2 platforms, nonlinear-access versions of loops with assumed-shape arrays are often unprofitable to vectorize.

With AVX-512, code with irregular memory access becomes easier and more profitable to vectorize because of scatter instruction introduction and overall gather/scatter u-arch improvements. However, the gap between linear-access- and nonlinear-access-optimized versions remains huge on AVX-512.

Unfortunately, it is unknown at compile time (at least when compiling cases like B.f90) if the actual access pattern in A and B is linear or not, therefore vectorization

```
!                A.f90
  real, target, allocatable, dimension(:,:) :: x
  real, pointer, dimension(:) :: a
!                somewhere in A.f90
    a => x(1:n:4, nl)
    b => y(1:n:4, nl)
    call assumed_shape_test(a, b)


!    contiguous actual parameters
!    a_unit => x(1:n, nl)
!    b_unit => x(1:n, nl)
!    b_unit => x(1:n, nl)
!    call assumed_shape_test(a, b)

!                compilation unit B.f90
  subroutine assumed_shape_test(A, B)
  real, intent(out), dimension(:) :: A
  real, intent(in), dimension(:) :: B

! This is a loop which is normally auto-vectorized
  A = B + 1
  return
end subroutine assumed_shape_test
```

FIG. 10.6

1D assumed-shape array.

must be done for the worst-case scenario using the general-purpose gather/scatter-based variant. Still, to avoid an undesirable slowdown, the compiler sometimes generates *multi-versioned code* with the following versions:

- *Optimistic* version vectorized such that all assumed-shape arrays are accessed as if they are unit-stride.
- *Pessimistic* (default) version vectorized such that all assumed-shape arrays are accessed as if they are non-unit-stride.

Every time an `assumed-shape` test function is invoked, a small runtime check is executed to decide which version to execute for a given set of parameters.

The multi-versioned approach is a powerful technique. However, it is not applicable for complex loops where tens and hundreds of array references are simultaneously used, because runtime checks would be too expensive. Also, even when multi-versioning is applicable, the presence of runtime checks and other miscellaneous computations could slow down the cumulative time spent in a multi-versioned loop.

Intel Advisor includes the *Ineffective memory access in assumed-shape array* issue/recommendation (Fig. 10.7). The algorithm behind this issue automatically checks for the following information:

- Presence or absence of compiler multi-versioning for assumed-shape arrays
  - In the absence of multi-versioning, the Vectorization Advisor leverages information about the usage of `assumed-shape` arrays in the loop
  - In the presence of multi-versioning, the Vectorization Advisor performs additional analysis to check which loop version was generated for linear access and for nonlinear access.
- Dynamic Access Pattern information (available only when the **MAP** feature is invoked)

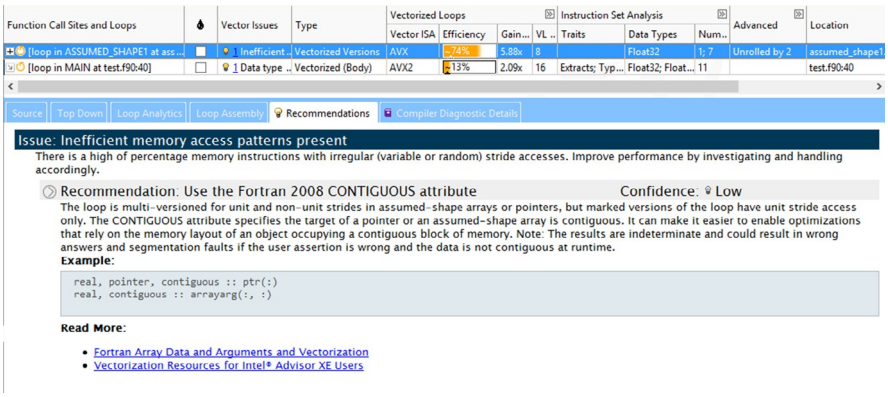


FIG. 10.7

Ineffective memory access in assumed-shape array.

- Miscellaneous information about cost/benefit trade-offs of using gather/scatter instructions in every loop

If data sources confirm linear access, the Vectorization Advisor conditionally recommends using the `CONTIGUOUS` attribute (introduced first in Fortran 2008) with an `assumed-shape` array to tell the compiler the data always occupies a contiguous block. The alternative Vectorization Advisor suggestion is replacing the `assumed-shape` array with an `assumed-size` or `adjustable-size` array. Vectorization Advisor also displays a warning: *The results could be indeterminate and could lead to wrong answers and segmentation faults if the user assertion is wrong and the data is not contiguous at some runtime workloads*.

To elaborate: Vectorization Advisor shows a *linear access-confirmed* vector issue in the following major sub-cases:

- If the loop is multi-versioned, but the nonlinear-access version was never executed.

  Advisor Recommendation: The loop is multi-versioned for unit and non-unit strides in assumed-shape arrays or pointers, but marked versions of the loop have unit stride access only. Use the CONTIGUOUS attribute which specifies the target of a pointer or an assumed-shape array is contiguous. It can make it easier to enable optimizations that rely on the memory layout of an object occupying a contiguous block of memory.

  Applying this recommendation may result in 10–40% speedups for the cases like our example in Fig. 10.6.
- If multi-versioning is not activated for loops with assumed-shape arrays, but **MAP** data confirms all assumed-shape accesses in the loop are unit-stride.

  Advisor Recommendation: The loop is using assumed-shape arrays or pointers, but marked versions of the loop have unit stride access only. Use CONTIGUOUS attribute or assumed-size/adjustable-size arrays to specify that the target of a pointer or an assumed-shape array is accessed contiguously.

  The resulting speedup could be as great as 3×–6× on AVX-512 platforms for the cases like our example in Fig. 10.6.
- If the loop is multi-versioned and both versions are executed, but the nonlinear-access version is only executed under specific conditions (e.g., on one particular path in a program call tree, let's say for the first phase of algorithm).

  The Vectorization Advisor could display a *low confidence Advisor Recommendation*: to explicitly distinguish/split loop and function implementations, so that the nonlinear-access version is differentiated from the linear-access version at compile time. The confidence level is low because such explicit multi-versioning is rarely more effective than compiler multi-versioning to split function or loop.

The algorithms described earlier are actually simplified forms of a real decision tree encapsulated within the **Recommendations** feature. This hidden complexity is often convenient for developers who choose to not worry about the version number for multi-versioned loops or the balance between `vinsert/vmove` and `vgather` instructions.

However, experienced developers can discover most of the corresponding data on their own by checking the **Advanced** columns in the **Survey Report** (to find out multi-versioning and unrolling information), **Type** and **Traits** columns, also in **Survey Assembly** (as shown in Fig. 10.8), or memory access pattern details in the **MAP** and **MAP Source Report**. This gives more flexibility to those who want to perform their own analysis without relying on higher-level product intelligence. The new **Loop Analytics** tab also offers low-level information for a loop, making deeper investigation more convenient.

Let's discuss what is special for AVX-512 when dealing with Fortran `assumed-shape` arrays. For the AVX-512 case, the loop versions with `nonlinear-access assumed-shape` arrays are vectorizable or profitably vectorizable more frequently than for the SSE/AVX/AVX2 case. As a result, more irregular-access loops with `assumed-shape` arrays become vectorizable and/or profitable to vectorize and thus *auto-vectorized* by the compiler by default. This means the average Fortran-code developer will face this *vector issue* more frequently.

More auto-vectorized loops in AVX-512-compiled code often lead to a performance increase, but they can also lead to an increased gap between default and explicitly vectorized (e.g., OpenMP 4.x) code versions. In other words, the code becomes faster, but speedup potential and low-hanging optimization opportunities become greater as well. This increases demand for more developer input, thereby requiring various kinds of performance analysis to make informed decisions about optimization techniques. This observation is equally applicable to various AVX-512 optimization techniques.

AVX-512 multi-versioning for `non-linear-access assumed-shape` arrays is implemented using gather/scatter AVX-512PF instructions, while SSE/AVX/AVX2 multi-versioning uses other ISA capabilities (like `vinsert/vextract` instructions). Optimization techniques applicable to gather/scatter code are slightly different than optimization techniques available for corresponding SSE/AVX instructions.

The *Ineffective memory access in assumed-shape array* issue/recommendation illuminates the importance of synergy between dynamic analysis data and advanced Intel compiler diagnostics (such as multi-versioning information) in a single place.

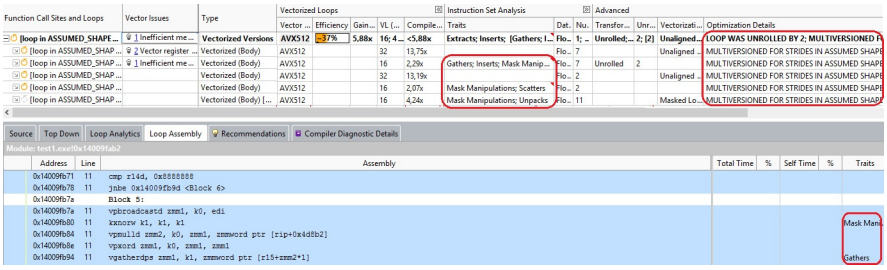


FIG. 10.8

Raw data in the **Survey Report** and **Survey Assembly/Source** to guide expert analysis similar to what **Recommendations** perform automatically.

It also highlights the productivity boost potential exposed by **Recommendations**, which automate and encapsulate complex performance analysis that otherwise requires deep exploration of multiple performance data types.

MAKING EXPERT USERS HAPPY: KNIGHTS LANDING-SPECIFIC TRAITS AND ISA ANALYSIS

AVX-512 exposes several new, unique instructions and corresponding unique performance optimization opportunities. Intel Advisor introduced new AVX-512-specific **Traits** for instructions whose presence or absence could significantly affect vector code performance (in a negative or positive way).

The table in Fig. 10.9 lists the most important AVX-512-specific **Traits** and comments on potential performance impact. All estimates in Fig. 10.9 are intentionally qualitative and approximate. The actual speedups depend on multiple factors,

Vectorization Advisor Trait and/or Recommendation	Area of Applicability	Theoretical Performance Impact Comments	Corresponding AVX-512 Instructions
Compress/Expand Trait and Recommendation	Vectorization of loops with conditional dense <-> sparse memory movement	>> **4×** speedup over scalar code	`v(p)expand*` `v(p)compress*`
Gather/Scatter Trait	Vectorization of loops with non-contiguous (non-unit-stride) memory access on AVX-512	**Up to 10× slower** than contiguous memory access for 16-wide SIMD parallelism. **>2× faster** than scalar code	`v(p)gather*` `v(p)scatter*`
Conflict Detection Trait	More effective vectorization of code with histogram patterns		`v(p)conflict*`

Approximate Reciprocals/Reciprocal SQRT; Approximate Reciprocal (SQRT) AVX-512ER Divisions, Square Roots Traits & Recommendations	Vectorization of all HPC codes dealing with divisions and square roots	**>10× faster** when replacing DIV/SQRT with appropriate VRCP/VRCSQRT	`vrcp*`
			`vrcsqrt*`
			`vdiv*`
			`vsqrt*`

Exponent extraction Mantissa extraction Traits	Various math algorithms, often in conjunction with reciprocal/reciprocal sqrt (Newton Rapson methods)		`vgetexp*` `vgetmant*`

L1 (L2) Prefetch L1 (L2) Sparse prefetch Trait	Improved throughput of bandwidth-bound codes on Knights Landingonly	Depends on memory access pattern, locality and memory subsystem characteristics	`prefetchw*`
			`vscatterpf*`
			`vgatherpf*`

FIG. 10.9

Traits summary.

including but not limited to used data types, compute versus memory subsystem balance, and branch-taken ratios for conditional codes.

Most newly introduced AVX-512 instructions (e.g., approximate AVX-512 reciprocals along with many other AVX-512 instructions) have vector and scalar forms. While in this chapter, we focus specifically on enabling AVX-512 vectorization, scalar (non-vectorized) codes also may greatly benefit from AVX-512. That's why Intel Advisor detects AVX-512 Traits not only in vector, but also in scalar codes, as shown in Fig. 10.10.

The rest of this section discusses three selected traits in more detail.

Compress/Expand Trait

The **Traits** feature checks for the presence of new AVX-512 `vpexpand` and `vpcompress` instructions (applicable to both integer and floating-point data types). These instructions enable effective vectorization of loops with conditional writes or reads, providing better performance than the more general-purpose gather/scatter instructions.

The sample code vectorized using compress, shown in Fig. 10.11, demonstrates:

- Reading all data elements from sparse memory in source array
- Checking some condition (in this case, checking whenever it is a positive value)
- Writing them into contiguous (compressed) memory area in `dest` array

The compress pattern can be applied in more general cases, where sparse source memory is conditionally stored into (compressed to) contiguous (dense) destination memory. The expand pattern can be applied where dense source memory is conditionally loaded into (expanded to) sparse destination memory. For a VL of 16

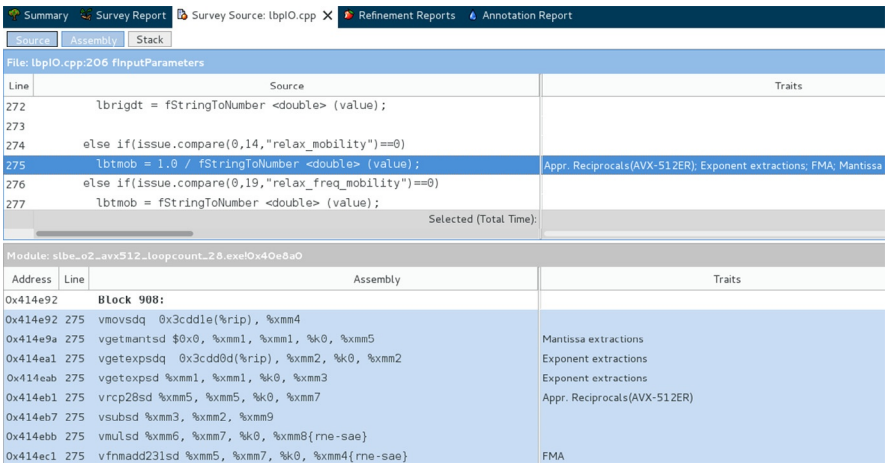


FIG. 10.10

Scalar AVX-512 traits: reciprocals and mantissa extractions.

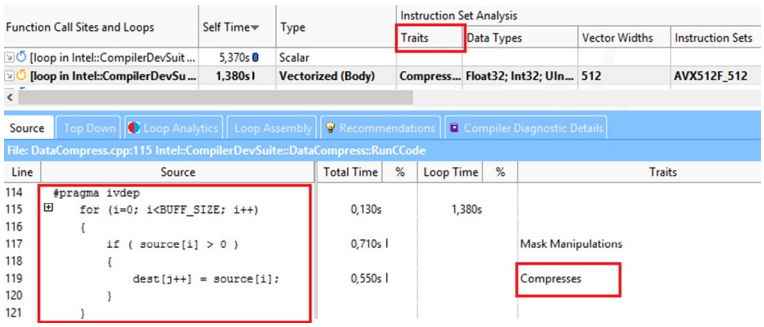


FIG. 10.11

AVX-512 analysis of code with compress pattern.

(32-bit integers on AVX-512), the speedup enabled by `vcompress/vexpand` instructions is 4×–10× greater than for scalar code.

Vectorizing code like that shown in Fig. 10.12 is also theoretically possible using older AVX/AVX2 shuffle/permute instructions. However, for vectorizing arrays with double-word elements, the size of the corresponding permutation mapping table would be as big as 2○22 bytes (about 4 Mb) *per single instruction* when dealing with 512-bit registers. That's why introducing specialized hardware-level support for compress/expand is critical to enable such patterns in AVX-512.

Compress/expand idioms are automatically recognized in AVX-512 code by the Intel compiler and thereby could be applied by the auto-vectorizer. However, in the case of assumed dependencies unrelated to compress/expand, we need to use `#pragma ivdep` or similar techniques to "enable" the auto-vectorizer capability of compress/expand pattern code generation. Note also that the `simd` directive is normally inapplicable to loops with compress/expand, but the landscape may change in future releases of these standards.

```
//Compress:
for (i)
if condition (i)
      dest [j++] = expression(src[i])
// Expand:
for (i)
if  condition (i)
      dest [i] = expression (src[j++])
```

FIG. 10.12

Compress/Expand examples.

The importance of compress/expand is difficult to overestimate; compress/expand-like code patterns can be seen in molecular dynamics, industrial DBMS, and compression and cryptography algorithms.

Gather/Scatter Traits

Gather/Scatter **Traits** highlight the presence of new AVX-512 instructions responsible for loading/storing data with irregular memory layout. The gather instruction was actually introduced in AVX2 (notice the Gather **Trait** even when profiling AVX2 code), but the AVX-512 variant of the gather instruction has more flexibility and also naturally becomes capable of dealing with 512-bit-wide, *ZMM* AVX-512 registers. The scatter instruction does not have an analog on AVX2, therefore the Scatter **Trait** is seen only in AVX-512 code.

On Knights Landing, the gather instruction may have up to one order of magnitude smaller throughput than a contiguous vector load of 16 elements. While gather/scatter-based vectorized code is still faster than its scalar counterparts, it is still wise to look for opportunities to avoid purely irregular memory access patterns (and therefore avoid gather/scatter usage) by optimizing memory access or explicitly communicating the presence of regular accesses to the compiler where applicable.

The Scatter and Gather **Traits** only highlight instruction presence, but do not provide exhaustive, detailed information about gather and scatter; instead they aim to quickly characterize vectorized code and motivate developers to run deeper-dive gather profiling available with Advisor Memory Access Pattern feature.

Conflict(−free) subset detection Trait

The Conflict Detection **Trait** detects usage of a special ISA subset called Conflict (−free) Detection Instructions (AVX-512CD). While these instructions do not belong to AVX-512F, they are supported by both Knights Landing and Intel Xeon processors. The purpose of this instruction subset is to enable effective vectorization of histogram code patterns.

Consider the example in Fig. 10.13. The naïve way of auto-vectorizing by the compiler relies on gather and scatter instructions usage. However, in the case when index[*i*] contains duplicated entries (e.g., index[2] == index[3] == 0), the gather/scatter-based vectorized version of this loop produces wrong results (because dest[0] will be updated only once).

This kind of code pattern is often called a histogram and is frequently seen in various areas of discrete math, such as graph processing, and more generally in arbitrary code actively dealing with indirect referencing. With the introduction of the

```
For (i)
  if(condition(i))
    dest[index[i]]++
```

FIG. 10.13

Histogram pattern example.

v(p)conflict instruction and AVX-512CD vectorizing, such algorithms are now supported at the hardware level. AVX-512CD usage is highlighted in both the **Traits** and **Instruction Sets** columns of the **Survey Report**.

MEMORY ACCESS PATTERN REPORT

An initial **Survey** analysis of hot loops often identifies inefficient memory access patterns as a main bottleneck. Memory access patterns issues are the toughest and most frequent performance problem in code not yet modernized for vector SIMD parallelism.

Applying *straightforward* SIMD and threading optimizations often does not provide desirable speedups because some parts of applications (including vectorized hot loops) become *memory bound*. Memory-access-patterns-bound code is just one subtype of a larger memory-bound class of problems, along with *memory-bandwidth-bound* and partially overlapping with *memory-latency-bound* sub-types.

The deeper-dive **Memory Access Patterns** (MAP) analysis in the Vectorization Advisor dynamically tracks memory access in selected parts of scalar and vector code. The resulting **MAP Report** and **MAP Source Report** provide:

- **Aggregated stride distribution** information for every profiled loop. Use as a high-level indicator of memory-access-pattern-bound code.
- **Detailed stride information** for every memory instruction or source line. Identifies if data is accessed in a unit stride (contiguous), constant stride, or irregular access (including gather/scatter) manner.
- **Memory footprint** characteristics (with cache line-number estimates). Provide a powerful and efficient way to quickly identify spatial and some of temporal memory locality bottlenecks, such as parts of the code that access a large amount of memory, leading to multiple cache misses or even memory page faults.
- **Variable references** integration. Makes it possible to map all the access pattern and locality data to specific locations in source code or addresses in assembly, as well as directly to the exact variables and data types that produced the memory access performance observation.
- **MAP Recommendations**. Classifies memory-bound codes and provides detailed advice on specific optimization techniques to improve performance and overcome a specific sub-type of memory *limits*.

The most common examples of Vectorization Advisor MAP **Recommendations** are:

- Use Array-of-Structure to Structure-of-Array (AoS → SoA) code transformation to fix memory-access-patterns-bound code demonstrating c*onstant stride*.
- Use Array-of-Structure-of-Arrays code transformation to fix memory-access-patterns-bound code that simultaneously demonstrates memory-locality issues.
- For code compiled using the 2016 Update 1 version of an Intel compiler: Use the SDLT library, which provides a unique capability to speed up

access-pattern-bound code without using expensive AoS → SoA-like, data-layout refactoring. See Chapter 11 to learn more about SDLT-based, data-layout transformations for C++.

- Better use new high bandwidth memory (MCDRAM, see Chapter 3) capabilities, introduced in Knights Landing, for codes with big enough footprint (indicating bad spatial or temporal locality), but having unit-stride memory access patterns (therefore bandwidth bound).

These recommendations (except those related to MCDRAM) are equally applicable to Intel Core, Intel Xeon, and Knights Landing. Additionally, new in the Intel Advisor 2017, there is an *AVX-512-specific* gather/scatter profiling capability (discussed in the next section).

AVX-512 GATHER/SCATTER PROFILER

Knights Landing introduces an AVX-512 `v(p)gather` instruction that normally provides better effectiveness and wider applicability/flexibility than `v(p)gather` instructions in AVX2 or Knights Corner (which is IMCI ISA based). AVX-512 gather and scatter support various combinations of index versus offset versus vector width, and introduces an explicit mask argument. Fig. 10.14 provides a typical example of `vgather` instruction operands and corresponding Intel Intrinsic function syntax.

However, as highlighted in previous sections, AVX-512 code utilizing `v(p)gather` (and newly introduced `v(p)scatter`) instructions still demonstrate substantially worse performance than similar code using contiguous vector data load/store. While

Instruction example (MASM syntax):

```
vgatherqpd zmm10, k2, zmmword ptr [r15+zmm0*1]
```

Corresponding intrinsic function example:

```
__m512d _mm512_mask_i64gather_pd (__m512d src, __mmask8 k, __m512i
vindex, void const* base_addr, int scale)
```

Operand explanation:

- zmm10 – **destination** 512-bit wide register, corresponds to return value of intrinsic function
- k2 – mask register (**write mask**) – corresponds to k parameter in intrinsic
- zmmword ptr [r15+zmm0*1] – **source** being gathered to dst. Superposition of base_addr, vindex (offsets index) and scale

FIG. 10.14

vgatherqpd instruction and corresponding Intel Intrinsic function.

gather/scatter–based vectorized code is faster than its scalar (or AVX2/IMCI) counterpart, it is still wise to look for opportunities to improve or avoid it.

In certain cases, it is possible to improve the performance of gather/scatter vectorized code by replacing gather/scatter instructions with sequences of linear load/store (or more complex memory shuffle/permute manipulations) instructions. From a programmer perspective, it is achievable by adjusting an algorithm and optimizing memory access, explicitly communicating the presence of regular accesses to the compiler, or falling back to intrinsics usage.

Intel Advisor introduces a **Gather/Scatter Profiler** with a dedicated **Gather/Scatter Report** and **Recommendations** (see Fig. 10.15) to assist with gather/scatter versus shuffle/load explicit vectorization choices. The **Gather/Scatter Profiler**:

- detects cases where actual data layout is not truly random;
- guides code transformation to avoid or optimize gather/scatter usage; and
- helps to better characterize vectorization efficiency and hardware utilization of sparse memory access codes, even when there is no room for performance improvements.

The compiler normally generates AVX-512 `v(p)gather` and `v(p)scatter` instructions in the following cases:

1. *Unrecognized regular pattern*. Data is accessed with a unit-stride, uniform, or constant stride pattern, but the compiler cannot detect or prove the data is accessed in a regular manner. This usually happens when index array values (such as `array_reference[index[i]]` or `array_reference[index_function(i)]`) are unknown at compile time. The compiler assumes the worst (random) access pattern, while, in fact, index values have a regular structure. Assumed-shape-arrays processing (discussed in the *Digging for AVX-512 speedups with Vectorization Advisor Recommendations* section) is another common example.

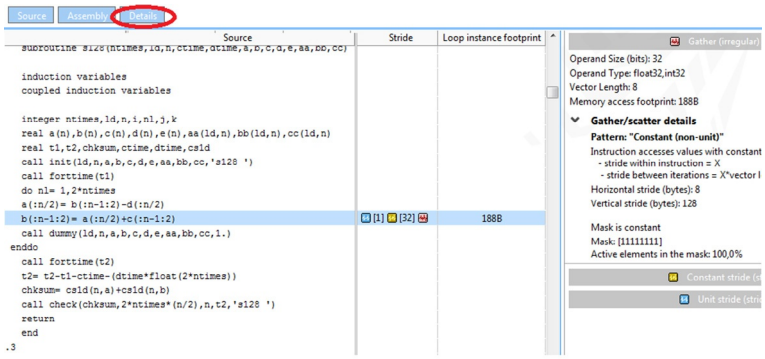


FIG. 10.15

Gather/Scatter Report.

2. *Regular constant stride*. Data is accessed using constant stride with a relatively large stride value. In most cases, the AVX-512 compiler uses gather/scatter instructions to vectorize code of this nature, although in a few special corner cases (especially for small stride values), a modern vectorizing compiler uses an alternate non-gather sequence of instructions.
3. *Truly irregular (or too complex) access pattern*. Data in a vectorized loop is accessed in a completely random (unpredictable) manner. In such cases, gather/scatter-based implementation is often the only viable choice for vectorizing code. Here, gather/scatter cannot be replaced with an implementation based on contiguous packed load/store without complete refactoring of code, such as replacing sparse 2D arrays with *compressed-sparse-row* data representation. Code where the index value is computed using complex nonlinear (e.g., transcendental) functions also fits into this category.

The **Gather/Scatter Profiler** automatically recognizes the cases in category #1 and #2, providing information about detected data-layout *meta-patterns* along with **Recommendations** about how to transform the code more efficiently. The Vectorization Advisor reports detected meta-patterns in the top part of **Gather/Scatter Report** (see Fig. 10.15). Figs. 10.16 and 10.17 summarize list of meta-patterns, corresponding vindex values, and Intel OpenMP syntax according to Gather/Scatter Advisor Recommendations.

In addition to meta-pattern detection and appropriate **Recommendations**, the **Gather/Scatter Report** provides detailed statistics about horizontal/vertical gather/scatter offset values and mask utilization.

The mask utilization information is relevant for AVX-512 code dealing with multiple branches. For example, when detecting extremely low mask utilization for gather/scatter in branch_A and extremely high mask utilization for gather/scatter

Pattern #	Pattern Name	Horizontal Stride Value	Vertical Stride Value	Example of Corresponding *Fix(es)*
1	**Invariant**	0	0	OpenMP uniform clause, simd pragma/directive, refactoring
2	**Uniform (horizontal invariant)**	0	Arbitrary	OpenMP uniform clause, simd pragma/directive
3	**Vertical invariant**	Arbitrary	0	intel_simd_lane() "manual" privatization
4	**Unit**	1 or –1	|Vertical Stride| = Vector Length	OpenMP linear clause, simd pragma/directive
5	**Constant**	Constant = X	Constant = X*VectorLength	Subject for AoS → SoA transformation

FIG. 10.16

Meta Patterns and corresponding **Recommendations** identified by **Gather/Scatter Report**.

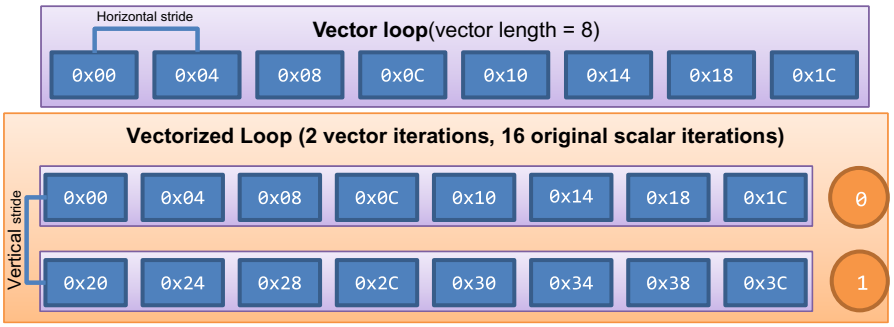


FIG. 10.17

Gather/scatter meta-patterns and horizontal/vertical stride correspondence.

in branch_B, we can communicate observed branch frequency to the compiler using `__builtin_expect` builtin; this may improve mask computation/predication strategies chosen by the compiler and lead to performance speedups.

Finally, for the *truly irregular* access pattern cases (in category #3 earlier), use the **Gather/Scatter Report** to guide replacement of multiple *adjacent* gathers with a more efficient single-gather-based implementation. Adjacent gather/scatter is out of scope for this chapter; there is more advanced information about the **Gather/Scatter Profiler** at the *Gather Scatter Blog* listed in the *For More Information* section at the end of this chapter.

MASK UTILIZATION AND FLOPS PROFILER

Counting FLOPs on Knights Landing is not supported by the hardware events, largely because there is no accounting for the values in mask registers when AVX-512 instructions are counted. New capabilities of the Advisor can make up for this lack of hardware support.

FLOP/s is a key way to measure efficiency of the workload or its individual loops or kernels. Measured FLOP/s can be compared against peak floating-point performance of target hardware (normally documented also as FLOP/s value).

> In this book, and especially in this chapter, we purposefully use "FLOPs" to refer to the number of floating point operations executed and "FLOP/s" to refer to the floating point operations per second. Elsewhere, it is not usual to see floating point operations per second abbreviated as FLOPs instead of FLOP/s. We need to talk both about FLOPs, and FLOP/s, in this chapter.

While masked computations enable much wider class of codes in terms of vectorization, they also complicate FLOPs accounting (and more generally performance

analysis), because traditional ways of measuring FLOPs or efficiencies cannot tell how many vector lanes were actually shut-off during the execution. For example, it is possible that code with significant FLOP/s value does not perform any computations at all, because an all-0 mask was used.

Intel Advisor introduces capability to precisely measure number of FLOPs and FLOP/s for user program as a whole, as well as for individual functions and loops. At the same time, it introduces **Mask Utilization Profiler** for AVX-512 (implemented in the same analysis type as FLOPs profiling). Altogether **FLOPs and Mask Profilers** make it possible to account both mask-aware FLOP/s, to see the number of effectively executed floating point operations, as well as traditional FLOP/s.

FLOP/s metric tells how many "useful computations" were executed in a given amount of time. It can be used as a good indication of optimization progress; successful vectorization should normally lead to FLOP/s metrics increase, especially when comparing a vectorized version to a scalar one. On the other hand, lower FLOP/s values are often an indication of significant latencies and overall performance bottlenecks.

FLOP/s analysis provides unique insight into the performance landscape, but it still has various limitations:

- Workloads with low mask register utilization may have over counted the FLOPs value. This is addressed in Advisor by implementing *mask-aware* FLOP/s metric. Be attentive to loops/functions with low mask-aware FLOP/s count, but with a high traditional FLOP/s value; these codes are often good candidates for simplifying control flow (via loops splitting, loop padding) or low-hanging code tuning using compiler hints (#pragma loop count, __builtin_expect).
- "More operations" does not always mean "more useful work." For example, in order to vectorize traditional SIMD reduction loops, compilers often have to do some special post-processing, which could be seen as expensive vectorization overhead. Various kinds of vectorization overheads could make the whole vectorization non-profitable, while the total number of corresponding FLOP/s will always be bigger than FLOP/s for original scalar loop. That's why in order to estimate profitability and speed-ups of highly vectorized codes, Advisor provides special mask-aware *Efficiency* metric, which characterizes quality of vector code generation done by the Intel Compiler. In order to estimate "Efficiency," Advisor enriches static compiler SIMD cost and benefit estimates with its own dynamic knowledge of trip counts, mask utilization, and memory accesses, providing unique hybrid static+dynamic SIMD code efficiency characteristics (see Fig. 10.18). It is highly recommended to analyze ***both*** efficiency and FLOP/s metrics, when estimating optimization progress for highly vectorized codes.
- Workloads with computationally intensive integer data processing will have low FLOP/s, although naturally it does not mean that the code or hardware does not perform well. Typical FLOPs analysis is only applicable to FP-centric HPC codes.

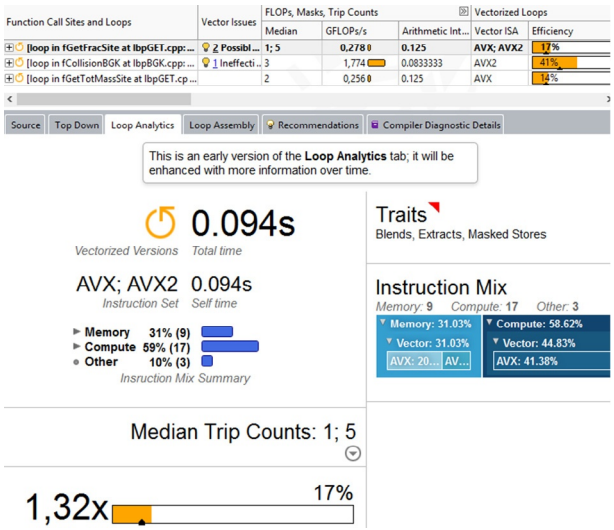


FIG. 10.18

Vectorization efficiency and FLOP/s in Survey Report and Loop Analytics.

Due to aforementioned and some other limitations, FLOP/s analysis still should be considered as just a single part of the bigger performance characterization picture provided by various components of Advisor, such as CPU time profile, Mask-aware **FLOPs Report**, **Vector Efficiency**, **Memory Access Pattern**, and **Roofline Analysis Graph**. The combination of these components enables deep understanding of the balances between memory- and compute-aspects of user code, at the same time providing a solid foundation for informed optimization advice provided to end-users in Advisor Recommendations feature.

The **FLOPs and Mask Profiler** is a relatively new feature, and is implemented as integral part of Advisor Trip Counts Profiler at the time we are writing this book. Since we anticipate evolving this support, please refer to our *FLOPS Mask Blog* to learn more on how to enable FLOPs and Masks profiling in the most recent versions of Intel Advisor 2017. The link for the blog can be found in *For More Information* at the end of this chapter.

ADVISOR ROOFLINE REPORT

Performance tuning experts talk about creating a *roofline model* for an application. In short, the concept is to decide if an application is compute bound or memory bound, and then compare the performance to the appropriate maximum FLOP/s or GB/s capabilities of the hardware. Before the Advisor Roofline Report, such a model would be a paper exercise based on a number of measurements taken manually.

The idea behind a roofline model is to estimate the "will not exceed" performance of an application assuming no radical changes to the fundamentals of how many computations and data accesses the application needs to make. This in turn gives us an idea of how close an application is to being fully optimized. Because, as we will see, a roofline model does not consider all factors that limit performance, there is no guarantee that an application can reach such levels. Nevertheless, roofline models can be very valuable input to us as we optimize. Of course, we should never forget that a different algorithm that requires less computation, or a radically different amount of data consumed, can be the best way to optimize. Such algorithm insights may be inspired during a roofline analysis — but they are not what a roofline model targets.

The Advisor **MAP Report** and **Gather/Scatter Analysis** aim to help with codes simultaneously affected by lack of vectorization (therefore "compute-bound") and lack of memory layout optimization (therefore "memory-bound").

Modernizing HPC codes is very often a process of balancing memory-bound and compute-bound aspects of the same application, requiring both optimizing for memory and for compute. However, memory performance analysis/optimization (via cache blocking, pre-fetching, AoSoA-like transformations) and compute optimization (vectorization, threading, or traditional serial hotspots optimization) are often done separately and rarely analyzed in an integral way.

To fill this gap between compute- and memory-optimization, performance experts traditionally introduce the notion of "computational density" or alternatively "arithmetic intensity" (*AI*). AI is defined as the ratio of total FLOPs divided by the amount of total bytes transferred for given part of user code to/from DRAM or to/from a cache sub-system level. We recommend focussing on memory optimization for codes with "low" AI, and on compute optimization (see also Chapter 14).

However, AI-based code characterization is not suitable to guide the optimization process or to identify specific boundaries preventing code from achieving desirable speed-ups. A **roofline model** supplements AI-based analysis with a dynamic FLOP/s profile and peak FLOPs and memory sub-system throughput levels (see Fig. 10.19), providing enlightening "bounds and bottlenecks" analysis for complex workloads. A roofline model finds its application in various areas of HPC codes performance characterization and future platform performance projections. But in the context of SIMD versus Memory analysis, a roofline model excels as a user-friendly actionable way to identify:

- what currently limits the performance of every part of the code?,
- which kind of optimization should be applied to which parts of the code first?, and
- what is the maximum speed-up achievable after applying identified optimizations?

Intel Advisor introduces a preview implementation of an automatic Roofline Model to help users decide between SIMD-focussed and Memory-focussed optimization. On the one hand, Roofline models can be seen as a user-friendly way to combine and generalize information provided by all main analysis types provided by

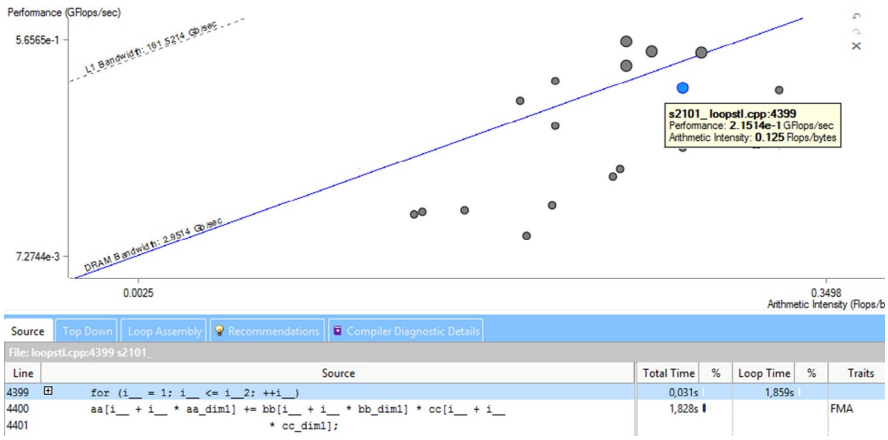


FIG. 10.19

Roofline model for an LCD benchmark application in Intel Advisor Roofline report.

Advisor, putting Mask-aware, FLOPs, Hotspots, and deeper dive Memory Analysis data all together. On the other hand, the line segments connecting the Roofline point (corresponding to particular loops or kernels) and the closest roof can be seen as a visualization of Advisor's Recommendations "Impact" or "Vectorization Gain" values.

After running an advanced version of Trip Count analysis, Advisor can generate a so-called "Cache-Aware roofline model" which extends the traditional Roofline with representations of cache hierarchy and provides a foundation for the most-general-purpose analysis of different levels of memory hierarchy. Each point on the Advisor Roofline corresponds to specific region in a user application, e.g., to regions with non-zero "exclusive" FLOPs value and non-zero CPU Self-Time.

Concepts and the implementation of the **Roofline Report** in Advisor will certainly continue to evolve. We encourage you to look for new roofline developments and user guides at our *Roofline Blog* listed in the *For More Information* section at the end of this chapter.

EXPLORE AVX-512 CODE CHARACTERISTICS WITHOUT AVX-512 HARDWARE

Thus far, we have focussed on using Intel Advisor when directly profiling an application on Knights Landing. However, the Intel Advisor can also analyze and compare AVX-512 and Knights Landing-specific code-generation without physical access to Knights Landing hardware (on an Intel Core-based laptop for instance). This could be useful when creating AVX-512 and/or Knights Landing code in

environments where hardware with AVX-512 support is not yet widely available. (We still recommend choosing native profiling on real AVX-512 hardware whenever possible.)

This feature provides support for analyzing non-executed code paths generated by the Intel compiler for various ISA. This functionality only works when using the Intel compiler `-axcode` (`ax*`-) compiler options and is only applicable to the **Survey Report**. The *For More Information* section at the end of this chapter has additional recommended readings to learn more about all the –axcode compilation options.

When enabled, this functionality complements the **Survey Report** with static analysis data and compiler diagnostics for the non-executed loops ↺, including:

- **Vector ISA** column, describes which ISA the code path targets;
- **VL** column, the *main* AVX-512 loop body normally has a VL twice as large as the *main* AVX2 loop body; and
- **Compiler Estimated Gain** column, shows compiler performance predictions. The estimated gain is compared to the scalar version of the same loop running on the hardware supporting the target ISA. For example, for the first loop version in Fig. 10.20, the compiler estimates a 6.05 × speedup for the vectorized version of the loop against its scalar version when both target the same AVX2-enabled machine. For the third loop from the bottom of Fig. 10.20, the compiler predicts a 13.29 × speedup for the vectorized version of the loop against its scalar version when both target the same AVX-512 machine.

Use the deeper-dive static data in the **Instruction Set Analysis** column to compare loop versions; for example, in terms of AVX2 versus AVX-512 **Traits**. A more detailed feature description can be found in an online article *Explore Intel® AVX-512 Code Paths* (see the *For More Information* section at the end of this chapter).

With the help of `-axcode` loop version analysis in the Intel Advisor, we can:

- generate and simultaneously analyze, on a single machine, code that targets multiple ISAs;
- make AVX-512 performance predictions based on compiler optimization details; and

Function Call Sites and Loops	Type	Vectorized Loops				
		Vector ISA	Efficiency	Gain ...	VL ..	Compiler Estimated Gain
[loop in s125_ at loopstl.cpp:942]	Vectorized Versions	AVX2; [AVX512]	~100%	13,54x	8; ...	13,54x
[loop in s128_ at loopstl.cpp:1131]	Vectorized Versions	AVX; [AVX; AVX...	~16%	2,54x	16; ...	2,54x
[loop in s2710_ at loopstl.cpp:39 ...	**Vectorized Versions**	**AVX2; [AVX512]**	**~32%**	**2,55x**	**8; ...**	**2,55x**
[loop in s2710_ at loopstl.cpp: ...	Peeled [Not Executed]					
[loop in s2710_ at loopstl.cpp: ...	Remainder [Not Executed]					
[loop in s2710_ at loopstl.cpp: ...	Vectorized (Body)	AVX2			8	2,55x
[loop in s2710_ at loopstl.cpp: ...	Vectorized (Body) [Not Executed]	AVX512			16	4,51x
[loop in s2710_ at loopstl.cpp: ...	Vectorized (Peeled) [Not Executed]	AVX512			16	2,01x
[loop in s2710_ at loopstl.cpp: ...	Vectorized (Remainder) [Not Executed]	AVX512			16	2,42x

FIG. 10.20

Survey Report for AVX2 (executed) and AVX-512 (not executed) loop versions.

- leverage ISA-specific static analysis information along with AVX-512 **Traits** and all other expert-oriented code characteristics without actually requiring AVX-512 hardware.

EXAMPLE — ANALYSIS OF A COMPUTATIONAL CHEMISTRY CODE

In this section, we share our experience performing an AVX-512 analysis and optimization of DL_MESO, a mesoscopic simulation package developed by research scientists at the STFC Daresbury Laboratory in the United Kingdom. It is not intended to be a complete case study. It highlights only the very first optimization techniques, applied with the guidance of the Vectorization Advisor, to inexpensively achieve a reasonable speedup on a Knights Landing platform.

The application domain for DL_MESO is computer-aided formulation of shampoos, detergent powders, agrochemicals, and petroleum additives. DL_MESO consists of several packages, including a C++ LBE package that simulates lattice-gas systems using lattice Boltzmann equation (LBE)-based fluids modeling. LBE performance profiles vary significantly depending on the input dataset characteristics, such as number of fluids, lattice models, or collision scheme type. In our Knights Landing experiments, we primarily used a D3Q15 lattice model with two fluids.

Developers of DL_MESO used the Vectorization Advisor in the Intel Advisor 2015 Beta to enable AVX vectorization in the LBE package. As described in *Get a Helping Hand from the Vectorization Advisor*, they applied a loop-padding technique and optimized for better memory access patterns. Their efforts resulted in up to a 3× speedup for their hottest kernels.

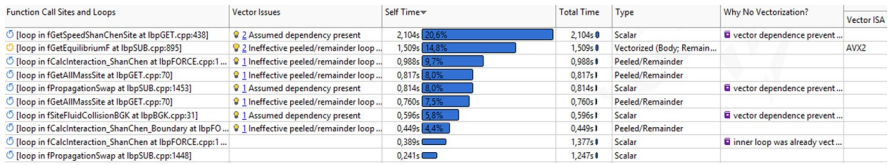
Here we focus on the next step: using the Vectorization Advisor to improve performance of AVX-512 vectorized kernels.

We first compiled the LBE package for the AVX2 ISA (using the `-xCORE-AVX2` compiler option in a 2016 version of an Intel C++ compiler) and ran it on Knights Landing. AVX2 is a predecessor instruction set of AVX-512, therefore we used the AVX2 version of DL_MESO_LBE *run on Knights Landing* as a performance *baseline* to compare with AVX-512 optimized versions.

This kind of comparison was impossible on Knights Corner because of instruction set incompatibilities. There is no such limitation with Knights Landing: applications compiled for Intel Core or Intel Xeon processors can run on Knights Landing (and future Intel Xeon processors), because AVX-512 is binary compatible with all predecessor instruction sets.

Fig. 10.21 shows the results of the **Survey** analysis of the AVX2 version profiled on Knights Landing. Notice only the loop in fGetEquilibriumF was vectorized.

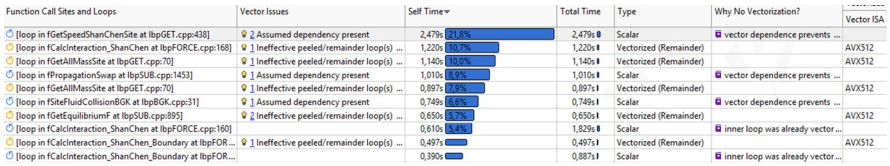
Fig. 10.22 shows the results of the **Survey** analysis after running LBE compiled for AVX-512 with the `–xMIC–AVX512` option, which enables AVX-512 Foundation (AVX-512F) and Knights Landing-specific extensions.



Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vector ISA
[loop in fGetSpeedShanChenSite at lbpGET.cpp:438]	2 Assumed dependency present	2,104s 20.6%	2,104s	Scalar	vector dependence prevent...	
[loop in fGetEquilibriumF at lbpSUB.cpp:895]	2 Ineffective peeled/remainder loop ...	1,509s 14.8%	1,509s	Vectorized (Body; Remain...		AVX2
[loop in fCalcInteraction_ShanChen at lbpFORCE.cpp:1 ...	1 Ineffective peeled/remainder loop ...	0,988s 9.7%	0,988s	Peeled/Remainder		
[loop in fGetAllMassSite at lbpGET.cpp:70]	1 Ineffective peeled/remainder loop ...	0,817s 8.0%	0,817s	Peeled/Remainder		
[loop in fPropagationSwap at lbpSUB.cpp:1453]	1 Assumed dependency present	0,814s 8.0%	0,814s	Scalar	vector dependence prevent...	
[loop in fGetAllMassSite at lbpGET.cpp:70]	1 Ineffective peeled/remainder loop ...	0,760s 7.4%	0,760s	Peeled/Remainder		
[loop in fSiteFluidCollisionBGK at lbpBGK.cpp:31]	1 Assumed dependency present	0,596s 5.8%	0,596s	Scalar	vector dependence prevent...	
[loop in fCalcInteraction_ShanChen_Boundary at lbpFO...	1 Ineffective peeled/remainder loop ...	0,449s 4.4%	0,449s	Peeled/Remainder		
[loop in fCalcInteraction_ShanChen at lbpFORCE.cpp:1 ...		0,389s	1,377s	Scalar	inner loop was already vect...	
[loop in fPropagationSwap at lbpSUB.cpp:1448]		0,241s	1,247s	Scalar		

FIG. 10.21

Survey Report for AVX2 performance baseline of DL_MESO LBE on Knights Landing.



Function Call Sites and Loops	Vector Issues	Self Time	Total Time	Type	Why No Vectorization?	Vector ISA
[loop in fGetSpeedShanChenSite at lbpGET.cpp:438]	2 Assumed dependency present	2,479s 21.8%	2,479s	Scalar	vector dependence prevents ...	
[loop in fCalcInteraction_ShanChen at lbpFORCE.cpp:168]	1 Ineffective peeled/remainder loop(s) ...	1,220s 10.8%	1,220s	Vectorized (Remainder)		AVX512
[loop in fGetAllMassSite at lbpGET.cpp:70]	1 Ineffective peeled/remainder loop(s) ...	1,140s 10.0%	1,140s	Vectorized (Remainder)		AVX512
[loop in fPropagationSwap at lbpSUB.cpp:1453]	1 Assumed dependency present	1,010s 8.9%	1,010s	Scalar	vector dependence prevents ...	
[loop in fGetAllMassSite at lbpGET.cpp:70]	1 Ineffective peeled/remainder loop(s) ...	0,897s 7.9%	0,897s	Vectorized (Remainder)		AVX512
[loop in fSiteFluidCollisionBGK at lbpBGK.cpp:31]	1 Assumed dependency present	0,749s 6.6%	0,749s	Scalar	vector dependence prevents ...	
[loop in fGetEquilibriumF at lbpSUB.cpp:895]	2 Ineffective peeled/remainder loop(s) ...	0,650s 5.7%	0,650s	Vectorized (Remainder)		AVX512
[loop in fCalcInteraction_ShanChen at lbpFORCE.cpp:160]		0,610s 5.4%	1,829s	Scalar	inner loop was already vector...	
[loop in fCalcInteraction_ShanChen_Boundary at lbpFOR...	1 Ineffective peeled/remainder loop(s) ...	0,497s	0,497s	Vectorized (Remainder)		AVX512
[loop in fCalcInteraction_ShanChen_Boundary at lbpFOR...		0,390s	0,887s	Scalar	inner loop was already vector...	

FIG. 10.22

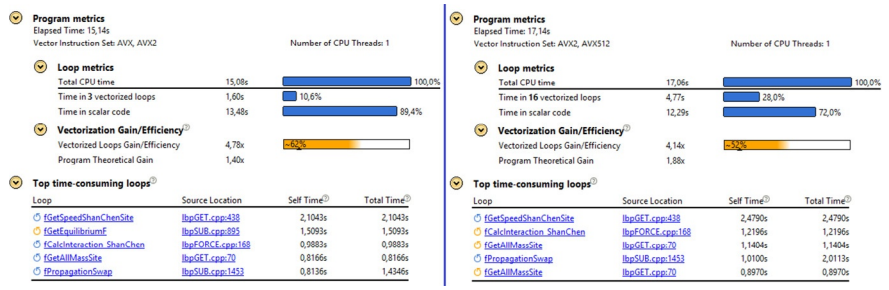
Survey Report for AVX-512 default code generation of DL_MESO LBE on Knights Landing.

We focussed mostly on the following loop hotspots in our AVX-512 analysis:

- Vectorized loop in fGetEquilibriumF (lbpSUB.cpp). It calculates equilibrium for compressible fluids and is responsible for 14.8% of baseline CPU time.
- Scalar (non-vectorized) remainder loop in fCalcInteraction_ShanChen (libFORCE.cpp). It is responsible for 9.7% of baseline CPU time.

Analyzing corresponding **Summary Reports** is one of the best ways to quickly compare AVX2 to AVX-512 performance, as shown in Fig. 10.23 (AVX2 on the left and AVX-512 on the right).

Fig. 10.23 shows that our initial naive AVX-512 version is even slower than the AVX2 version. To understand this phenomenon, note two potentially controversial **Summary** metrics:



Program metrics
Elapsed Time: 15,14s
Vector Instruction Set: AVX, AVX2
Number of CPU Threads: 1

Loop metrics

Total CPU time	15,08s	100,0%
Time in 3 vectorized loops	1,60s	10,6%
Time in scalar code	13,48s	89,4%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	4,78x	~62%
Program Theoretical Gain	1,40x	

Top time-consuming loops

Loop	Source Location	Self Time	Total Time
fGetSpeedShanChenSite	lbpGET.cpp:438	2,1043s	2,1043s
fGetEquilibriumF	lbpSUB.cpp:895	1,5093s	1,5093s
fCalcInteraction_ShanChen	lbpFORCE.cpp:168	0,9883s	0,9883s
fGetAllMassSite	lbpGET.cpp:70	0,8166s	0,8166s
fPropagationSwap	lbpSUB.cpp:1453	0,8136s	1,4346s

Program metrics
Elapsed Time: 17,14s
Vector Instruction Set: AVX2, AVX512
Number of CPU Threads: 1

Loop metrics

Total CPU time	17,06s	100,0%
Time in 16 vectorized loops	4,77s	28,0%
Time in scalar code	12,29s	72,0%

Vectorization Gain/Efficiency

Vectorized Loops Gain/Efficiency	4,14x	~52%
Program Theoretical Gain	1,88x	

Top time-consuming loops

Loop	Source Location	Self Time	Total Time
fGetSpeedShanChenSite	lbpGET.cpp:438	2,4790s	2,4790s
fCalcInteraction_ShanChen	lbpFORCE.cpp:168	1,2196s	1,2196s
fGetAllMassSite	lbpGET.cpp:70	1,1404s	1,1404s
fPropagationSwap	lbpSUB.cpp:1453	1,0100s	2,0113s
fGetAllMassSite	lbpGET.cpp:70	0,8970s	0,8970s

FIG. 10.23

Summary Report for AVX2 version (left side) and AVX-512 version (right side) of DL_MESO LBE code on Knights Landing.

- The number of vectorized loops increased in the AVX-512 version from 3 to 16, and the fraction of time spent in vectorized loops grew from 10.6% to 28.0%.
- However, the duration of the scalar part of the application decreased by only 1.2 s, which is smaller than the duration time added to the vectorized part (~3 s).

Our assumption: as the amount of vectorized code increased, the transformation of some kernels from scalar to vectorized possibly did not result in a speed-up of those kernels. This assumption was supported by the list of top time-consuming loops at the bottom of Fig. 10.23. At least two hot scalar loops for AVX2 (loops in fCalcInteraction ShanChen and fGetAllMassSite) were vectorized, but time spent in these vectorized AVX-512 loops was greater than time spent in their original AVX2 scalar versions.

We returned to the **Survey Report** to find out more about these two loops and other hot, newly vectorized, AVX-512 kernels. For convenience, we toggled the **Vectorized/Not Vectorized** switch off to focus on vectorized loops as shown in Fig. 10.24.

Focussing on vectorized AVX-512 loops in Fig. 10.25 resulted in another interesting discovery. All but one of the hot, vectorized loops followed an identical pattern: they all showed an *Ineffective peeled/remainder loop(s) present* **Vector Issue** and had a *Vectorized Remainder* **Type**. Also, all these loops contained masked computations (per data in the **Advanced** column).

We additionally toggled on the special **Non-executed loops** switch and clicked the **Expand All Loops With Nested Loops** control shown in Fig. 10.26. This enabled advanced representation of all loop versions generated by the compiler

FIG. 10.24

Vectorized and *Not Vectorized* **Survey** perspective switcher buttons.

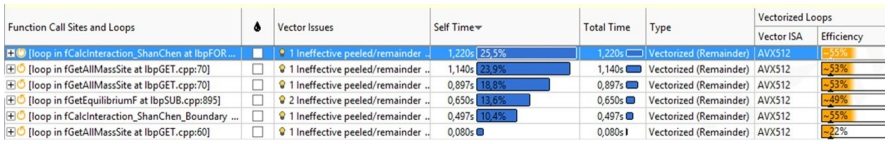


FIG. 10.25

Hot vectorized loops.

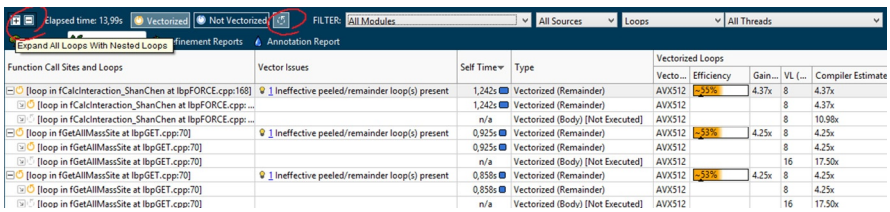


FIG. 10.26

Hot vectorized loops (remainder loops) with their non-executed counterparts (loop bodies).

and supplemented the list of executed loops with additional versions that were not executed in runtime.

Now we had all the data required to understand the root cause of our performance problem: 100% of vectorized loop execution time was spent in remainder loops. Think of remainder loops as unprofitable overhead. Unfortunately in our case, 100% of aggregated loop time was spent in unprofitable remainder loops, while the more profitable loop bodies in the vectorized loops were all marked *Not Executed*.

In addition, the **Compiler Estimated Gain** for remainder loops was more than 2× smaller than that for vectorized loop bodies. This means that by executing 100% of computations in remainder loops instead of loop bodies, we potentially lost even more than 2× potential vectorization gain.

Why were the loop bodies not executed? When we looked at the implementation of the loop in fCalcInteraction_ShanChen in Fig. 10.27, we noticed that it has a number of iterations equal to `lbsy.nf`, which physically corresponds to the number of modeled fluids. In typical DL_MESO computational chemistry experiments, `number of fluids` is ≤4 (6 is the recommended maximum), while in our workload it was actually equivalent to 2. The majority of other newly vectorized AVX-512 loops with an *Ineffective peeled/remainder loop(s) present* **Vector Issue** also had a number of iterations equal to `lbsy.nf`.

The optimal VL for an AVX-512 loop body processing large arrays of four-byte data types (int, float in C++) varies from 8 to 16. All loops with a fewer iterations must be executed in remainder loops. According to the **Survey Report**, these remainder loops also tended to have a VL of 8.

Executing remainder loops with a VL of 8 for a workload with only 2 iterations means the number of masked, executed iterations is equal to $8-2=6$. In other words, about 75% of the computations were wasted (suppressed by zero mask bits, but still spending some resources), while the other 25% that corresponded to *useful* iterations were executed less effectively. Bottom line: it is not surprising the vectorized version of the loop was slower than the scalar version. The same considerations applied to other workloads, because the number of fluids (up to 6) was always smaller than the optimal VL.

```
#pragma loop_count min(2), max(6), avg(4)
    for(int k=0; k<lbsy.nf; k++) {
        double psi = lbft[spos * lbsy.nf + k];
        factor0[k] += lbvwx[m] * psi;
        factor1[k] += lbvwy[m] * psi;
        factor2[k] += lbvwz[m] * psi;
    }
```

FIG. 10.27

Hot vectorized loop in fCalcInteraction_ShanChen using pragma loop_count.

What could we do to improve the situation? Should we just selectively disable vectorization?

In the Vectorization Advisor **Recommendations** tab, the first suggestion was to leave the decision up to compiler and just tell it the actual range of .nf values using `#pragma loop_count`.

In summary, we:

1. took a **Snapshot** of the default AVX-512 version, as suggested in the Fig. 10.1 workflow;
2. applied the loop_count pragma (as shown in Fig. 10.27) to all 13 newly vectorized loops. Applying this technique was inexpensive. We just provided min, max, and average values for the number of fluids normally modeled by physicists;
3. recompiled the code; and
4. re-ran the **Survey** analysis (and optionally the **Trip Counts** analysis).

As shown in Fig. 10.28, the new AVX-512 version was much faster than the previous (default code generation) version, as well as faster than the AVX2 baseline.

Let's summarize our learnings from this optimization exercise and formulate principles of peeled/remainder loop optimization for AVX-512 platforms.

On AVX-512 platforms, peeled and remainder loops are normally generated by the compiler using masked computations. This is beneficial, making it possible to perform something similar to automatic padding, avoid expensive conditional jumps, and increase utilized vector register length. Masked operations still have higher latency than unmasked ones. For that reason, vectorized loop bodies (where most computations normally belong) are always implemented without masking (using *k0* — special all-1 mask instruction form).

This strategy (masked body + unmasked remainder) works fine for loops with many iterations (a simple heuristic says *many* means at least 2 × more iterations than the optimal VL). However, not all hot loops have large trip counts and, more importantly, not all loop trip counts are known at compile time. As a result, the default strategy does not always lead to optimal performance and the amount of time spent in peeled/remainder loops could be high because of small trip counts unknown at compile time.

	Total Time	Speedup over Baseline	Time in Scalar Loops	Time in Vectorized Loops
AVX2 baseline	15.1 s	1x	13.4 s	1.7 s
AVX-512 default	17.1 s	0.88x	12.3 s	4.8 s
AVX-512 loop_count	10.5 s	1.44x	2.1 s	8.4 s

FIG. 10.28

Comparison of AVX2, AVX-512, and optimized AVX-512 versions.

There are two reasons why we should avoid too much time spent in remainder loops:

- Unlike loop bodies (which do not use masking, therefore have all SIMD lanes in SIMD registers doing useful work), vectorized remainder or peeled loops are typically generated using masks. As a result, those SIMD lanes that correspond to remaining iterations do not perform useful work. The amount of extra overhead is somewhat proportional to the number of remaining iterations and, to some extent, the chosen VL. Note: There are two compiler approaches for dealing with masked iterations within remainder loops. First approach: Split the remainder into two separate branches within the remainder loop (one branch for wasted iterations with the mask-0 bit value and another branch for iterations with the mask-1 bit value). Second approach: Mask only load/store instructions, while performing computational instructions unconditionally for all SIMD lanes including masked, that is, those with useless data. Both approaches introduce extra overhead somewhat proportional to the number of remaining elements (and that's why SIMD lanes suppressed by a zero mask bit still have some impact on execution time).
- Even in best-case scenarios when remainder loops and loop bodies have identical VL and there are zero fake iterations (consider a scalar loop with 16 iterations evenly distributed between a vectorized unmasked body and a masked remainder loop), the loop body still has higher vector efficiency. This is because unmasked operations have smaller latency than masked operations.

Now we can formulate the following optimization rule:

For loops with small trip counts, try to minimize the amount of time spent in vectorized remainder loops and increase the amount of time spent in loop bodies. The simplest way to do this: communicate the typical trip count values range to the compiler. For example: use `#pragma loop_count`.

The performance comparison of the AVX-512 DL_MESO version against the AVX2 version (Fig. 10.28) justified the importance of this rule. We tried to apply the rule to the loop in fGetEquilibriumF, which was the hottest vectorized loop in the AVX2 version and, after applying optimization to 13 nf-pattern loops, became the #1 vectorized hotspot again in the AVX-512 version.

The loop at fGetEquilibriumF (see Fig. 10.29) calculates the equilibrium distribution of compressible fluids. Depending on the number of fluids being modeled, this can be the hotspot with the most **Self Time**. The `lbv*` arrays store the velocities for the lattice in each dimension. The loop count variable `lbsy.nq` is the number of velocities (equal to 15 for the D3Q15 scheme). The resulting equilibrium is stored in the array `feq[i]`.

The 2016 version of the Intel compiler automatically vectorized the loop. Just like the loop in fCalcInteraction_ShanChen, this auto-vectorized version resulted in a

```
int fGetEquilibriumF(double *feq, double *v, double rho)
{
 double modv = v[0]*v[0] + v[1]*v[1] + v[2]*v[2];
 double uv;
 //#pragma loop_count 15
 for(int i=0; i<lbsy.nq; i++)
  {
   uv = lbvx[i] * v[0] + lbvy[i] * v[1] + lbvz[i] * v[2];
   feq[i] = rho * lbw[i] * (1 + 3.0 * uv + 4.5 * uv * uv -1.5 * modv);
  }  return 0;
}
```

FIG. 10.29

Code listing — loop for calculating equilibrium distribution.

non-executed loop body and spent all loop time in vectorized AVX-512 remainder loops. While it is clear why this happens for loops with a small trip count of 2 (as defined by the `lbsq.nf` variable), it is possibly surprising the same behavior occurs when trip counts are as large as 15.

After analyzing the breakdown of loop bodies, peeled loops, and remainder loops with the **Non-executed loops** switch toggled on (see Fig. 10.26), we realized the compiler automatically chose a loop body VL of 16. Therefore loop instances with a trip count value smaller than 16 executed completely in remainder loops. In our case, 15 iterations (which is smaller than 16) were split between two instances of vectorized remainder loops. The first eight iterations executed in the first remainder loop (with eight SIMD lanes, where all mask bits were equal to 1). The remaining seven iterations executed in the second remainder loop (where the first seven mask bits were equal to 1, and the last bit was set to 0 to suppress the non-existent *16th iteration*).

According to the rule formulated earlier, this is not optimal; we needed to minimize time spent in the remainder loop. The relatively low vector **Efficiency** (49%) also confirmed the need for local optimizations.

According to the Vectorization Advisor **Recommendation** for this loop (and, again, in line with our optimization rule), we needed to inform the compiler of the expected loop count range. After applying the fix (with the `loop_count` pragma) and re-surveying LBE, we made the following observations:

- Time spent in the loop decreased by ~20%.
- Part of the loop time was spent in the loop body, but another part still executed in the remainder loop. In both cases, the compiler used a VL of 8.
- This means the compiler changed the code generation strategy by choosing a VL of 8 for the loop body, executed the first eight iterations in the most effective (unmasked) vectorized loop body, and kept the remaining seven iterations in the remainder loop.

Our optimization rule paid off again: more iterations executed without using masked load/store instructions.

However, some computations were still done by masked remainder loops. We decided to check if we could completely avoid remainder loops by using *traditional* loop padding, as we did when optimizing loops for AVX on Knights Corner. After enabling loop padding, thereby increasing the loop trip count to 16 (adjusting the loop_count pragma accordingly), we re-ran the **Survey Report**. We observed the following:

- Very modest performance progress (only ~7% time decrease). This somewhat matched our expectations, because hardware-level support of masking partially eliminates the benefit from software-level padding.
- All loop iterations executed in the loop body, but the compiler still chose a VL of 8 instead of 16. This appears to be a sub-optimal choice worthy of redefinition with `#pragma omp simd simdlen (16)` (because 512-bit ZMM registers allow simultaneous processing of 16 elements); however the Vectorization Advisor Recommendation was not marked *High Confidence*. Deeper analysis of the Survey data indicated loop process types of different widths (integer lbv* velocities and double floating-point equilibrium arrays), thus the compiler chose a VL of 8 as a better trade-off between a wider floating-point double (8 bytes) data type and an integer (4 bytes) data type. Real measurements of the simdlen (16) optimized version confirmed this improvement did not lead to any additional speedup.

All the optimizations we did for the AVX-512 variant of DL_MESO can be best described as *low-hanging fruit*, in that we made no changes to the algorithm or data structures. All our changes were directly guided by Vectorization Advisor **Recommendations** and were supported by detailed information from the **Survey Report**. There was no need for expensive optimizations like memory access pattern tuning.

To be clear, expensive optimizations were unnecessary because they were done at the first stage of DL_MESO vectorization enabling. In addition, to get prepared for the AVX-512 Knights Landing platform in advance, Daresbury developers reworked most of the hot loops in terms of access patterns with guidance from Vectorization Advisor MAP data. More specifically, they replaced all instances of *array of structure* (AoS) for lbv* velocities coordinate arrays with *structure of array* (SoA) data layout.

This effort pays off where the hottest AVX2 and AVX-512 loops no longer struggled from non-unit stride low-efficiency vectorization and did not require gather/scatter instructions usage anymore. As a result, there was little memory access pattern optimization needed for LBE for this case study. That's why this chapter primarily focussed on trade-offs among loop padding, masking, and explicit vectorization for AVX-512.

Whenever code is already "prepared for" AVX-512 enabling (and thus only low-hanging fruit optimization is needed) or there is also a need for more expensive memory optimizations or refactoring, Advisor Survey, Recommendations, and MAP features assist in enabling effective AVX-512 usage.

SUMMARY

Intel (Vectorization) Advisor provides **AVX-512 analysis** capabilities guiding users in enabling the vectorization potential of Knights Landing for their applications. For scalar loops, Vectorization Advisor helps to discover what prevents code from being vectorized. For vectorized loops, it provides detailed AVX-512 performance characterization. In both cases, Advisor gives tips ("Recommendations") on how to vectorize the code or how to improve its vector parallel efficiency. Optimization insights delivered by user code-oriented Recommendations are additionally supplemented with low-level Advisor data such as AVX-512 Traits and FLOPs, masks, and Gather/Scatter Reports. The synergies between dynamic analysis data and advanced Intel compiler diagnostics in Advisor help to quickly identify area for "low-hanging" performance boost as we showcased in computational chemistry code case study.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this chapter.

- FLOPS Mask Blog, http://lotsofcores.com/advisor01.
- Gather Scatter Blog, http://lotsofcores.com/advisor02.
- Roofline Blog, http://lotsofcores.com/advisor03.
- Explore Intel® AVX-512 Code Paths, http://lotsofcores.com/explore512.
- Compiling for the Intel® Xeon Phi™ processor x200 and the Intel® AVX-512 ISA, http://lotsofcores.com/avx512knl.
- Intel® Compiler Options for …AVX-512… and processor-specific optimizations, http://lotsofcores.com/avx512options.
- Intel Advisor product page: https://software.intel.com/en-us/intel-advisor-xe.
- Intel Advisor Getting Started Guide: https://software.intel.com/en-us/get-started-with-advisor.
- Williams, S., Waterman, A. and Patterson, D. (2009) Roofline: an insightful visual performance model for multicore architectures. Communications of the ACM - A Direct Path to Dependable Software.
- Cache-aware roofline Model: http://dl.acm.org/citation.cfm?id=2776841.
- Explore Intel® AVX-512 Code Paths with Intel® Advisor XE while not Having Compatible Hardware, https://software.intel.com/en-us/node/607961.
- Fortran Array Data and Arguments and Vectorization (including assumed shape arrays topic), https://software.intel.com/en-us/articles/fortran-array-data-and-arguments-and-vectorization.
- Getting a Helping Hand from The Vectorization Advisor (Optimizing DL_MESO for AVX), https://software.intel.com/en-us/articles/get-a-helping-hand-from-the-vectorization-advisor.
- M.A. Seaton, R.L. Anderson, S. Metz and W. Smith, "DL_MESO: highly scalable mesoscale simulations", Mol. Sim. 39 (10), 796–821 (2013), doi:10.1080/08927022.2013.772297.

CHAPTER

Vectorization with SDLT 11

What is new with Knights Landing in this chapter?
AVX-512

This chapter introduces Intel® SIMD Data Layout Templates (SDLT) containers (used in place of `std::vector`). For C++ code, this can offer an effective method to achieve superior performance by increasing vectorization through "AOS to SOA or ASA" conversions. This conversion can enhance performance of Knights Landing or any processor. We present in this chapter, with sample codes, how to transition from Array of Structures (AOS) to Structure of Arrays (SOA) or Arrays of Structure of Arrays (ASA) by utilizing SDLT while maintaining a high-level object-oriented structure. We share the resulting improvements in vectorization (SIMD code generation) to illustrate how we obtained and measured the performance of SIMD code improving by 7× and 9× on Knights Landing and 5× and 2.2× on a 2.30 GHz Intel® Xeon® E5-2699 v3 processor (Haswell-EP) with SDLT for our examples and illustrated in Figs. 11.1 and 11.2.

WHAT IS SDLT?

SDLT is a C++11 template library providing generic containers that abstract their internal memory data layout. The abstraction allows for SIMD-friendly data layouts that differ from the normal C++ structure layout. The altered data layout is transparent allowing full use of "plain old data" structures and inlined methods inside of algorithms.

When we have data parallel algorithms in C++, and we are interested adding vectorization while still maintaining a high-level objected-oriented representation of our algorithm, we should consider utilizing SDLT. We can express our algorithms with "plain old data" structures by keeping them inside SDLT containers. We can transparently utilize an SIMD-friendly memory layout, such as SOA, without manually managing an array for each data member, without worrying about aligned data allocation, or providing alignment hints to the compiler. Because the transformation is transparent, the method and function arguments continue to be the same data types of the original structure.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00011-9

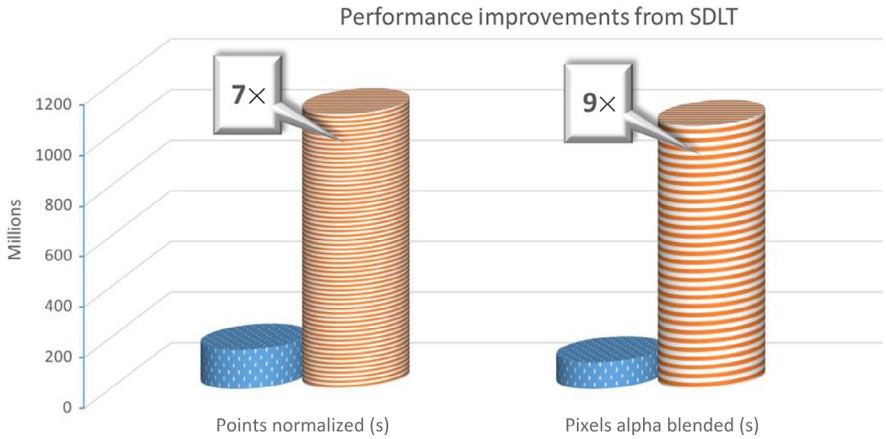


FIG. 11.1

Performance impact using SDLT on Knights Landing.

FIG. 11.2

Performance impact using SDLT on Haswell-EP.

GETTING STARTED

Fig. 11.3 shows a simple program, "Print Points," that assigns values to 128 3d points and prints their values. Although not computationally interesting, we will use it as our "Hello World" program in order to examine how to integrate SDLT to change the array of points to an SOA memory layout.

Fig. 11.4 shows our example modified to use SDLT. An accessor is used with the `[]` operator to import or export primitive objects to or from the container for a

```
#include <cstdio>

#define LEN 128

struct Point3s
{
  float x, y, z;
};

Point3s array[LEN];

void main()
{
  for (int i=0; i<LEN; i++) {
    Point3s p;
    p.x = 10000 + i;
    p.y = 20000 + i;
    p.z = 30000 + i;

    array[i] = p;
  }
  for (int i=0; i< LEN; i++) {
    Point3s p = array[i];
    printf(""%6.0f,%6.0f, %6.0f\n"",
           p.x,
           p.y,
           p.z);
  }
}
```

FIG. 11.3

Code for the original "Print Points."

```
#include <cstdio>
#include <sdlt/sdlt.h>
#define LEN 128

struct  Point3s
{
  float x, y, z;
};
SDLT_PRIMITIVE(Point3s, x, y, z)

sdlt::soa1d_container<Point3s> container(LEN);
auto array = container.access();

void main()
{
  for (int i=0; i<LEN; i++) {
    Point3s p;
    p.x = 10000 + i;
    p.y = 20000 + i;
    p.z = 30000 + i;

    array[i] = p;
  }
  for (int i=0; i< LEN; i++) {
    Point3s p = array[i];
    printf(""%6.0f,%6.0f, %6.0f\n"",
           p.x,
           p.y,
           p.z);
  }
}
```

Include SDLT types and macros

Declare structure to be a primitive, identifying its data members

Use a sdlt container instead of C array or std::vector

Obtain an accessor instead of a pointer

FIG. 11.4

Code for SDLT integrated "Print Points."

particular element. The data layout inside the SDLT container is abstracted. In this case it is an SOA. Note how the looping code still uses the `Point3s` despite the data in the container being a different memory layout. No other code changes were needed.

Methods cannot be called on primitives while inside a container. They must first be exported to a local variable. However, SDLT does provide the ability to access underlying data members without importing or exporting an entire primitive. The proxy objects returned by SDLT accessors provide methods matching the names of the data members which return a proxy that can import or export just that data member within the primitive. This can result in a few more code changes when integrating SDLT.

Fig. 11.5 shows the loops in "Print Points" changed to access data members directly inside the array instead of a local object. Fig. 11.6 enumerates the changes we had to make to integrate SDLT.

Because `printf` is variadic and takes any type as a parameter (similar to a templated parameter), it will not automatically export the value from the SDLT proxy. Instead, it would try and pass the proxy object itself. To handle these scenarios, we need to call the free function `unproxy` to export the value from a proxy.

The original version of the code provides a cleaner solution. When integrating SDLT, consider changing data member-based array accesses to operate on local primitive variables instead. However, when updating only a subset of data members of a primitive inside a container, better performance will be achieved by updating the individual data members versus the entire object.

SDLT BASICS

SDLT provides concepts of primitives, containers, accessors, proxy objects, offsets, and indexes to abstract different aspects of creating an efficient, data parallel SIMD program through vectorization. This section introduces a few of the key concepts to build a mental model and describe how they interact.

```
for (int i=0; i<LEN; i++) {
  array[i].x = 10000 + i;
  array[i].y = 20000 + i;
  array[i].z = 30000 + i;
}
for (int i=0; i< LEN; i++) {
  printf("%6.0f,%6.0f, %6.0f\n",
         array[i].x,
         array[i].y,
         array[i].z);
}
```

FIG. 11.5

Code for "Print Points" accessing data members inside the array instead of a local object.

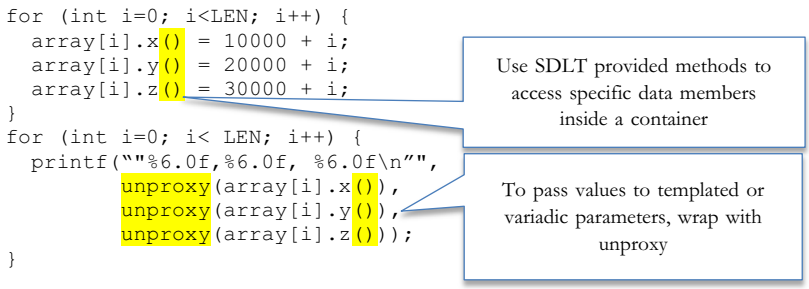


FIG. 11.6

Code for SDLT integrated "Print Points" accessing data members inside the array instead of a local object.

PRIMITIVES

Primitives represent the data we want an algorithm to process with SIMD operations. They can be more than just data structures. As a C++ object, it can have its own methods that modify its data. A primitive must be a "plain old data" object and meet other requirements documented in the Intel® C++ Compiler User and Reference Guide.

CONTAINERS

Containers encapsulate an array of primitives and abstract the in-memory data layout. The one-dimensional containers are dynamically sized and present a high-level interface similar to `std::vector`. Because the in-memory data layout is abstracted, SDLT containers may use an SIMD-friendly data layout such as SOA. Alternatively for compatibility, comparison, legacy, or performance reasons, we may choose to use an SDLT container with an AOS layout. Advanced users might choose to use the ASA layout.

To achieve an abstraction of the memory layout inside the containers, no reference to the underlying primitive is exposed by any interface. Instead, a primitive can be exported from, or imported to, a particular element in the container.

ACCESSORS

As the memory layout is abstracted, SDLT disallows pointer-based array accesses that are typically used with C arrays or `std::vector`. The container itself is managing ownership of the data and not suitable for passing by value to functions or lambda functions. The concept of an "accessor" provides an array subscript interface to the underlying data. Call the methods `access()` or `const_access()` on an SDLT container to obtain an accessor or constant accessor. Pass accessors by value where one would have passed an array pointer, reference to a `std::vector`, or iterator.

PROXY OBJECTS

As the memory layout is abstracted, SDLT disallows a direct reference to a primitive. Instead, the array subscript operator of an accessor returns a proxy object. A proxy object can export a primitive value corresponding to an element in a container via a conversion operator. A proxy object can import a primitive value into an element in the container via an assignment operator. The proxy objects know the underlying in-memory layout of a container and are responsible for ferrying the individual data members of the primitive between the value being imported or exported and the container.

EXAMPLE NORMALIZING 3D POINTS WITH SIMD

The next example, Fig. 11.7, adds helper methods and overloaded operators to the Point3s object in order to enable calculating its normal. The methods are inlined and represent a typical C++ implementation. The program instantiates a `std::vector` of 200,000,000 points and loops over them in SIMD normalizing each one.

```
#include <cstdio>
#include <cmath>
#include <vector>

struct Point3s
{
  float x, y, z;

  float lengthSquared() const {
    return x*x + y*y + z*z;
  }
  float length() const {
    return std::sqrt(lengthSquared());
  }
  Point3s operator *(float iFactor) const {
      Point3s scaled;
      scaled.x = x*iFactor;
      scaled.y = y*iFactor;
      scaled.z = z*iFactor;
      return scaled;
  }
};


void main() {
  std::vector<Point3s> points(200000000);


  #pragma forceinline recursive
  {
    const int count = static_cast<int>(points.size());
    #pragma omp simd
    for (int i = 0; i < count; ++i) {
      const Point3s p = points[i];
      float reciprocolOfLength = 1.0f/p.length();
      points[i] = p*reciprocolOfLength;
    }
  }
```

Ensure code block is inlined

Calculate iteration bounds before for loop

Enforce vectorization of the loop

FIG. 11.7

Code for original AOS SIMD "Normalize Points."

Compilers may choose not to inline a function for a variety of reasons: e.g., per file limits, recursion limits, performance heuristics. To ensure the compiler respects our intent that the methods called are inlined, we place a "`#pragma forceinline recursive`" above the code block where we want to ensure inlining occurs. The Intel C++ compiler will inline all calls, allowed by C++ language rules, including any nested calls made by those inlined functions recursively. Typically function calls made inside an SIMD loop can prevent vectorization or degrade performance by requiring the data layout of any parameters passed to meet Application Binary Interface standards. By ensuring calls made inside the SIMD loop are inlined, we remove any function call overhead and enable the compiler to be in control of the memory layout for any local variables.

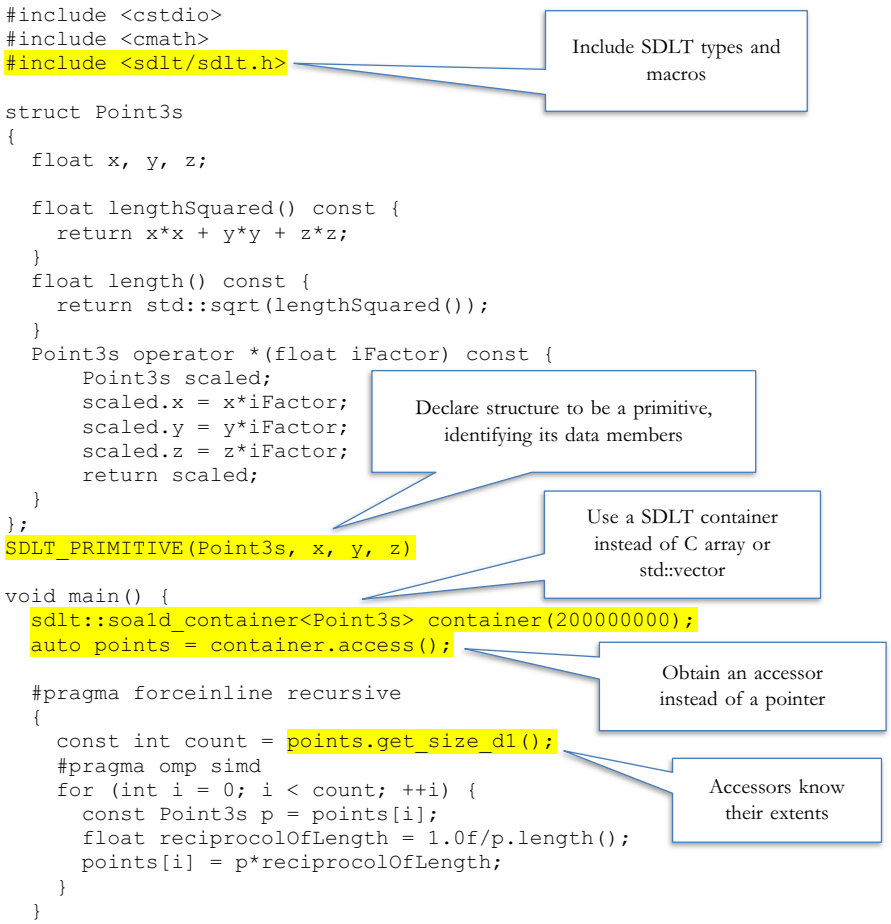


FIG. 11.8

Code for SDLT SOA SIMD "Normalize Points."

Although compilers may automatically vectorize a loop, it may choose to not do so based on performance heuristics, loop complexity, or assumed trip counts. When targeting SIMD execution, we should explicitly identify which loops should be vectorized by placing a "`#pragma omp simd`" immediately prior to the loop where we wish to force vectorization to occur. That instructs the compiler that each iteration of the loop is truly independent, can execute in parallel, and to vectorize the loop (ignoring heuristics that would otherwise cause the compiler to avoid vectorization).

A SIMD loop needs to conform to the for-loop style of an OpenMP canonical loop (per the OpenMP specification, see "For More Information" at the end of this chapter). Most notably, the iteration count should be computed before executing the loop. We declare a local variable "`count`" and calculate the size of the points vector before entering the loop. On the Knights Landing, this vectorized SIMD loop provides a 2.2× speedup over scalar code.

Fig. 11.8 shows the SDLT integration. Note that accessors know their extents and can be used directly in a canonical for loop. On Knights Landing, the SDLT vectorized SIMD loop provides over a 15× speedup over scalar code. That is 7× better than the original SIMD speedup with an AOS data layout.

WHAT IS WRONG WITH AOS MEMORY LAYOUT AND SIMD?

SIMD instructions allow the same operation to be applied to multiple data elements efficiently with a single instruction as illustrated in Fig. 11.9.

Unfortunately, getting AOS in-memory data layout loaded into a vector register is a "strided" load/store operation requiring multiple load/shuffle/insert or gather instructions as illustrated in Fig. 11.10.

With AOS, as vector register width increases so do the number of instructions required to populate the vector registers and performance can plateau. If a processor

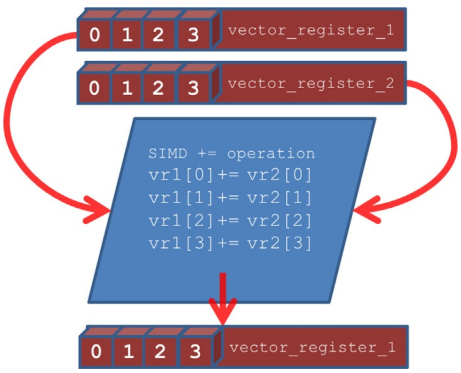


FIG. 11.9

Four wide vector registers performing += operation.

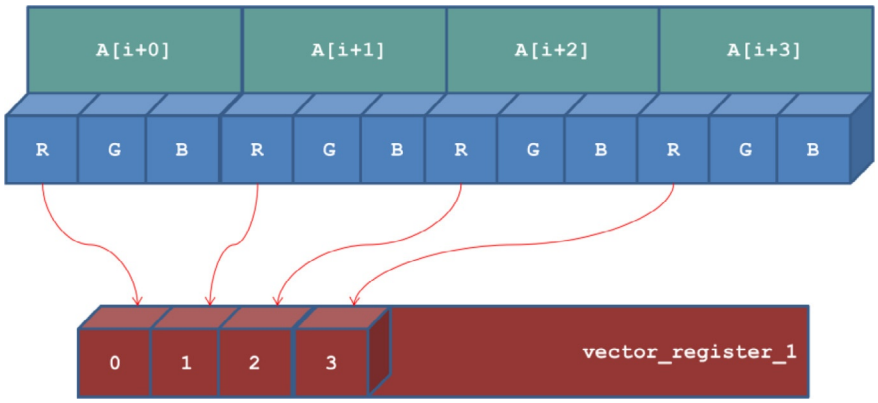


FIG. 11.10

Loading four wide vector registers from AOS memory layout.

reduces frequency to execute vector instructions, a limited SIMD improvement might not overcome scalar code operating at a higher frequency. However, if the compute portion of the algorithm is large relative to the number of memory accesses, then the AOS overhead may be amortized.

SIMD PREFERS UNIT-STRIDE MEMORY ACCESSES

If memory layout has multiple instances of a data member adjacent in memory and aligned on a byte boundary matching the vector register width:

- Single load/store instruction to move the data into or out of a vector register
- Many SIMD operations can reference an aligned unit-stride memory access as part of the instruction, avoiding a separate load/store instruction altogether.

A properly aligned SOA in-memory data layout provides SIMD compatible Unit-Stride memory accesses as illustrated in Fig. 11.11.

Reductions in SIMD efficiency due to gather/scatter or strided loads/stores of AOS data can be restored by switching to SOA.

INTEGRATING SOA DATA LAYOUT

To achieve SOA, we have traditionally had to abandon their high-level objects, define an SOA, and implement their algorithms accessing individual array elements representing the data members of the object. This approach can require large rewrites, obfuscation of the algorithm, modifying function parameters from objects to individual arrays for each data member, and introduction of compiler-specific alignment hints.

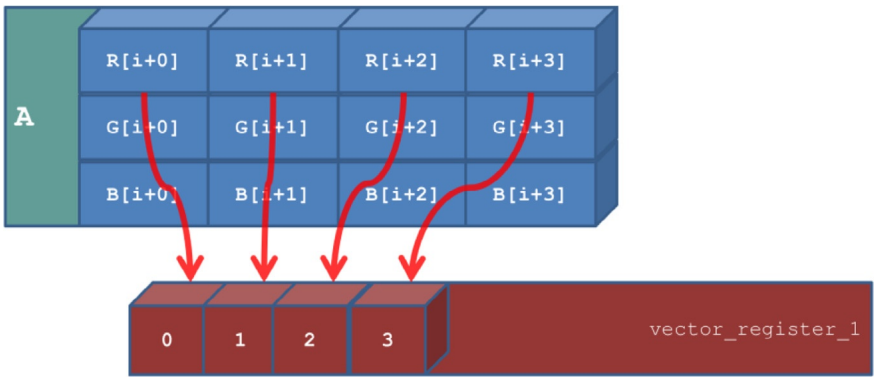


FIG. 11.11

Loading four wide vector registers from SOA memory layout.

What SDLT proposes is to restrict the capabilities of the objects involved to meet its primitive requirements. SDLT can transparently use an SOA memory layout inside a container and keep the high-level scalar object representation of the algorithm. If all of the methods used inside the loop body can be inlined, the compiler can track the underlying SOA data accesses through the scalar representation of the algorithm. When the compiler can keep the SOA format through the entire algorithm, it can generate efficient SIMD code while taking advantage of compiler optimizations. Furthermore, SDLT containers guarantee aligned buffer allocation, and SDLT proxy objects will emit the necessary alignment hints keeping them out of your code.

ALPHA-BLENDED OVERLAY REFERENCE

To further illustrate SDLT usage, we show a reference solution to the following scenario:

- a base image and output image represented by single precision floating point red, green, and blue values,
- an overlay image represented by single precision floating point red, green, blue, and alpha values, and
- perform an alpha blend of the overlay image on top of base image and place the result into the output image.

In our solution, we choose to use high-level objects to represent each pixel in the images and overlay and add reusable methods to the objects to assist in implementing the alpha blend algorithm as depicted in Fig. 11.12.

As the algorithm will be applied to all pixels regardless of position in the image, the code linearizes the 2d image into a 1d array. For simplicity and familiarity

```
struct RGBs
{
    float red;
    float green;
    float blue;

    RGBs operator * (float iScale) const {
        RGBs scaled;
        scaled.red =     red*iScale;
        scaled.green = green*iScale;
        scaled.blue =   blue*iScale;
        return scaled;
    }

    RGBs operator + (const RGBs & iOther) const {
        RGBs sum;
        sum.red =     red + iOther.red;
        sum.green = green + iOther.green;
        sum.blue =   blue + iOther.blue;
        return sum;
    }
};

struct RGBAs : public RGBs
{
    float alpha;
};
```

FIG. 11.12

Code for high-level objects to represent a pixel.

```
const int width = 256;
const int height = 256;
const int pixelCount = width*height;

std::vector<RGBs> baseImage(pixelCount);
std::vector<RGBAs> overlayImage(pixelCount);
std::vector<RGBs> outputImage(pixelCount);

loadImageStub(baseImage.data());
loadImageStub(overlayImage.data());

referenceAlphaBlendOverlay(
    pixelCount,
    baseImage.data(),
    overlayImage.data(),
    outputImage.data());
```

FIG. 11.13

Code for reference container definition and image setup.

a `std::vector` is used, shown in Fig. 11.13. Stub (empty) functions pretend to populate the images as the performance is independent of pixel values.

The algorithm is implemented in Fig. 11.14. It simply loops over each pixel, computes the alpha-blended result, and stores it to the output image. It uses array pointers with the loop index to access image data. Note that the loop body implements the alpha blend algorithm in terms of high-level objects utilizing overloaded

```
void referenceAlphaBlendOverlay(
    const int pixelCount,
    RGBs const * baseImage,
    RGBAs const * overlayImage,
    RGBs * outputImage)
{
    #pragma forceinline recursive
    {
        #pragma novector
        for (int i = 0; i < pixelCount; ++i)
        {
            const RGBs basePixel = baseImage[i];
            const RGBAs overlayPixel = overlayImage[i];
            float oneMinusAlpha = (1.0f - overlayPixel.alpha);
            RGBs blendedPixel =
                basePixel*oneMinusAlpha +
                (RGBs(overlayPixel)*overlayPixel.alpha);
            outputImage[i] = blendedPixel;
        }
    }
}
```

FIG. 11.14

Code to loop over pixels.

operators (+ and *). To obtain a performance baseline, we disable vectorization on the loop with `#pragma novector`.

The code presented in Figs. 11.12–11.14 is a reasonable representation of how we might typically implement the algorithm. Timing the algorithm running repeatedly on a single thread on a single core of a Knights Landing system, we measured 96,646,925 pixels processed per second. This rate will be used as the baseline for the rest of the examples.

VECTORIZING WITH AOS

We explicitly disabled vectorization on the reference version of the algorithm. That prevents the compiler from taking advantage of Intel® AVX-512, where 16 floating point operations could execute at a time. Change that line above the loop to explicitly vectorize the loop:

```
#pragma omp simd
```

To examine how the compiler vectorized the loop, instruct the compiler to generate an optimization report for the vectorization phase by adding the following command line flags:

```
-qopt-report=5 -qopt-report-phase=vec
```

Fig. 11.15 shows the generated `*.optrpt` file.

SIMD code for vector length of 16 was generated, but due to the AOS memory layout of the pixels, complicated load sequences, gather, or scatters were generated. This reduces the efficiency of the SIMD code. Although the report estimates a "potential" speedup of 2.36×, in actual testing the pixels processed per second achieved was 104,980,015 (only a 1.09× speedup).

```
LOOP BEGIN at AlphaBlendOverlay.cc(97,5)
   ...
   remark #15305: vectorization support: vector length 16
   remark #15309: vectorization support: normalized
vectorization overhead 0.047
   remark #15301: SIMD LOOP WAS VECTORIZED
   remark #15460: masked strided loads: 7
   remark #15462: unmasked indexed (or gather) loads: 3
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 44
   remark #15477: vector loop cost: 17.370
   remark #15478: estimated potential speedup: 2.360
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

FIG. 11.15

Optimization report for vectorization of reference alpha blend algorithm.

The difference between the estimated potential speedup, and the measured speedup, is large enough that we should suspect something is wrong. Investigating the generated assembly code, we observed an optimization where 64 bytes of the AOS is loaded then permuted into SIMD registers versus issuing a gather instruction. When 64 bytes are loaded from an unaligned memory address, two cache lines are accessed. These wide loads perform best when the address is cache line aligned. The `std::vector` makes no such guarantees of alignment. By switching from `std::vector` to plain array allocated by `_mm_malloc` specifying 64-byte alignment, the same SIMD loop achieved 180,091,650 pixels processed per second (a 1.89× speedup).

The measured speedup is much closer to the estimated potential. Latencies introduced by memory accesses and other factors inhibit code from reaching its potential. However, when generating SIMD code that performs 16 floating point operations at a time, we would ideally reach a higher efficiency (speedup).

ALPHA-BLENDED OVERLAY WITH SDLT

We will reuse the exact, same RGBs and RGBAs from the reference code as it already met the requirements, of SDLT, to be a primitive. C++ lacks compile time reflection, so we must provide SDLT with some information about the layout of the data structure for the primitive. This is easily done with the SDLT_PRIMITIVE helper macro that accepts a `struct` type followed by a list of its data members.

```
SDLT_PRIMITIVE(RGBs, red, green, blue)
SDLT_PRIMITIVE(RGBAs, red, green, blue, alpha)
```

In Fig. 11.16, we change the `std::vector` to `sdlt::soa1d_container`, and pass SDLT accessors instead of array pointers.

The SDLT version of the algorithm in Fig. 11.17 is unchanged, except the parameters changed from array pointers to SDLT accessors. Using templated parameters will not burden development with the concrete type names. Also, it allows for array pointers to be passed to the same algorithm code for comparison purposes. Fig. 11.18 shows the generated `*.optrpt` file after rebuilding.

```
const int width = 256;
const int height = 256;
const int pixelCount = width*height;

sdlt::soa1d_container<RGBs> baseImage(pixelCount);
sdlt::soa1d_container<RGBAs> overlayImage(pixelCount);
sdlt::soa1d_container<RGBs> outputImage(pixelCount);

loadImageStub(baseImage.access());
loadImageStub(overlayImage.access());

sdltAlphaBlendOverlay(
    pixelCount,
    baseImage.const_access(),
    overlayImage.const_access(),
    outputImage.access());
```

Use a SDLT container instead of std::vector

Pass accessors instead of a pointers

const_access for read only

FIG. 11.16

Code for SDLT container definition and image setup.

```
template <
    typename BaseAccessorT,
    typename OverlayAccessorT,
    typename OutAccessorT>
void sdltAlphaBlendOverlay(
    const int pixelCount,
    BaseAccessorT baseImage,
    OverlayAccessorT overlayImage,
    OutAccessorT outputImage)
{
    #pragma forceinline recursive
    {
        #pragma omp simd
        for (int i = 0; i < pixelCount; ++i)
        {
            const RGBs basePixel = baseImage[i];
            const RGBAs overlayPixel = overlayImage[i];
            float oneMinusAlpha = (1.0f - overlayPixel.alpha);
            RGBs blendedPixel =
                basePixel*oneMinusAlpha +
                RGBs(overlayPixel)*overlayPixel.alpha;
            outputImage[i] = blendedPixel;
        }
    }
}
```

Convenience to make accessor types templated

Pass SDLT accessors instead of array pointers

FIG. 11.17

Code for SDLT loop over pixels using accessors.

```
LOOP BEGIN at AlphaBlendOverlay.cc(165,9)
...
   remark #15305: vectorization support: vector length 16
   remark #15309: vectorization support: normalized
vectorization overhead 0.245
   remark #15301: SIMD LOOP WAS VECTORIZED
   remark #15448: unmasked aligned unit stride loads: 7
   remark #15449: unmasked aligned unit stride stores: 3
   remark #15475: --- begin vector loop cost summary ---
   remark #15476: scalar loop cost: 79
   remark #15477: vector loop cost: 3.310
   remark #15478: estimated potential speedup: 22.080
   remark #15488: --- end vector loop cost summary ---
LOOP END
```

FIG. 11.18

Optimization report for vectorization of SDLT alpha blend algorithm.

Again SIMD code for vector length of 16 was generated. But this time all of the loads and stores are unmasked, aligned, and unit stride. That means the fastest method of loading data into the vector registers is being used.

The "estimated" potential speedup is 22×. However, that is comparing SDLT SOA scalar versus vectorized code. Using the AOS reference version optimization report scalar loop cost of 44 versus SDLT vector loop cost of 3.31, combined with normalized vectorization overhead 0.245, we can estimate a more realistic $(44/(3.31+0.245)) = 12.37\times$ speedup. When measured on hardware, memory latency will have an effect. The real-world performance achieved is 1,038,843,701 pixels processed per second. That brings the total speedup due to vectorization up to 10.75× (1.9× on Haswell-EP) as shown in Figs. 11.19 and 11.20. Compared to the aligned vectorized AOS reference code, SDLT SOA shows a 5.8× speedup.

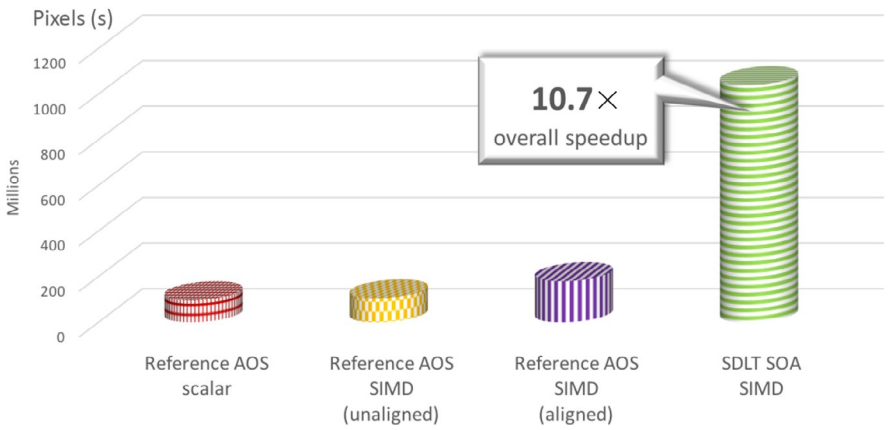


FIG. 11.19

Pixels processed per second on Knights Landing.

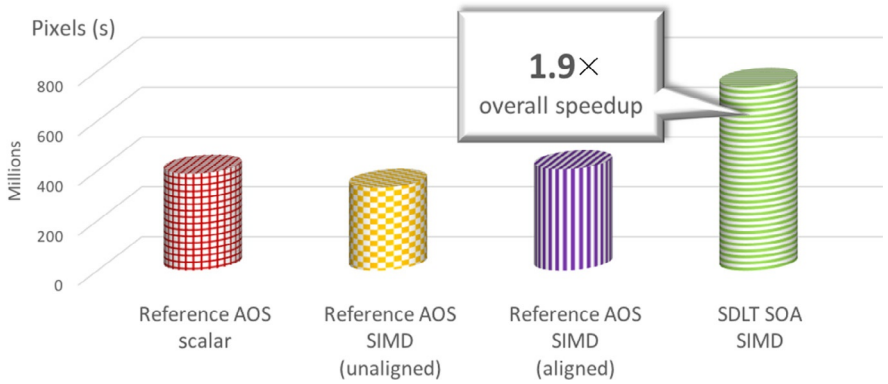


FIG. 11.20

Pixels processed per second on Haswell-EP.

ADDITIONAL FEATURES

The design of SDLT intends performance code to use accessors to get at the data. Although for ease of integration, the containers support a similar interface to `std::vector` and can be resized dynamically and provide iterators compatible with STL algorithms (`std::sort`, `std::find`, etc.).

Primitive structures can have other declared primitive data types as their own data members. So primitives can nest and be more complex hierarchical structures versus just a flat data structure.

Proxy objects overload all operators supported by the underlying primitive data types. Therefore, proxy objects can be used directly in an algorithm without first exporting them to a local variable.

Because accessors are used instead of pointers, where we would have added an offset to a pointer, we would instead pass that offset as a parameter to the `access(offset)` or `const_access(offset)` methods for the container to obtain an accessor with the offset embedded inside. The embedded offset is automatically applied to all indexes passed to the `[]` operator for the accessors. SDLT provides the `aligned_offset<AlignmentT>` object to model an offset which guarantees it is a multiple of an `AlignmentT`. This allows for aligned SIMD code generation even when offsets are applied to an index. The `fixed_offset<IntegerT>` object models an offset known at compile time, allowing the compiler to detect when it is a multiple of the SIMD width and maintain aligned code generation.

SUMMARY

SDLT provides containers to abstract data layout and a methodology to efficiently vectorize algorithms for Knights Landing or any processor. It helps us leverage SIMD compilers and hardware performance without requiring an expert. It can be

used with any compiler supporting C++11, and its SIMD-friendly data layout can better take advantage of the performance features in the Intel compiler such as the OpenMP SIMD pragmas.

SDLT provides the following benefits:

- Enables generic programming with independence from data layout
- Allows object-oriented representation of an algorithm with "plain old data"
- Keeps the algorithm at the top level
- Enables programming in an AOS style while data layout is SOA for vectorization
- Encapsulate best known usage patterns and avoid common pitfalls

SDLT began shipping as part of the Intel® C++ Compiler 16 Update 1 in late 2015.

FOR MORE INFORMATION

Here are some additional reading materials that we recommend related to this chapter.

- Intel® C++ Compiler 16.0 User and Reference Guide, https://software.intel.com/en-us/node/524394.
- Intel® SIMD Data Layout Compiler Feature, https://software.intel.com/en-us/code-samples/intel-compiler/intel-compiler-features/intel-sdlt.
- Alpha-Blending Code, https://en.wikipedia.org/wiki/Alpha_compositing.
- DreamWorks Animation (DWA): How We Achieved a 4× Speedup of Skin Deformation with SIMD, http://www.slideshare.net/IntelSoftware/dreamwork-animation-dwa.
- OpenMP Application Program Interface, http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.
- Download the code from this and other chapters, http://lotsofcores.com.

CHAPTER

Vectorization with AVX-512 intrinsics 12

What is new with Knights Landing in this chapter?
AVX-512

This chapter introduces programming with intrinsics for Intel® Advanced Vector Extensions 512-bits (AVX-512). Writing a program, in Fortran, C, or C++, and having the compiler generate AVX-512 instructions should be the first choice for us because that offers better portability. However, there are times when the richness of the AVX-512 instructions is not easily reached due to the combined limitations of the languages and compilers being used. For such instances, AVX-512 intrinsics offer direct and efficient ways to utilize AVX-512 instructions from the comfort of C or C++ code. Fortran programs utilize C interfaces to get the capabilities through C programming. Using intrinsics in one part of an application does not restrict choices for vectorization techniques elsewhere in an application in any way. We recommend also reading the section *Advice: Use AVX-512* in Chapter 6 at some point.

WHAT ARE INTRINSICS?

Intrinsics look like functions within the C and C++ languages, but they do not generate function calls. Instead, they have a direct correspondence to the actual SIMD instructions in Knights Landing (or other processors supporting AVX-512). Intrinsics for SIMD instructions, as an alternative to assembly code and inline assembly code, started with MMX in 1997. Since then, SIMD intrinsics have been added for SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE 4.2, AVX, AVX2, and now AVX-512.

The advantage of intrinsics is that the programmer does not have to write low-level assembly and also manage low-level instruction scheduling and register allocation, all of which is microarchitecture dependent and better handled by the compiler. Further, with intrinsics the compiler can generate better code such as fusing two SIMD instructions on platforms that support particular combinations.

By virtue of the 512-bit wide capabilities, AVX-512 offers the highest performance for floating-point arithmetic. Intrinsics are handled specially by the compiler to generate the instructions inline (i.e., without a function call). A simple example to

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00012-0

load two arrays containing 16 floating-point numbers and add them together looks like this:

```
__m512 simd1 = _mm512_load_ps(a); // read 16 floats from memory
__m512 simd2 = _mm512_load_ps(b); // read 16 more floats
__m512 simd3 = _mm512_add_ps(simd1, simd2); // add them
```

Using intrinsics appears no different than calling a function. We simply include the header file for all AVX intrinsics (immintrin.h) and call the desired intrinsic function. As with other functions, we must observe the parameter and return types of the intrinsic function. Fortunately, there is extensive documentation online (see *For More Information* at the end of this chapter). The "ps" suffix on these instructions means "packed single" as explained in a handy table of suffixes shown in Fig. 12.1.

Using the –S compiler option is one way to see the actual assembly code generated by the compiler including the AVX-512 instructions. Our previous three lines of C code were responsible for creating the assembly code shown here:

```
vmovaps 256(%rsp), %zmm1
vmovaps 320(%rsp), %zmm2
vaddps  %zmm2, %zmm1, %zmm3
vmovaps %zmm3, (%rsp)
```

Our short sequence of C code assumes that we previously declared `float a[16]` and `float b[16]`. Note that we declared the three `simd` variables with `__m512`, which is a special 512-bit wide variable type. One benefit of this data type is that the compiler will automatically align the `__m512` to a 64-byte boundary. Figs. 12.2 and 12.3 expand on our three lines with a simple "Hello AVX512" which uses the SIMD add, subtract, multiply, and divide instructions and then shows use of a masked fused multiply-add (FMA) instruction with different zeroing-masks. Fig. 12.4 shows the output of this simple program. The code can be downloaded from our website, see *For More Information* at the end of this chapter.

Suffix	Type	Description
`pd`	**__m512d**	Packed Doubles: 8 doubles in 512-bits
`ps`	**__m512s**	Packed Singles: 16 floats in 512-bits
`sd`	**double**	Single Double: double in lower 64-bits
`ss`	**float**	Single Single: float in lower 32-bits
`si512`	**__m512i**	Single Integer 512-bits: think "512 booleans"
`epi32`	**__m512i**	Extended Packed Integer: 32 bit signed integers
`epi64`	**__m512i**	Extended Packed Integer: 64 bit signed integers
`f32x4`	**__m512**	Floating-point 32-bits x4: four floats in 128-bits
`f64x4`	**__m512d**	Floating-point 64-bits x4: four doubles in 256-bits
`i32x4`	**__m512i**	Integer 32-bits x4: four 32-bit integers in 128-bits
`i64x4`	**__m512i**	Integer 64-bits x4: four 64-bit integers in 256-bits

FIG. 12.1

Most common Knights Landing AVX-512 intrinsic data-type suffixes; the top four being the most used.

```
#include <stdio.h>
#include "immintrin.h"
int main(int argc, char *argv[]) {
  float a[] = { 9.9,-1.2, 3.3,4.1, -1.1,0.2,-1.3,4.4,
                2.4, 3.1,-1.3,6.0,  1.5,2.4, 3.1,4.2 };
  float b[] = { 0.3, 7.5, 3.2,2.4,  7.2,7.2, 0.6,3.4,
                4.1, 3.4, 6.5,0.7,  4.0,3.1, 2.4,1.3 };
  float c[] = { 0.1, 0.2, 0.3,0.4,  1.0,1.0, 1.0,1.0,
                2.0, 2.0, 2.0,2.0,  3.0,3.0, 3.0,3.0 };
  float o[] = { 0,0,0,0, 0,0,0,0,  0,0,0,0, 0,0,0,0 };
  __m512 simd1, simd2, simd3, simd4;
  __mmask16 m16z = 0;
  __mmask16 m16s = 0xAAAA;
  __mmask16 m16a = 0xFFFF;
  print("  a[]",a,16);
  print("  b[]",b,16);
  print("  c[]",c,16);
  if ( _may_i_use_cpu_feature( _FEATURE_AVX512F ) ) {
    simd1 = _mm512_load_ps(a);
    simd2 = _mm512_load_ps(b);
    simd3 = _mm512_load_ps(c);
    simd4 = _mm512_add_ps(simd1, simd2);
    _mm512_store_ps(o, simd4);
    print("  a+b",o,16);
    simd4 = _mm512_sub_ps(simd1, simd2);
    _mm512_store_ps(o, simd4);
    print("  a-b",o,16);
    simd4 = _mm512_mul_ps(simd1, simd2);
    _mm512_store_ps(o, simd4);
    print("  a*b",o,16);
    simd4 = _mm512_div_ps(simd1, simd2);
    print("  a/b",(float *)&simd4,16);
    printf("FMAs with mask 0, then mask 0xAAAA, ");
    printf("then mask 0xFFFF\n");
    simd4 = _mm512_maskz_fmadd_ps(m16z,simd1,simd2,simd2);
    print("a*b+b",(float *)&simd4,16);
    simd4 = _mm512_maskz_fmadd_ps(m16s,simd1,simd2,simd3);
    print("a*b+b",(float *)&simd4,16);
    simd4 = _mm512_maskz_fmadd_ps(m16a,simd1,simd2, simd3);
    print("a*b+b",(float *)&simd4,16);
  }
  return 0;
}
```

FIG. 12.2

Hello AVX-512.

```
void print(char *name, float *a, int num)
{
  int i;
  printf("%s =%6.1f",name,a[0]);
  for (i = 1; i < num; i++)
    printf(",%s%4.1f",(i&3)?"":" ",a[i]);
  printf("\n");
}
```

FIG. 12.3

`print` function for Hello AVX-512.

```
   a[] =   9.9,-1.2, 3.3, 4.1, -1.1, 0.2,-1.3, 4.4,  2.4, 3.1,-1.3, 6.0,  1.5, 2.4, 3.1, 4.2
   b[] =   0.3, 7.5, 3.2, 2.4,  7.2, 7.2, 0.6, 3.4,  4.1, 3.4, 6.5, 0.7,  4.0, 3.1, 2.4, 1.3
   c[] =   0.1, 0.2, 0.3, 0.4,  1.0, 1.0, 1.0, 1.0,  2.0, 2.0, 2.0, 2.0,  3.0, 3.0, 3.0, 3.0
   a+b =  10.2, 6.3, 6.5, 6.5,  6.1, 7.4,-0.7, 7.8,  6.5, 6.5, 5.2, 6.7,  5.5, 5.5, 5.5, 5.5
   a-b =   9.6,-8.7, 0.1, 1.7, -8.3,-7.0,-1.9, 1.0, -1.7,-0.3,-7.8, 5.3, -2.5,-0.7, 0.7, 2.9
   a*b =   3.0,-9.0,10.6, 9.8, -7.9, 1.4,-0.8,15.0,  9.8,10.5,-8.4, 4.2,  6.0, 7.4, 7.4, 5.5
   a/b =  33.0,-0.2, 1.0, 1.7, -0.2, 0.0,-2.2, 1.3,  0.6, 0.9,-0.2, 8.6,  0.4, 0.8, 1.3, 3.2
FMAs with mask 0, then mask 0xAAAA, then mask 0xFFFF
a*b+b =   0.0, 0.0, 0.0, 0.0,  0.0, 0.0, 0.0, 0.0,  0.0, 0.0, 0.0, 0.0,  0.0, 0.0, 0.0, 0.0
a*b+b =   0.0,-8.8, 0.0,10.2,  0.0, 2.4, 0.0,16.0,  0.0,12.5, 0.0, 6.2,  0.0,10.4, 0.0, 8.5
a*b+b =   3.1,-8.8,10.9,10.2, -6.9, 2.4, 0.2,16.0, 11.8,12.5,-6.4, 6.2,  9.0,10.4,10.4, 8.5
```

FIG. 12.4

Output of Hello AVX-512.

WHICH COMPILERS SUPPORT INTRINSICS?

All widely used compilers implement intrinsics for MMX, SSE, SSE2, SSE3, SSE4, AVX, and AVX2. At the time of publication for this book, Intel, Clang/LLVM, and gcc compilers have had AVX-512 support well before Knights Landing was available. We know that additional compilers are adding AVX-512 support as well. Because Intel has made AVX-512 a part of its official processor instruction set, AVX-512 support is expected to make it into all major compilers. This is in contrast with the 512-bit SIMD instructions for Knights Corner, which only found support in the Intel compilers due to their coprocessor specific nature. AVX-512 are processor instructions supported in Knights Landing processors, some newer Intel® Xeon® processors, and they are also supported by Knights Landing coprocessors.

ASSEMBLY LANGUAGE PROGRAMMING OPTIONS

In this chapter, we emphasize programming AVX-512 instructions with intrinsics. There are, however, three basic options for programming AVX-512 instructions as explained in Fig. 12.5. Of these three options, we recommend intrinsics as they are the easiest to program once the decision to use assembly language is made, and intrinsics are the most portable of the assembly coding methods.

Chapter 21 utilizes inline assembly to optimize small matrix multiply kernels for use in seismic simulations. Fig. 12.6 shows an example of `__asm__` usage from `sgemm_knl.cpp` from the code used in Chapter 21. The `__volatile__` indicates that the assembly code may have side effects which in turn limits the ability for the compiler to optimize across the `__asm__`. The `__asm__` has four clauses separated by colons `(:)`. The first is the assembly code, the second is an optional list of outputs, the third is an optional list of inputs, and the fourth lists clobbered registers. We can contrast this with writing in assembly directly (Fig. 12.7).

Description	Advantage	Disadvantage
Intrinsics: built-in function style interfaces that directly correspond to certain assembly instructions. Example of syntax shown in Fig. 12.2	• Program in C / C++ • Supported in major C / C++ compilers (Intel, gcc, Clang/LLVM, Microsoft, etc.) • Most portable of the assembly language options • Instructions abstracted as function call • Use C/C++ variables • Very simple function-style interface • No need to manage machine registers • Focus on writing only the key code, no need to manage function entry/exit code	• Less portable than pure C/C++ code because it is hard coded to a specific SIMD instruction set, whereas a compiler code compiler to different widths, or different instruction sets • Compiler responsible for instruction selection and register coloring. No way to force a different encoding if the compiler does not match your expectations (not usually an issue)
Inline assembly code: use __asm__ construct (or __asm with Microsoft). Example of syntax shown in Fig. 12.6	• Program in C / C++ • More control than with intrinsics • Supported with high compatibility between Intel, gcc and Clang/LLVM compilers • Use C/C++ variables • Focus on writing only the key code, no need to manage function entry/exit code	• Less portable than intrinsics because compiler support is not as widespread and not as compatible • Program in assembly language, not C/C++; more complex than a function-style interface • Clang/LLVM is stricter than GCC leading to some challenges in staying portable • Microsoft Visual Studio syntax available but more primitive and not compatible
Assembly code: use gas, nasm, MASM (or, with some compilers, specify an assembly file as the file to compiler and have the compiler invoke the assembler). Example of syntax shown in Fig. 12.7	• Full control • Can start with C/C++ source code to create initial assembly language file by using -S compiler option (/FA for Microsoft compiler), then editing can be limited to the key parts, having gotten the interface done by the compiler	• Program in assembly language, not C/C++; more complex than a function-style interface • Programmer must manage registers, select exact instructions, write the proper entry, and exit code to define a function or subroutine • Assembly code is not portable across all assemblers

FIG. 12.5

Assembly language programming options.

```
__asm__ __volatile__(
  "movq %0, %%rsi\n\t"
  "movq %1, %%rdi\n\t"
  "movq %2, %%rdx\n\t"
  "movq $0, %%r12\n\t"
  "movq $0, %%r13\n\t"
  "movq $0, %%r14\n\t"
  "0:\n\t"
  "addq $16, %%r12\n\t"
  "vpxord %%zmm23, %%zmm23, %%zmm23\n\t"
  "vpxord %%zmm24, %%zmm24, %%zmm24\n\t"
  "vpxord %%zmm25, %%zmm25, %%zmm25\n\t"
  "vpxord %%zmm26, %%zmm26, %%zmm26\n\t"
  "vpxord %%zmm27, %%zmm27, %%zmm27\n\t"
  "vpxord %%zmm28, %%zmm28, %%zmm28\n\t"
  "vpxord %%zmm29, %%zmm29, %%zmm29\n\t"
  "vpxord %%zmm30, %%zmm30, %%zmm30\n\t"
  "vpxord %%zmm31, %%zmm31, %%zmm31\n\t"
  "vmovaps 0(%%rdi), %%zmm0\n\t"
  "vfmadd231ps 0(%%rsi)%{1to16%}, %%zmm0, %%zmm23\n\t"
  "vfmadd231ps 64(%%rsi)%{1to16%}, %%zmm0, %%zmm24\n\t"
  "vfmadd231ps 128(%%rsi)%{1to16%}, %%zmm0, %%zmm25\n\t"
  "vfmadd231ps 192(%%rsi)%{1to16%}, %%zmm0, %%zmm26\n\t"
  "vfmadd231ps 256(%%rsi)%{1to16%}, %%zmm0, %%zmm27\n\t"
  "vfmadd231ps 320(%%rsi)%{1to16%}, %%zmm0, %%zmm28\n\t"
  "vfmadd231ps 384(%%rsi)%{1to16%}, %%zmm0, %%zmm29\n\t"
  "vfmadd231ps 448(%%rsi)%{1to16%}, %%zmm0, %%zmm30\n\t"
  "vfmadd231ps 512(%%rsi)%{1to16%}, %%zmm0, %%zmm31\n\t"
  "vmovaps 256(%%rdi), %%zmm0\n\t"
  ((39 lines omitted in this listing))
  "addq $64, %%rdx\n\t"
  "subq $960, %%rdi\n\t"
  "cmpq $16, %%r12\n\t"
  "jl 0b\n\t"
  : : "m"(B), "m"(A), "m"(C) :
"k1","rax","rbx","rcx","rdx","rdi","rsi","r8","r9","r10","r12",
"r13","r14","r15","zmm0","zmm1","zmm2","zmm3","zmm4","zmm5","zm
m6","zmm7","zmm8","zmm9","zmm10","zmm11","zmm12","zmm13","zmm14
","zmm15","zmm16","zmm17","zmm18","zmm19","zmm20","zmm21","zmm2
2","zmm23","zmm24","zmm25","zmm26","zmm27","zmm28","zmm29","zmm
30","zmm31");
```

FIG. 12.6

Example of inline assembly code used in SeisSol (Chapter 21) from sgemm_knl.cpp.

AVX-512 OVERVIEW

AVX-512 instructions are the first 512-bit SIMD instructions for processors. Applications can pack eight double-precision or sixteen single-precision floating-point numbers, or eight 64-bit integers, or sixteen 32-bit integers within the 512-bit vectors. This enables processing of twice the number of data elements that AVX/AVX2 can process with a single instruction and four times that of SSE.

Intel AVX-512 instructions are important because they offer higher performance for the most demanding computational tasks. Intel AVX-512 instructions offer the highest degree of compiler support by including an unprecedented level of richness in the design of the instructions. Intel AVX-512 features include 32 vector registers

```
L_B1.58:                        # Preds L_B1.57
  vmovaps   256(%rsp), %zmm1          #27.28
  lea       L_2__STRING.0(%rip), %rdi #32.5
  vmovaps   320(%rsp), %zmm2          #28.28
  vmovaps   384(%rsp), %zmm0          #29.28
  vmovaps   %zmm1, 192(%rsp)          #27.28
  vmovaps   %zmm2, 128(%rsp)          #28.28
  vmovaps   %zmm0, 64(%rsp)           #29.28
  vxorpd    %xmm0, %xmm0, %xmm0       #32.5
  vaddps    %zmm2, %zmm1, %zmm3       #30.13
  vmovaps   %zmm3, 448(%rsp)          #31.21
  lea       L_2__STRING.8(%rip), %rsi #32.5
  vmovaps   %zmm3, (%rsp)             #30.5
  vcvtss2sd 448(%rsp), %xmm0, %xmm0   #32.5
  movl      $1, %eax                  #32.5
L____tag_value__main.104:             #32.5
  call      _printf                   #32.5
```

FIG. 12.7

Sample of assembly code (generated by Intel compiler for part of Fig. 12.2).

(ZMM) each 512-bits wide, eight dedicated mask registers, 512-bit operations on packed floating-point data or packed integer data, embedded rounding controls (override global settings), embedded broadcast, embedded floating-point fault suppression, embedded memory fault suppression, new operations, additional gather/scatter support, high speed math instructions, compact representation of large displacement value, and the ability to have optional capabilities beyond the foundational capabilities. It is interesting to note that the 32 ZMM registers represent 2 KB of register space (2KB×72×4=576KB across 72 cores)!

AVX/AVX2 registers YMM0-YMM15 map into the Intel AVX-512 registers ZMM0-ZMM15, very much like SSE registers map into AVX registers. Therefore, in processors with Intel AVX-512 support, AVX and AVX2 instructions operate on the lower 128 or 256 bits of the first 16 ZMM registers with legacy instruction formats. Additionally, legacy AVX and SSE instructions can be extended to operate on the additional registers when using an EVEX-encoded form. Fig. 12.8 illustrates the relationship of ZMM, YMM, and XMM registers.

MASK REGISTERS AND PREDICATION

Wider SIMD operations have the advantage of more throughput, but the need to do partials (e.g., less than 16 single-precision floating-point operations) is very high. In particular, exiting from a loop often has remainder operations to perform because the original loop is not necessarily set to run an exact multiple of the SIMD length (16 or 8). The ability to use masking for predication is very helpful for such cases, and also for vectorizing a loop which contains conditional statements. Masking with memory operands will even avoid creating page faults if the masks avoid requiring a load or store to a page. This is very useful for preserving the original semantics of a program when vectorizing, something that was not possible with Knights Corner SIMD.

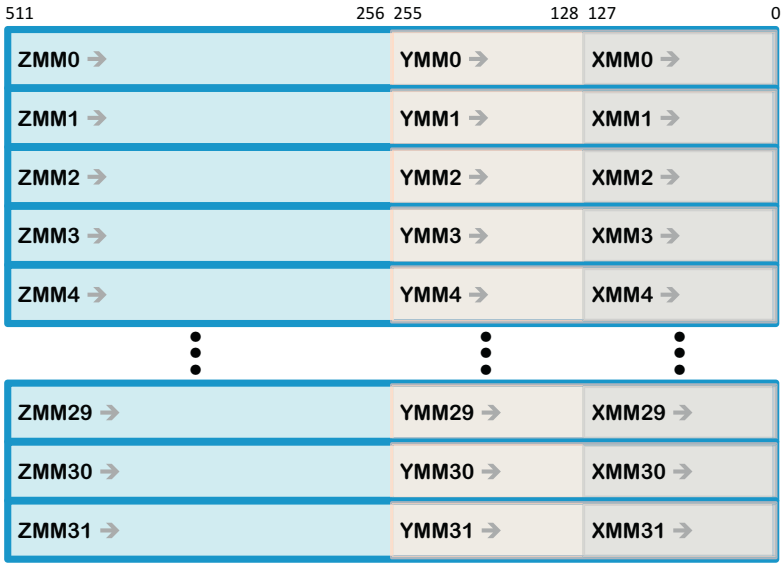


FIG. 12.8

Relationship of ZMM, YMM, and XMM registers.

AVX-512 instructions support 8 mask registers (k0–k7). The width of each mask register is architecturally defined as size MAX_KL (64 bits). Seven of the eight mask registers (k1–k7) can be used in conjunction with EVEX-encoded Intel AVX-512 F instructions to provide conditional execution and efficient merging of data elements in the destination operand. The encoding of mask register k0 does not actually use the value in the k0 register, instead it uses the value `0xFFFFFFFFFFFFFFFF`, which disables masking.

AVX-512 INSTRUCTION ENCODINGS

AVX/AVX2 instructions use the VEX prefix, while Intel AVX-512 instructions use the EVEX prefix which is one byte longer. The EVEX prefix enables the additional functionality of AVX-512. In general, if the extra capabilities of the EVEX prefix are not needed, then the AVX/AVX2 instructions can be used, coded using the VEX prefix saving a byte in certain cases. Such optimizations can be done in compiler code generators or assemblers automatically. This also works in reverse, where AVX/AVX2 instructions previously limited to 16 registers can use 16 more when encoded with the EVEX prefix. SSE instructions can also utilize all 32 registers when encoded with EVEX.

INTEL SOFTWARE DEVELOPMENT EMULATOR

In order to help with development, before Knights Landing was available, the Intel® Software Development Emulator (Intel® SDE) supports Intel AVX-512—see *For More Information* at the end of this chapter. The SDE has proven very popular and works very well. Of course, it does not offer the performance of real hardware,

but it does allow the functionality of AVX-512 to be exercised including feature detection sequences using `CPUID` and `XGETBV`. The SDE is based on runtime binary instrumentation, so it does not require any changes to binaries that are built and ready to run on Knights Landing.

In order to compile and run the code in Fig. 12.2 on a vintage 2010 MacBook Pro, we used two commands:

```
icc -xMIC-AVX512 -o helloavx512 helloavx512.c
sde -knl - ./helloavx512
```

The "icc" command uses the Intel compiler to create a binary ready to run on a Knights Landing (option -xMIC-AVX512). With the SDE installed, the Knights Landing binary (`helloavx512`) is run with the command "`sde -knl`." The `-knl` option creates a run environment for the binary that supports all instructions as if running on a Knights Landing.

INNOVATION BEYOND INTEL AVX-512 FOUNDATIONAL INSTRUCTIONS

Intel AVX-512 foundation (AVX-512 F) instructions and Intel AVX-512 conflict detection (AVX-512CD) instructions will be included in all implementations of Intel AVX-512 for Intel Xeon Phi processors and Intel Xeon processors. Products may also include capabilities that extend Intel AVX-512 and have distinct CPUID bits for detection. Knights Landing supports three sets of capabilities to augment the foundation instructions. This is documented in the programmer's guide; they are known as AVX-512CD instructions, Intel AVX-512 Exponential and Reciprocal (AVX-512ER) instructions, and Intel AVX-512 Prefetch (AVX-512PF) instructions. These capabilities provide efficient conflict detection to allow more loops to be vectorized, exponential and reciprocal operations for accelerating some mathematical operations, and new prefetch capabilities to help hide memory latency, respectively.

MIGRATING FROM KNIGHTS CORNER

AVX-512 binary encodings differ from Knights Corner binaries, so recompilation is always required. However, at the source code level, the intrinsics are largely compatible. Most intrinsics used by programmers for Knights Corner work with AVX-512 Knights Landing without modification. For more information on migrating from Knights Corner, or AVX2, we recommend reading the *IMCI to AVX-512* sections in Chapter 6.

This illustrates an interesting advantage of intrinsics over assembly language. Consider the intrinsic _mm512_add_ps. For Knights Corner, this produces VADDPS instructions which can only operate on ZMM registers and never on memory directly. In general, Knights Corner could only operate on memory operands with load and store instructions. Knights Landing, with AVX-512, has more efficient encodings which can utilize memory operands. The same _mm512_add_ps intrinsic compiled for Knights Landing produces the AVX-512 instruction ADDPS which can address memory in one operand. Since intrinsics do not actually specify registers, the compiler does the selection of how to map to the most efficient instruction formats. Thereby, automatically

helping bridge intrinsic code from Knights Corner to Knights Landing. This is an example of the advantage of intrinsics over other options as described in Fig. 12.5.

AVX-512 has many enhancements over the 512-bit SIMD available with Knights Corner, so there may be opportunities to rewrite code for AVX-512 to obtain some efficiencies. For specifics, please read the *IMCI to AVX-512* sections in Chapter 6.

AVX-512 DETECTION

Knights Landing supports AVX-512 instructions, specifically AVX-512 F (foundation), AVX-512CD (conflict detection), AVX-512ER (exponential and reciprocal), and AVX-512PF (prefetch).

If we will run an application exclusively on Knights Landing, we can simply use the instructions without any run time checks. For a little more protection, if we compile using the Intel compiler with option `-xMIC-AVX512`, the compiler will emit start-up code that will abort the program with the following message if the application is run on a processor with less support than Knights Landing:

```
Please verify that both the operating system and the processor support
Intel(R) MOVBE, F16C, AVX, FMA, BMI, LZCNT, AVX2, AVX512F, ADX, RDSEED,
AVX512ER, AVX512PF and AVX512CD instructions.
```

When we want an application to run everywhere, we should make sure that the operating system *and* the processor have support for AVX-512 when the application is run. Checking *both* is important. The Intel compiler provides a single function `_may_i_use_cpu_feature` that does all this easily. The program in Fig. 12.9 shows how we can use `_may_i_use_cpu_feature` to test for the ability to use AVX-512 F, AVX-512ER, AVX-512PF, and AVX-512CD instructions.

In order to run on all processors, we compile and run as follows:

```
icc -axMIC-AVX512 -o sample sample.c
./sample
```

```
#include <immintrin.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  const unsigned long knl_features =
      (_FEATURE_AVX512F | _FEATURE_AVX512ER |
       _FEATURE_AVX512PF | _FEATURE_AVX512CD );
  if ( _may_i_use_cpu_feature( knl_features ) )
    printf("This CPU supports AVX-512F+CD+ER+PF"
           " as introduced in Knights Landing.\n");
  else
    printf("This CPU does not support all"
           " Knights Landing AVX-512 features.\n");
  return 0;
}
```

FIG. 12.9

Using _may_i_use_cpu_feature to check for Knights Landing AVX-512.

When we run on a Knights Landing, it prints:

```
This CPU supports AVX-512F+CD+ER+PF as introduced in Knights Landing.
```

When we run on a processor without the AVX-512 support at least equivalent to Knights Landing, it prints:

```
This CPU does not support all Knights Landing AVX-512 features.
```

```
#if defined(__INTEL_COMPILER) && (__INTEL_COMPILER >= 1300)
#include <immintrin.h>
int has_intel_knl_features() {
  const unsigned long knl_features =
      (_FEATURE_AVX512F | _FEATURE_AVX512ER |
       _FEATURE_AVX512PF | _FEATURE_AVX512CD );
  return _may_i_use_cpu_feature( knl_features );
}
#else /* non-Intel compiler */

#include <stdint.h>
#if defined(_MSC_VER)
#include <intrin.h>
#endif

void run_cpuid(uint32_t eax, uint32_t ecx, uint32_t* abcd) {
#if defined(_MSC_VER)
  __cpuidex(abcd, eax, ecx);
#else
  uint32_t ebx, edx;
 #if defined( __i386__ ) && defined ( __PIC__ )
  /* in case of PIC under 32-bit EBX cannot be clobbered */
  __asm__ ( "movl %%ebx, %%edi \n\t cpuid"
            " \n\t xchgl %%ebx, %%edi"
            : "=D" (ebx),
 # else
  __asm__ ( "cpuid" : "+b" (ebx),
 # endif
                  "+a" (eax), "+c" (ecx), "=d" (edx) );
          abcd[0] = eax; abcd[1] = ebx;
          abcd[2] = ecx; abcd[3] = edx;
#endif
}

int check_xcr0_zmm() {
  uint32_t xcr0;
  uint32_t zmm_ymm_xmm = (7 << 5) | (1 << 2) | (1 << 1);
#if defined(_MSC_VER)
  /* min VS2010 SP1 compiler is required */
  xcr0 = (uint32_t)_xgetbv(0);
#else
  __asm__ ("xgetbv" : "=a" (xcr0) : "c" (0) : "%edx" );
#endif
  /* check if xmm, zmm and zmm state are enabled in XCR0 */
  return ((xcr0 & zmm_ymm_xmm) == zmm_ymm_xmm);
}

int has_intel_knl_features() {
  uint32_t abcd[4];
  uint32_t osxsave_mask = (1 << 27); // OSX.
  uint32_t avx2_bmi12_mask = (1 << 16) | // AVX-512F
                             (1 << 26) | // AVX-512PF
```

FIG. 12.10

Checking for Knights Landing AVX-512 in a manner that supports many compilers.

(Continued)

```
                                (1 << 27) | // AVX-512ER
                                (1 << 28);  // AVX-512CD
  run_cpuid( 1, 0, abcd );
  // step 1 - must ensure OS supports extended
  //          processor state management
  if ( (abcd[2] & osxsave_mask) != osxsave_mask )
    return 0;
  // step 2 - must ensure OS supports ZMM registers
  //          (and YMM, and XMM)
  if ( ! check_xcr0_zmm() )
    return 0;

  return 1;
}
#endif /* non-Intel compiler */

static int can_use_intel_knl_features() {
  static int knl_features_available = -1;
  /* test is performed once */
  if (knl_features_available < 0 )
    knl_features_available = has_intel_knl_features();
  return knl_features_available;
}

#include <stdio.h>
int main(int argc, char *argv[]) {
  if ( can_use_intel_knl_features() )
    printf("This CPU supports AVX-512F+CD+ER+PF"
           " as introduced in Knights Landing.\n");
  else
    printf("This CPU does not support all"
           " Knights Landing AVX-512 features.\n");
  return 0;
}
```

FIG. 12.10—CONT'D

If we want to support compilers other than Intel, the code is slightly more complex because the function `_may_i_use_cpu_feature` is not standard (and neither are the `__buildin` functions in gcc and clang/LLVM). The code in Fig. 12.10 works with at least the Intel compiler, gcc, clang/LLVM, and Microsoft compilers.

These run time checks allow binaries to selectively use instructions based on the machine they are run upon. Compilers also support compile time options to reflect compiler options used to build an application, which we may want to use. Compilers will define preprocessor symbols `__AVX512F__`, `__AVX512CD__`, `__AVX512ER__`, and `__AVX512PF__` when compiling for Knights Landing.

QUIRKS ABOUT WHEN INTRINSICS CAN BE USED

With Intel and Microsoft compilers, intrinsics can always be used (just include the right header file). Therefore, we can write intrinsics independent of the static code options (`-x, -ax, -m`). This gives us flexibly and enables an application to dispatch code manually according to the CPUID flags.

For apparently historical reasons, the gcc compiler limits use of intrinsics to those supported by the architectures specified in compiler flags. Therefore, the only way to write intrinsic code with gcc when using compiler flags that do not match the

intrinsics we want to use is by playing around with function attributes to get them to be permitted function by function.

LEARNING AVX-512 INSTRUCTIONS

Chapter 6 provides an introduction to AVX-512 and also provides detailed information on migrating from IMCI (512-bit SIMD supported by Knights Corner). We supply a detailed example of intrinsics usage later in this chapter as well as shorter examples in other chapters. We will not provide a detailed tutorial on the AVX-512 instructions themselves in this book. We recommend either reading Intel documentation online or searching for a number of AVX-512 summaries online (the Wikipedia article on AVX-512 is a good overview which includes AVX-512 extensions supported in Intel Xeon processors too).

> While writing code using AVX-512 intrinsics, we like to open up both the "Intel Intrinsics Guide" and the AVX-512 instruction documentation from Intel on our screens as well.

We highly recommend having two things open on your screen (Fig. 12.11) when you are programming with intrinsics for AVX-512. Links are provided in *For More Information* at the end of this chapter:

- *Intel® Intrinsics Guide* with Knights Landing specific intrinsics selected.
- AVX-512 documentation related to Knights Landing. This has been in the Intel document *Intel® Architecture Instruction Set Extensions Programming Reference* but will eventually transition to *Intel 64 and IA-32 Architectures Software Developer Manuals*.

LEARNING AVX-512 INTRINSICS

In the next section, we present a detailed example of using AVX-512 intrinsics for Knights Landing. It exemplifies the sort of careful coding which can do better than any compiler with very careful data handling. In addition to the example in this chapter, there are additional examples in this book and in prior books:

- Chapter 20 utilizes AVX-512 intrinsics to create a function called `conflict_safe_accumulate` to do a conflict safe gather-modify-scatter force update. The code is shown in Chapter 20. The use of AVX-512CD allows accumulation of forces from multiple lanes with the same neighbor index into a single data lane so that the last value written to memory will contain the correct result.
- Chapter 21 utilizes AVX-512 inline assembly, instead of intrinsics, to ensure the issuing of two FMAs per clock instead of one per clock that was more likely with compiled code for their application. A peek at that code is given in Fig. 12.6.

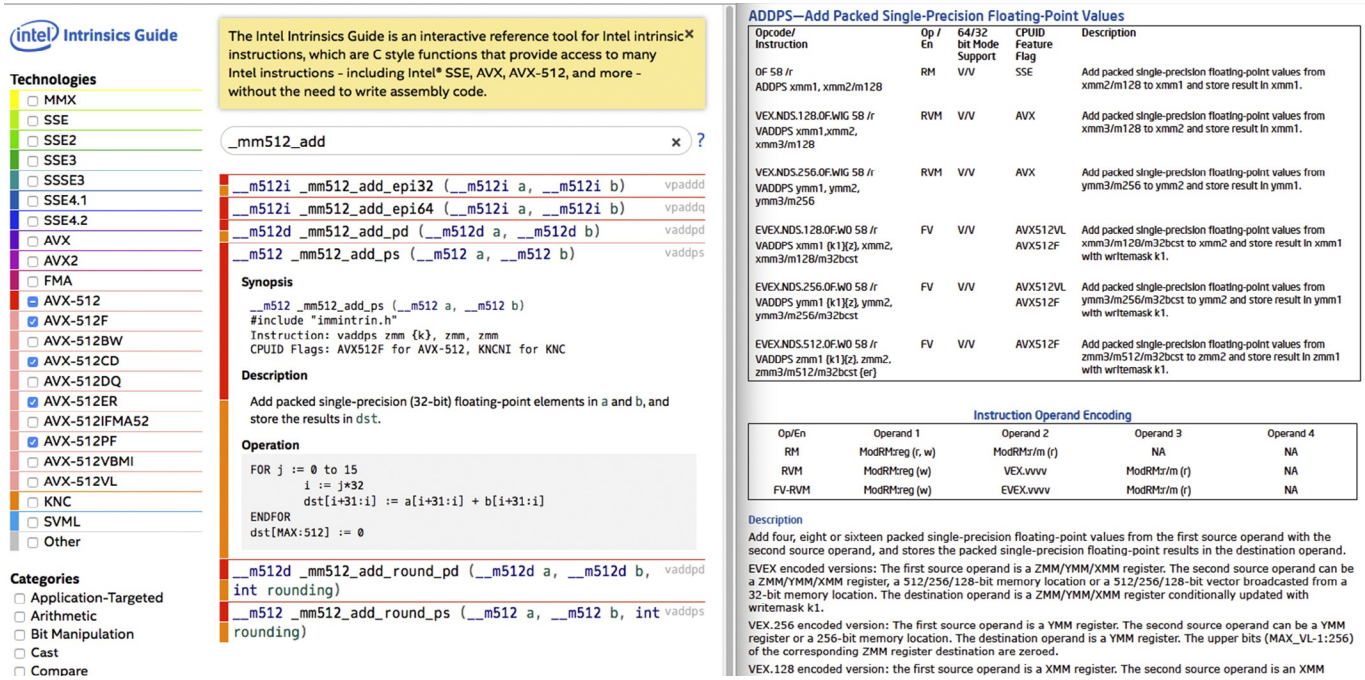


FIG. 12.11

Tip: open both the "Intel Intrinsics Guide" (left) and the AVX-512 instruction documentation (right) any time you write using AVX-512 intrinsics.

- In the Pearls Books, Volumes One and Two, the codes used by 11 chapters used intrinsics. Some used only a few intrinsics to access very specific capabilities, while others had extensive usage of intrinsics. While all those examples were created for Knights Corner, they will generally be source compatible for Knights Landing. See *For More Information* at the end of this chapter for a list of the chapters in Pearls Volumes One and Two which contain Knights Corner examples using intrinsics. Refer to *IMCI to AVX-512* sections, in Chapter 6, for information on changes between AVX-512 and the 512-bit SIMD of Knights Corner coprocessors (IMCI).

STEP-BY-STEP EXAMPLE USING AVX-512 INTRINSICS

In this section, we walk through an excellent example of programming with AVX-512 intrinsics on Knights Landing from a real-life application. This example is compact while illustrating what a creative programmer can do to use the full power of AVX-512.

INTRINSICS FOR MILC

We will look at how and why we used intrinsics in the MILC code. The MILC code was provided by NERSC to be part of the Trinity workload suite (Chapter 25). It represents part of a set of codes written by the MIMD Lattice Computation (MILC) collaboration used to study quantum chromodynamics (QCD), the theory of the strong interactions of subatomic physics.

When we profiled the application, we identified the `mult_adj_su3_mat_4vec` routine to be the key routine to optimize. Reviewing a compiler optimization report, as well as examination of the assembly code, showed that the code is not vectorized well. While the compiler appears to avoid vectorizing due to pointer disambiguation issues, we also see that the compiler did not take advantage of the fact that the real and imaginary parts of complex numbers are packed together. C/C++ compilers do not understand user defined complex data types and thus do not take advantage of data-locality. We decided that the best route to vectorization will be hand optimization and use of intrinsics.

THE DATA TYPES

The `su3_rmd` application data-type is single-precision floating point, and the key user defined data types are:

(1) Complex which is defined in `include\complex.h`:

```
/* generic precision complex number definition */
/* specific for float complex */
typedef struct {
    float real;
    float imag;
} fcomplex;
```

(2) Su3_matrix is an array of 3×3 complex elements. Defined in `include\su3.h`:

```
typedef struct { fcomplex e[3][3]; } fsu3_matrix;
#define su3_matrix   fsu3_matrix
```

(3) Su3_vector is an array of 3 complex elements. Defined in `include\su3.h`:

```
typedef struct { fcomplex c[3]; } fsu3_vector;
#define su3_vector   fsu3_vector
```

The function **mult_adj_su3_mat_4vec** function is defined in `m_amv_4vec.c` in the libraries sub-directory. It multiplies an `su3_vector` by four adjoint `su3_matrices` resulting in four separate `su3_vectors`. The code is shown in Figs. 12.12 and 12.13.

UNDERSTANDING OPERATIONS IN MULT_ADJ_SU3_MAT_4VEC FUNCTION

We see that the `su3_matrix` "a" contains three vectors, and each vector has 3 complex elements. For the loop, the input vector "b" is always constant and hence can be hoisted above the loop, so it is not recomputed for every iteration of the loop. We focus on intraloop vectorization of the "**mult_adj_su3_mat_4vec**" function. We observe that:

(A) Input vector "**src**" also known as vector "b" inside the function is constant.
(B) Input matrix "**mat**" also known as matrix "a" inside the function contains four adjoint matrices. Thus, there is locality here which we can exploit.
(C) Multiplication of complex numbers offers opportunities for reuse as we will see in our example.

We start by arranging to have three 512-bit vectors filled with a single element from the "b" vector as shown in Fig. 12.14. It is interesting to see that we have pairs of intrinsics

```
void mult_adj_su3_mat_4vec(
su3_matrix *mat,
su3_vector *src,
      su3_vector *dest0,
su3_vector *dest1,
      su3_vector *dest2,
su3_vector *dest3 ) {
register int n;
register su3_matrix *a;
register su3_vector *b,*c;
su3_vector *cc[4] ;
cc[0] = dest0 ; cc[1] = dest1 ;
cc[2] = dest2 ; cc[3] = dest3 ;
a = mat ; c=dest0 ; b = src;

for (n=0;n<4;n++,a++,c=cc[n]){
  /* Body of loop is listed in Figure 12.13 */
  }
}
#endif
```

FIG. 12.12

mult_adj_su3_mat_4vec function.

```
br=b->c[0].real; bi=b->c[0].imag;
a0=a->e[0][0].real; a1=a->e[0][1].real; a2=a->e[0][2].real;
c0r = a0*br;  c1r = a1*br;
c2r = a2*br;  c0i = a0*bi;
c1i = a1*bi;  c2i = a2*bi;
a0=a->e[0][0].imag; a1=a->e[0][1].imag; a2=a->e[0][2].imag;
c0r += a0*bi;  c1r += a1*bi;
c2r += a2*bi;  c0i -= a0*br;
c1i -= a1*br;  c2i -= a2*br;

br=b->c[1].real;    bi=b->c[1].imag;
a0=a->e[1][0].real; a1=a->e[1][1].real; a2=a->e[1][2].real;
c0r += a0*br;  c1r += a1*br; c2r += a2*br;
c0i += a0*bi;  c1i += a1*bi;  c2i += a2*bi;
a0=a->e[1][0].imag; a1=a->e[1][1].imag; a2=a->e[1][2].imag;
c0r += a0*bi;  c1r += a1*bi; c2r += a2*bi;
c0i -= a0*br;  c1i -= a1*br;  c2i -= a2*br;

br=b->c[2].real;    bi=b->c[2].imag;
a0=a->e[2][0].real; a1=a->e[2][1].real; a2=a->e[2][2].real;
c0r += a0*br;  c1r += a1*br; c2r += a2*br;
c0i += a0*bi;  c1i += a1*bi;  c2i += a2*bi;
a0=a->e[2][0].imag; a1=a->e[2][1].imag; a2=a->e[2][2].imag;
c0r += a0*bi;  c1r += a1*bi; c2r += a2*bi;
c0i -= a0*br;  c1i -= a1*br;  c2i -= a2*br;

c->c[0].real = c0r;  c->c[0].imag = c0i;
c->c[1].real = c1r;  c->c[1].imag = c1i;
c->c[2].real = c2r;  c->c[2].imag = c2i;
```

FIG. 12.13

Loop body from **mult_adj_su3_mat_4vec**.

```
__m128i z_b_0_ri = _mm_loadu_si64(&(b->c[0]));
__m512 z_b_0_0_ri =
   _mm512_broadcastsd_pd(_mm_castsi128_pd(z_b_0_ri));

__m128i z_b_1_ri = _mm_loadu_si64(&(b->c[1]));
__m512 z_b_1_1_ri =
   _mm512_broadcastsd_pd(_mm_castsi128_pd(z_b_1_ri))

__m128i z_b_2_ri = _mm_loadu_si64(&(b->c[2]));
__m512 z_b_2_2_ri =
   _mm512_broadcastsd_pd(_mm_castsi128_pd(z_b_2_ri))
```

FIG. 12.14

Fill vectors with elements from the b vector.

that actually map into single instructions. This is slightly unusual as most intrinsics map to a single instruction. In this case, Knights Landing is able to combine `_mm_loadu_si64` and `_mm512_broadcastsd_pd` into a single instruction because of AVX-512 embedded broadcast capabilities. As a programmer, we can write portable code while enjoying this optimization by the compiler. Fig. 12.15 shows the contents of the variables as they would appear in ZMM registers or memory with the lane numbers at the top. Grayed cells have unknown values. Logically, `_mm_loadu_si64` loads into "lane 0" and `_mm512_broadcastsd_pd` broadcasts that single value into all 16 lanes.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
z_b_0_ri															b_0^{img}	b_0^{img}
z_b_0_0_ri	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}
z_b_1_ri															b_1^{img}	b_1^{real}
z_b_1_1_ri	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}
z_b_2_ri															b_2^{img}	b_2^{real}
z_b_2_2_ri	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}
z_b_0_0_rr	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}	b_0^{real}
z_b_1_1_rr	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}	b_1^{real}
z_b_2_2_rr	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}	b_2^{real}
z_b_0_0_ii	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}	b_0^{img}
z_b_1_1_ii	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}	b_1^{img}
z_b_2_2_ii	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}	b_2^{img}
z_b_0_0_ir	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}	b_0^{real}	b_0^{img}
z_b_1_1_ir	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}	b_1^{real}	b_1^{img}
z_b_2_2_ir	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}	b_2^{real}	b_2^{img}

FIG. 12.15

Variables (ZMM like) from the initial loads from vector b.

INTRINSICS COMPILE TIME TYPE CHECKING

Notice that the output type of the load is "`si64`" (consult Fig. 12.1) as specified by the suffix on the intrinsic name. The compiler will help us track that type and will not allow the output to be used as a parameter that is not explicitly "`si64`" unless we use an explicit cast. Such explicit casts do not generate instructions and have no cost. They are only required to enable compile time type checking which in turn help catch programming errors. Fig. 12.16 shows that the broadcast intrinsic will expect a packed double, so we

```
__m512d _mm512_broadcastsd_pd (__m128d a)                    vbroadcastsd
```

Synopsis

```
__m512d _mm512_broadcastsd_pd (__m128d a)
#include "immintrin.h"
Instruction: vbroadcastsd zmm {k}, xmm
CPUID Flags: AVX512F
```

Description

Broadcast the low double-precision (64-bit) floating-point element from a to all elements of dst.

FIG. 12.16

Broadcast definition from the Intel Intrinsics Guide (online).

need to cast the `si64` into a packed double type with `_mm_castsi128_pd`. For historic reasons, there is no `_mm_castsi64_pd` even though that might seem to be the perfect cast to choose. Actually, it would seem like `_mm_castsi64_sd` would be needed given what broadcast actually does. Neither `_mm_castsi64_pd` nor `_mm_castsi64_sd` exists as cast operations, so there is a little bit of trial and error figuring out the perfect cast to make the compiler happy. Tables, in Figs. 12.17–12.19, list casting intrinsics that are available. Of course, we are actually moving around single-precision complex numbers and calling them 64-integers (si64) or packed doubles (pd) because C/C++ does not have complex data types built-in.

Cast TO	Cast FROM	Description
32-bit integer	float	_castf32_u32(float)
64-bit integer	double	_castf64_u64(double)
__m128i	__m128d	_mm_castpd_si128(__m128d)
__m128i	__m128	_mm_castps_si128(__m128)
__m128i	__m256i	_mm256_castsi256_si128(__m256i)
__m128i	__m512i	_mm512_castsi512_si128(__m512i)
__m256i	__m256d	_mm256_castpd_si256(__m256d)
__m256i	__m256	_mm256_castps_si256(__m256)
__m256i	__m128i	_mm256_castsi128_si256(__m128i)
__m256i	__m512i	_mm512_castsi512_si256(__m512i)
__m512i	__m512d	_mm512_castpd_si512(__m512d)
__m512i	__m512	_mm512_castps_si512(__m512)
__m512i	__m128i	_mm512_castsi128_si512(__m128i)
__m512i	__m256i	_mm512_castsi256_si512(__m256i)

FIG. 12.17

Casting intrinsics resulting in integers.

Cast TO	Cast FROM	Description
float	unsigned 32-bit integer	_castu32_f32(unsigned__int32)
__m128	__m128d	_mm_castpd_ps(__m128d)
__m128	__m128i	_mm_castsi128_ps(__m128i)
__m128	__m256	_mm256_castps256_ps128(__m256)
__m128	__m512	_mm512_castps512_ps128(__m512)
__m256	__m256d	_mm256_castpd_ps(__m256d)
__m256	__m256i	_mm256_castsi256_ps(__m256i)
__m256	__m128	_mm256_castps128_ps256(__m128)
__m256	__m512	_mm512_castps512_ps256(__m512)
__m512	__m512d	_mm512_castpd_ps(__m512d)
__m512	__m512i	_mm512_castsi512_ps(__m512i)
__m512	__m128	_mm512_castps128_ps512(__m128)
__m512	__m256	_mm512_castps256_ps512(__m256)

FIG. 12.18

Casting intrinsics resulting in floats.

Cast TO	Cast FROM	Description
double	unsigned 64-bit integer	_castu64_f64(unsigned__int64)
__m128d	__m128	_mm_castps_pd(__m128)
__m128d	__m128i	_mm_castsi128_pd(__m128i)
__m128d	__m256d	_mm256_castpd256_pd128(__m256d)
__m128d	__m512d	_mm512_castpd512_pd128(__m512d)
__m256d	__m256	_mm256_castps_pd(__m256)
__m256d	__m256i	_mm256_castsi256_pd(__m256i)
__m256d	__m128d	_mm256_castpd128_pd256(__m128d)
__m256d	__m512d	_mm512_castpd512_pd256(__m512d)
__m512d	__m512	_mm512_castps_pd(__m512)
__m512d	__m512i	_mm512_castsi512_pd(__m512i)
__m512d	__m128d	_mm512_castpd128_pd512(__m128d)
__m512d	__m256d	_mm512_castpd256_pd512(__m256d)

FIG. 12.19

Casting intrinsics resulting in doubles.

```
/* use lanes 2,2,0,0 to get only the real parts */
__m512 z_b_0_0_rr = _mm512_permute_ps(z_b_0_0_ri, 0b10100000);
__m512 z_b_1_1_rr = _mm512_permute_ps(z_b_1_1_ri, 0b10100000);
__m512 z_b_2_2_rr = _mm512_permute_ps(z_b_2_2_ri, 0b10100000);

/* use lanes 3,3,1,1 to get only the imaginary parts */
__m512 z_b_0_0_ii = _mm512_permute_ps(z_b_0_0_ri, 0b11110101);
__m512 z_b_1_1_ii = _mm512_permute_ps(z_b_1_1_ri, 0b11110101);
__m512 z_b_2_2_ii = _mm512_permute_ps(z_b_2_2_ri, 0b11110101);

/* use lanes 2,3,0,1 to swap the real and imaginary parts */
__m512 z_b_0_0_ir = _mm512_permute_ps(z_b_0_0_ri,0b10110001);
__m512 z_b_1_1_ir = _mm512_permute_ps(z_b_1_1_ri,0b10110001);
__m512 z_b_2_2_ir = _mm512_permute_ps(z_b_2_2_ri,0b10110001);
```

FIG. 12.20

Create permutations of values from b which will be reused.

Previously, we mentioned with our observation C that there are opportunities for reuse. Fig. 12.20 has code to create values we can reuse. The `_mm512_permute_ps` intrinsics permute values between lanes within each set of four lanes. For instance, the first group of permutes seeks to end up only with the real components in all lanes. The intrinsic outputs a value in lane 1 with the input value from lane 0. Likewise, the intrinsic will output values in lane 3 from the input value in lane 2. A permutation is specified via four numbers each 0–4 (i.e., 00, 01, 10, or 11) in the specified value `0b10100000`. This simply means that lanes 3, 2, 1, 0 will get the values from lanes 2, 2, 0, 0, respectively. This same logic is repeated for the other groups of four lanes using the same math but +4, +8, and +12.

We are ready to load the "a" Matrices (we label them as M0,M1,M2,M3 in our figures). There are four "a" matrices that the function (Fig. 12.12) steps through. Each matrix is 3×3 in size and therefore contains three vectors each with three

complex numbers. We need the three vectors from the matrix arranged (effectively an SoA to AoS conversion) into three different "ZMM" registers to prepare for the mulitplications with vector "b." A vector of three complex numbers is actually only 192 bits in size (32 bits/single × 2 singles/complex × 3 complex/vector). This means that such a short vector can fit entirely within a 256 bit "lane." This arrangement is shown in Fig. 12.21.

To achieve the packing shown in Fig. 12.21, for a given matrix we will first load two of the vectors into a ZMM register/variable, use an `alignr` instruction on the original register to get the second vector into upper 256b lane, and finally merge the registers from the first and second operations such that we have the first vector in the low lane and the second vector in the high lane. Using the `alignr` with identical inputs allows us to effectively do a rotate operation. The code for the first High-256b/Low-256b pair is shown in Fig. 12.22, and additional code (not shown) uses the same sequence on different data. The complete code is freely available online, see *For More Information* at the end of this chapter. Fig. 12.23 shows the resulting values, and the grayed cells indicate cells that were loaded but we do not care about. We could have used a masked load to limit the loading in which case those values would have been left over from the prior contents of the register. A masked load would better imitate the original code by not accessing data beyond the last input matrix. In this case, the programmer was not concerned that the extra reach was an issue. Either way, the values are of no interest to us even though they will participate in future mathematical operations on the full 512-bit register.

"ZMM" variable	High-256b	Low-256b
z_a_01_00	$M0_{1,0..2}$	$M0_{0,0..2}$
z_a_10_02	$M1_{0,0..2}$	$M0_{2,0..2}$
z_a_12_11	$M1_{2,0..2}$	$M1_{1,0..2}$
z_a_21_20	$M2_{1,0..2}$	$M2_{0,0..2}$
z_a_30_22	$M3_{0,0..2}$	$M2_{2,0..2}$
z_a_32_31	$M3_{2,0..2}$	$M3_{1,0..2}$

FIG. 12.21

Packing of 36 complex numbers into six 512-bit "ZMM" variables.

```
__m512 z_a_01_00_012_ri_f =
  _mm512_loadu_ps ((const float *) &(mat0->e[0][0]));
__m512 z_a_01_00_012_ri_f_hi2lo =
  _mm512_alignr_epi32(_mm512_castps_si512(z_a_01_00_012_ri_f),
                      _mm512_castps_si512(z_a_01_00_012_ri_f),
                      14);
__m512 z_a_01_00_012_ri =
  _mm512_mask_mov_pd(z_a_01_00_012_ri_f,
                     0xF0,
                     z_a_01_00_012_ri_f_hi2lo);
```

FIG. 12.22

Loading z_a_01_00 according to Fig. 12.21.

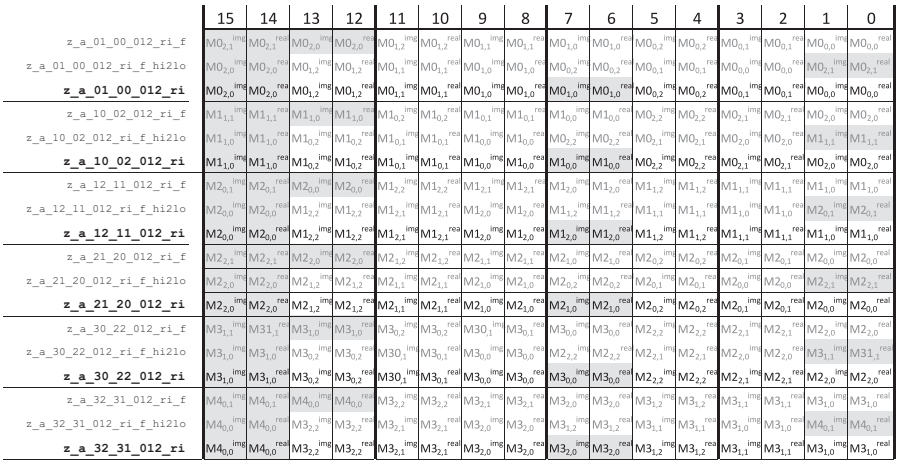


FIG. 12.23

Loading all ZMMs according to Fig. 12.21.

Next, we perform a transpose from AoS to SoA so that they are ready to multiply by vector "b." This clever arrangement fits within a single 256-bit lane and makes great use of the AVX-512. The code is shown in Fig. 12.24, and the step-by-step values in the ZMM registers are shown in Fig. 12.25. Next, for Complex multiply, we need the real and imaginary components swapped as shown in Fig. 12.26 with results shown in Fig. 12.27.

Now we are ready to compute the vector-vector complex multiplies using a vector from a matrix (M0…M3) and a vector derived from the "b" vector. Note that the real and imaginary components are computed as:

Imaginary Part	Real
M.real * b.imag	M.imag * b.imag
minus	***plus***
M.imag * b.real	M.real * b.real

We can make use of an FMA instruction called "`vfmsubadd`" which does subtractions in the odd lanes and additions in the even lanes. The instruction documentation will note that there are three versions of the instruction, to allow different orders and types of operands, namely, `vfmsubadd132ps`, `vfmsubadd213ps`, and `vfmsubadd231ps`. Since we are using intrinsics, we can ignore the subtleties of operand ordering and memory verses register instructions and simply focus on "`fmsubadd`" with the intrinsic `_mm512_fmsubadd_ps` as shown in Fig. 12.28 with results shown in Fig. 12.29.

```
__m512 z_a_10_00_012_ri =
    _mm512_mask_mov_pd(z_a_01_00_012_ri,
                       _mm512_int2mask(0xF0),
                       z_a_10_02_012_ri);
__m512 z_a_01_11_012_ri_tmp =
    _mm512_mask_mov_pd(z_a_12_11_012_ri,
                       _mm512_int2mask(0xF0),
                       z_a_01_00_012_ri);
__m512 z_a_11_01_012_ri =
    _mm512_shuffle_f32x4(z_a_01_11_012_ri_tmp,
                         z_a_01_11_012_ri_tmp,
                         0b01001110);
__m512 z_a_12_02_012_ri =
   _mm512_mask_mov_pd(z_a_10_02_012_ri,
                      _mm512_int2mask(0xF0),
                      z_a_12_11_012_ri);
__m512 z_a_30_20_012_ri =
   _mm512_mask_mov_pd(z_a_21_20_012_ri,
                      _mm512_int2mask(0xF0),
                      z_a_30_22_012_ri);
__m512 z_a_21_31_012_ri_tmp =
   _mm512_mask_mov_pd(z_a_32_31_012_ri,
                      _mm512_int2mask(0xF0),
                      z_a_21_20_012_ri);
__m512 z_a_31_21_012_ri =
   _mm512_shuffle_f32x4(z_a_21_31_012_ri_tmp,
                        z_a_21_31_012_ri_tmp,
                        0b01001110);
__m512 z_a_32_22_012_ri =
   _mm512_mask_mov_pd(z_a_30_22_012_ri,
                      _mm512_int2mask(0xF0),
                      z_a_32_31_012_ri);
```

FIG. 12.24

AoS to SoA transformation via intrinsics.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
z_a_10_00_012_ri	z_a_**10**_02_012_ri								z_a_01_**11**_012_ri							
z_a_10_00_012_ri	$M1_{1,0}^{img}$	$M1_{1,0}^{real}$	$M1_{0,2}^{img}$	$M1_{0,2}^{real}$	$M1_{0,1}^{img}$	$M1_{0,1}^{real}$	$M1_{0,0}^{img}$	$M1_{0,0}^{real}$	$M0_{1,0}^{img}$	$M0_{1,0}^{real}$	$M0_{0,2}^{img}$	$M0_{0,2}^{real}$	$M0_{0,1}^{img}$	$M0_{0,1}^{real}$	$M0_{0,0}^{img}$	$M0_{0,0}^{real}$
z_a_01_11_012_ri_tmp	z_a_**01**_00_012_ri								z_a_12_**11**_012_ri							
z_a_01_11_012_ri_tmp	$M0_{2,0}^{img}$	$M0_{2,0}^{real}$	$M0_{1,2}^{img}$	$M0_{1,2}^{real}$	$M0_{1,1}^{img}$	$M0_{1,1}^{real}$	$M0_{1,0}^{img}$	$M0_{1,0}^{real}$	$M1_{2,0}^{img}$	$M1_{2,0}^{real}$	$M1_{1,2}^{img}$	$M1_{1,2}^{real}$	$M1_{1,1}^{img}$	$M1_{1,1}^{real}$	$M1_{1,0}^{img}$	$M1_{1,0}^{real}$
z_a_11_01_012_ri	$M1_{2,0}^{img}$	$M1_{2,0}^{real}$	$M1_{1,2}^{img}$	$M1_{1,2}^{real}$	$M1_{1,1}^{img}$	$M1_{1,1}^{real}$	$M1_{1,0}^{img}$	$M1_{1,0}^{real}$	$M0_{2,0}^{img}$	$M0_{2,0}^{real}$	$M0_{1,2}^{img}$	$M0_{1,2}^{real}$	$M0_{1,1}^{img}$	$M0_{1,1}^{real}$	$M0_{1,0}^{img}$	$M0_{1,0}^{real}$
z_a_12_02_012_ri	z_a_**12**_11_012_ri								z_a_10_**02**_012_ri							
z_a_12_02_012_ri	$M2_{0,0}^{img}$	$M2_{0,0}^{real}$	$M1_{2,2}^{img}$	$M1_{2,2}^{real}$	$M1_{2,1}^{img}$	$M1_{2,1}^{real}$	$M1_{2,0}^{img}$	$M1_{2,0}^{real}$	$M1_{0,0}^{img}$	$M1_{0,0}^{real}$	$M0_{0,2}^{img}$	$M0_{0,2}^{real}$	$M0_{0,1}^{img}$	$M0_{0,1}^{real}$	$M0_{0,0}^{img}$	$M0_{0,0}^{real}$
z_a_30_20_012_ri	z_a_**30**_22_012_ri								z_a_21_**20**_012_ri							
z_a_30_20_012_ri	$M3_{1,0}^{img}$	$M3_{1,0}^{real}$	$M3_{0,2}^{img}$	$M3_{0,2}^{real}$	$M30_{1}^{img}$	$M3_{0,1}^{real}$	$M3_{0,0}^{img}$	$M3_{0,0}^{real}$	$M2_{1,0}^{img}$	$M2_{1,0}^{real}$	$M2_{0,2}^{img}$	$M2_{0,2}^{real}$	$M2_{0,1}^{img}$	$M2_{0,1}^{real}$	$M2_{0,0}^{img}$	$M2_{0,0}^{real}$
z_a_21_31_012_ri_tmp	z_a_**21**_20_012_ri								z_a_32_**31**_012_ri							
z_a_21_31_012_ri_tmp	$M2_{2,0}^{img}$	$M2_{2,0}^{real}$	$M2_{1,2}^{img}$	$M2_{1,2}^{real}$	$M2_{1,1}^{img}$	$M2_{1,1}^{real}$	$M2_{1,0}^{img}$	$M2_{1,0}^{real}$	$M3_{2,0}^{img}$	$M3_{2,0}^{real}$	$M3_{1,2}^{img}$	$M3_{1,2}^{real}$	$M3_{1,1}^{img}$	$M3_{1,1}^{real}$	$M3_{1,0}^{img}$	$M3_{1,0}^{real}$
z_a_31_21_012_ri	$M3_{2,0}^{img}$	$M3_{2,0}^{real}$	$M3_{1,2}^{img}$	$M3_{1,2}^{real}$	$M3_{1,1}^{img}$	$M3_{1,1}^{real}$	$M3_{1,0}^{img}$	$M3_{1,0}^{real}$	$M2_{2,0}^{img}$	$M2_{2,0}^{real}$	$M2_{1,2}^{img}$	$M2_{1,2}^{real}$	$M2_{1,1}^{img}$	$M2_{1,1}^{real}$	$M2_{1,0}^{img}$	$M2_{1,0}^{real}$
z_a_32_22_012_ri	z_a_**32**_31_012_ri								z_a_30_**22**_012_ri							
z_a_32_22_012_ri	$M4_{0,0}^{img}$	$M4_{0,0}^{real}$	$M3_{2,2}^{img}$	$M3_{2,2}^{real}$	$M3_{2,1}^{img}$	$M3_{2,1}^{real}$	$M3_{2,0}^{img}$	$M3_{2,0}^{real}$	$M3_{0,0}^{img}$	$M3_{0,0}^{real}$	$M2_{2,2}^{img}$	$M2_{2,2}^{real}$	$M2_{2,1}^{img}$	$M2_{2,1}^{real}$	$M2_{2,0}^{img}$	$M2_{2,0}^{real}$

FIG. 12.25

AoS to SoA transformations in the ZMM registers.

```
__m512 z_a_10_00_012_ir =
  _mm512_permute_ps(z_a_10_00_012_ri, 0b10110001);
__m512 z_a_11_01_012_ir =
  _mm512_permute_ps(z_a_11_01_012_ri, 0b10110001);
__m512 z_a_12_02_012_ir =
  _mm512_permute_ps(z_a_12_02_012_ri, 0b10110001);

__m512 z_a_30_20_012_ir =
  _mm512_permute_ps(z_a_30_20_012_ri, 0b10110001);
__m512 z_a_31_21_012_ir =
  _mm512_permute_ps(z_a_31_21_012_ri, 0b10110001);
__m512 z_a_32_22_012_ir =
  _mm512_permute_ps(z_a_32_22_012_ri, 0b10110001);
```

FIG. 12.26

Swapping the real and imaginary parts.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
z_a_10_00_012_ir	$M1_{1,0}^{img}$	$M1_{1,0}^{rea}$	$M1_{0,2}^{rea}$	$M1_{0,2}^{img}$	$M1_{0,1}^{rea}$	$M1_{0,1}^{img}$	$M1_{0,0}^{rea}$	$M1_{0,0}^{img}$	$M0_{1,0}^{rea}$	$M0_{1,0}^{img}$	$M0_{0,2}^{rea}$	$M0_{0,2}^{img}$	$M0_{0,1}^{rea}$	$M0_{0,1}^{img}$	$M0_{0,0}^{rea}$	$M0_{0,0}^{img}$
z_a_11_01_012_ir	$M1_{2,0}^{img}$	$M1_{2,0}^{rea}$	$M1_{1,2}^{rea}$	$M1_{1,2}^{img}$	$M1_{1,1}^{rea}$	$M1_{1,1}^{img}$	$M1_{1,0}^{rea}$	$M1_{1,0}^{img}$	$M0_{2,0}^{rea}$	$M0_{2,0}^{img}$	$M0_{1,2}^{rea}$	$M0_{1,2}^{img}$	$M0_{1,1}^{rea}$	$M0_{1,1}^{img}$	$M0_{1,0}^{rea}$	$M0_{1,0}^{img}$
z_a_12_02_012_ir	$M2_{0,0}^{img}$	$M2_{0,0}^{real}$	$M1_{2,2}^{real}$	$M1_{2,2}^{img}$	$M1_{2,1}^{rea}$	$M1_{2,1}^{img}$	$M1_{2,0}^{real}$	$M1_{2,0}^{img}$	$M1_{0,0}^{real}$	$M1_{0,0}^{img}$	$M0_{2,2}^{real}$	$M0_{2,2}^{img}$	$M0_{2,1}^{real}$	$M0_{2,1}^{img}$	$M0_{2,0}^{real}$	$M0_{2,0}^{img}$
z_a_30_20_012_ir	$M3_{1,0}^{real}$	$M3_{1,0}^{img}$	$M3_{0,2}^{real}$	$M3_{0,2}^{img}$	$M3_{0,1}^{real}$	$M30_{1}^{img}$	$M3_{0,0}^{real}$	$M3_{0,0}^{img}$	$M2_{1,0}^{real}$	$M2_{1,0}^{img}$	$M2_{0,2}^{real}$	$M2_{0,2}^{img}$	$M2_{0,1}^{real}$	$M2_{0,1}^{img}$	$M2_{0,0}^{real}$	$M2_{0,0}^{img}$
z_a_31_21_012_ir	$M3_{2,0}^{real}$	$M3_{2,0}^{img}$	$M3_{1,2}^{real}$	$M3_{1,2}^{img}$	$M3_{1,1}^{real}$	$M3_{1,1}^{img}$	$M3_{1,0}^{real}$	$M3_{1,0}^{img}$	$M2_{2,0}^{real}$	$M2_{2,0}^{img}$	$M2_{1,2}^{real}$	$M2_{1,2}^{img}$	$M2_{1,1}^{real}$	$M2_{1,1}^{img}$	$M2_{1,0}^{real}$	$M2_{1,0}^{img}$
z_a_32_22_012_ir	$M4_{0,0}^{real}$	$M4_{0,0}^{img}$	$M3_{2,2}^{real}$	$M3_{2,2}^{img}$	$M3_{2,1}^{real}$	$M3_{2,1}^{img}$	$M3_{2,0}^{real}$	$M3_{2,0}^{img}$	$M3_{0,0}^{real}$	$M3_{0,0}^{img}$	$M2_{2,2}^{real}$	$M2_{2,2}^{img}$	$M2_{2,1}^{real}$	$M2_{2,1}^{img}$	$M2_{2,0}^{real}$	$M2_{2,0}^{img}$

FIG. 12.27

The real and imaginary parts are now swapping in the ZMM registers.

```
__m512 z_a_10_00_X_b_0_0_012_arXbr_aiXbr_ri =
  _mm512_mul_ps(z_a_10_00_012_ri, z_b_0_0_rr);
__m512 z_a_11_01_X_b_1_1_012_arXbr_aiXbr_ri =
  _mm512_mul_ps(z_a_11_01_012_ri, z_b_1_1_rr);
__m512 z_a_12_02_X_b_2_2_012_arXbr_aiXbr_ri =
  _mm512_mul_ps(z_a_12_02_012_ri, z_b_2_2_rr);
__m512 z_a_30_20_X_b_0_0_012_arXbr_aiXbr_ri =
  _mm512_mul_ps(z_a_30_20_012_ri, z_b_0_0_rr);
__m512 z_a_31_21_X_b_1_1_012_arXbr_aiXbr_ri =
  _mm512_mul_ps(z_a_31_21_012_ri, z_b_1_1_rr);
__m512 z_a_32_22_X_b_2_2_012_arXbr_aiXbr_ri =
  _mm512_mul_ps(z_a_32_22_012_ri, z_b_2_2_rr);
__m512 z_a_10_00_X_b_0_0_012_aiXbiPLarXbr_arXbiMIaiXbr_ri =
  _mm512_fmsubadd_ps(z_a_10_00_012_ir, z_b_0_0_ii,
                     z_a_10_00_X_b_0_0_012_arXbr_aiXbr_ri);
__m512 z_a_11_01_X_b_1_1_012_aiXbiPLarXbr_arXbiMIaiXbr_ri =
  _mm512_fmsubadd_ps(z_a_11_01_012_ir, z_b_1_1_ii,
                     z_a_11_01_X_b_1_1_012_arXbr_aiXbr_ri);
__m512 z_a_12_02_X_b_2_2_012_aiXbiPLarXbr_arXbiMIaiXbr_ri =
  _mm512_fmsubadd_ps(z_a_12_02_012_ir, z_b_2_2_ii,
                     z_a_12_02_X_b_2_2_012_arXbr_aiXbr_ri);
__m512 z_a_30_20_X_b_0_0_012_aiXbiPLarXbr_arXbiMIaiXbr_ri =
  _mm512_fmsubadd_ps(z_a_30_20_012_ir, z_b_0_0_ii,
                     z_a_30_20_X_b_0_0_012_arXbr_aiXbr_ri);
__m512 z_a_31_21_X_b_1_1_012_aiXbiPLarXbr_arXbiMIaiXbr_ri =
  _mm512_fmsubadd_ps(z_a_31_21_012_ir, z_b_1_1_ii,
                     z_a_31_21_X_b_1_1_012_arXbr_aiXbr_ri);
__m512 z_a_32_22_X_b_2_2_012_aiXbiPLarXbr_arXbiMIaiXbr_ri =
  _mm512_fmsubadd_ps(z_a_32_22_012_ir, z_b_2_2_ii,
                     z_a_32_22_X_b_2_2_012_arXbr_aiXbr_ri);
```

FIG. 12.28

Intrinsics for compute operation using “vfmsubadd.”

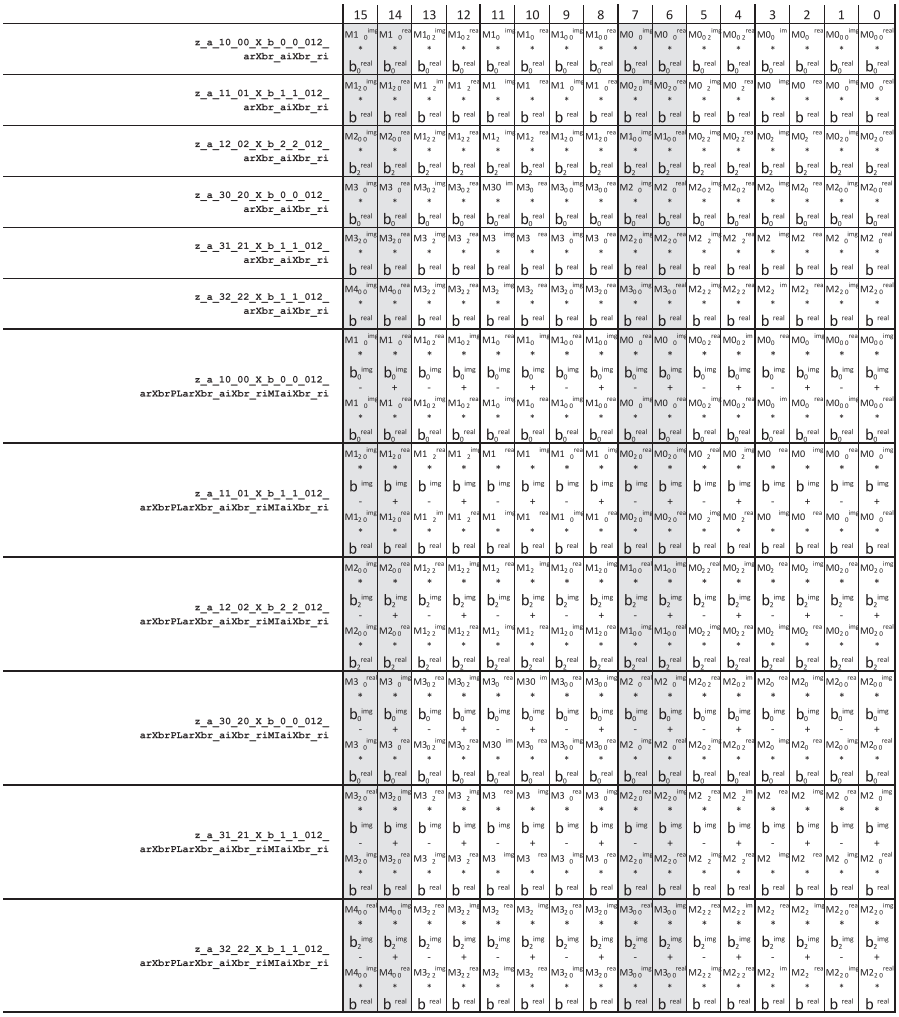


FIG. 12.29

ZMM registers as we do the compute operation using "vfmsubadd."

Finally, we need to add across the vectors to get the results as shown in the code in Fig. 12.30 and store the results in the four distinct vectors. Note that each result vector has three complex numbers in it. We need to limit our stores, via masking, to only write the lowest six single-precision elements hence a mask with only the lower six bits on (i.e., 0x003F). Since half the resulting vectors are in the upper 256-bits of the ZMM registers, we use `_mm512_extractf64x4_pd` to move them into the lower 256-bits before storing. The code is shown in Fig. 12.31.

```
__m512 z_c_1_0_012_ri =
  _mm512_add_ps(
    z_a_10_00_X_b_0_0_012_aiXbiPLarXbr_arXbiMIaiXbr_ri,
    z_a_11_01_X_b_1_1_012_aiXbiPLarXbr_arXbiMIaiXbr_ri);
__m512 z_c_3_2_012_ri =
  _mm512_add_ps(
    z_a_30_20_X_b_0_0_012_aiXbiPLarXbr_arXbiMIaiXbr_ri,
    z_a_31_21_X_b_1_1_012_aiXbiPLarXbr_arXbiMIaiXbr_ri);
z_c_1_0_012_ri =
  _mm512_add_ps(
    z_c_1_0_012_ri,
    z_a_12_02_X_b_2_2_012_aiXbiPLarXbr_arXbiMIaiXbr_ri);
z_c_3_2_012_ri =
  _mm512_add_ps(
    z_c_3_2_012_ri,
    z_a_32_22_X_b_2_2_012_aiXbiPLarXbr_arXbiMIaiXbr_ri);
```

FIG. 12.30

Add across the vectors to get the results.

```
_mm512_mask_storeu_ps((float *) &(dest0->c[0]),
                      0x003F,
                      z_c_1_0_012_ri);

__m512 z_c_X_1_012_ri =
  _mm512_castpd256_pd512(
    _mm512_extractf64x4_pd( z_c_1_0_012_ri, 1));
_mm512_mask_storeu_ps((float *) &(dest1->c[0]),
                      0x003F,
                      z_c_X_1_012_ri);

_mm512_mask_storeu_ps((float *) &(dest2->c[0]),
                      0x003F,
                      z_c_3_2_012_ri);

__m512 z_c_X_3_012_ri =
  _mm512_castpd256_pd512(
    _mm512_extractf64x4_pd( z_c_3_2_012_ri, 1));
_mm512_mask_storeu_ps((float *) &(dest3->c[0]),
                      0x003F,
                       z_c_X_3_012_ri);
```

FIG. 12.31

Write out the destination vectors.

RESULTS USING OUR INTRINSICS CODE

Our vectorization strategy focused on intraloop vectorization with operations per site versus interloop vectorization. We restricted a vector to a single 256-bit lane. Considering that our vector size is three single-precision complex numbers that means we use six single-precision floating-point elements in each 256-bit lane. That leaves 2 lanes unused out of 8, or 4 out of 16 considering the full 512-bit width. Therefore, our vector efficiency is 75% (12/16). Nevertheless, we have been able to effectively vectorize the matrix-vector multiplication.

```
//ORIG Code
mpiexec -n 1 -env OMP_NUM_THREADS 1 -env KMP_AFFINITY verbose
./su3_rmd.xMIC-AVX512.Orig.timer < ../n8_single.4x4x4x4.in
CONGRAD5: time = 1.179695e-02 (fn F) masses = 1 iters = 68
mflops = 1.751579e+03   //Last CG results with largest
iteration count
Time spent in dslash_fn_field_special function,
mult_adj_su3_mat_4vec loop = 4.008532e-03 //corresponding time
spent in dslash_fn_field_special for above CG operation

//VECT Code
mpiexec -n 1 -env OMP_NUM_THREADS 1 -env KMP_AFFINITY verbose
./su3_rmd.xMIC-AVX512.INTRIN.m_amv_4vec.timer <
../n8_single.4x4x4x4.in
CONGRAD5: time = 9.550095e-03 (fn F) masses = 1 iters = 68
mflops = 2.163674e+03
Time spent in dslash_fn_field_special function,
mult_adj_su3_mat_4vec loop = 1.979828e-03
```

FIG. 12.32

Running the original and the vectorized code.

The code and detailed instructions are available online—see *For More Information* at the end of this chapter.

We ran the application on a Knights Landing processor with 64 cores, 1.3 GHz, 16GB MCDRAM (in cache mode) and 96GB DRAM. The application was set to run for a 4^4 lattice and run with one MPI rank which was bound to a single core. Timings, shown in Fig. 12.32, comparing the original code to vectorized code shows that we gained 2.02× (**4.008532/1.979828**) from our vectorization of the `dslash_fn_field_special` within that function. Overall, our Vectorization optimization in `dslash_fn_field_special` function helped improve time spent in CG by 1.24× (**.01179695/.009550095**) resulting in 1.24× (**2.163674/1.751579**) gains in CG FLOPS. CG is slowed by other similar hotspots. Note that, with the 4^4 lattice, the problem does not entirely fit within the MCDRAM, so additional work on tiling or prefetching would likely help as well.

FOR MORE INFORMATION

Here are some additional reading materials that we recommend related to this chapter.

- Intel® Intrinsics Guide with Knights Landing specific intrinsics selected, http://lotsofcores.com/KNLintrinsics.
- Intel® 64 and IA-32 Architectures Software Developer Manuals, http://lotsofcores.com/Intel_ISA1.
- Intel® Architecture Instruction Set Extensions Programming Reference, http://lotsofcores.com/Intel_ISA2.
- Intel® Software Development Emulator (Intel® SDE), http://lotsofcores.com/Intel_SDE.
- Data Alignment to Assist Vectorization, https://software.intel.com/articles/data-alignment-to-assist-vectorization.

- Intel Xeon Phi Users Group, Vectorization Working Group, repository of some interesting vectorization examples, https://github.com/IXPUG/WG_Vectorization/.
- A methodology for tuning prefetching: High-Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, Chapter 21—Prefetch Tuning Optimizations, ISBN 978-0-12-803819-2.
- How to detect Knights Landing AVX-512 support (Intel Xeon Phi processor), https://software.intel.com/articles/how-to-detect-knl-instruction-support.
- Links related to inline assembly code (__asm__)
  - Links for code from Chapter 21 showing inline assembly code
    - The version of SeisSol which was used to produce the results of Chapter 21 can be found on github: https://github.com/SeisSol/SeisSol/releases/tag/201511.
    - The assembly language kernels used in those SeisSol kernels are generated by the LIBXSMM open source project which is hosted at github: https://github.com/hfp/libxsmm.
  - Links related to learning __asm__ coding
    - GCC Inline Assembly Tutorial, www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html.
    - GCC documentation on Extended Asm, https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html.
- Chapters from the "Pearls" book with Intel Xeon Phi coprocessor intrinsic usage (often source compatible with AVX-512).
  - High-Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, ISBN 978-0-12-803819-2.
    - Chapter 4—Pairwise DNA Sequence Alignment Optimization.
    - Chapter 9—Wilson Dslash Kernel From Lattice QCD Optimization.
    - Chapter 19—OpenCL: There and Back Again.
    - Chapter 24—Portable Explicit Vectorization Intrinsics.
  - High-Performance Parallelism Pearls: Multicore and Many-core Programming Approaches, ISBN 978-0-12-802118-7.
    - Chapter 2—From "Correct" to "Correct & Efficient": A Hydro2D Case Study with Godunov's Scheme.
    - Chapter 8—Optimizing Gather/Scatter Patterns.
    - Chapter 10—*N*-Body Methods.
    - Chapter 21—High-Performance Ray Tracing.
    - Chapter 23—Characterization and Optimization Methodology Applied to Stencil Computations.
    - Chapter 27—Sparse Matrix-Vector Multiplication: Parallelization and Vectorization.
    - Chapter 28—Morton Order Improves Performance.
- MILC benchmark code, provided by NERSC, is a part of the Trinity workload suite and using the code written by MILC collaboration, http://lotsofcores.com/milc.
- Download the code from this and other chapters, http://lotsofcores.com.

CHAPTER

Performance libraries 13

INTEL PERFORMANCE LIBRARY OVERVIEW

What is new with Knights Landing in this chapter?
MCDRAM support, IPP and DAAL libraries, AVX-512 support

There are three libraries from Intel, collectively referred to as the Intel® Performance Libraries, aimed at supplying high-performance versions of various important computations. These libraries are available free to anyone as community-licensed libraries, and are also supplied as commercially supported libraries included within Intel products such as Intel® Parallel Studio. There is no difference in functionality between the free and the commercially supported versions. Because these libraries offer cross Intel® Architecture platform compatibility, Knights Landing can utilize each of them. Intel is constantly releasing library support for new microarchitectures; Intel has endowed these libraries with Knights Landing optimizations including support for Intel® Advanced Vector eXtensions 512-bit (AVX-512) vector instructions.

These libraries are often used within other software packages as the core acceleration for key routines. Math libraries from other vendors, such as IMSL Numerical Libraries from Rogue Wave software and the NAG Library from the Numerical Algorithms Group Ltd., also provide support for Intel® Xeon Phi products. Both of these libraries integrate portions of the Intel® Math Kernel Library (MKL) as well to provide optimizations of highly computationally intensive functions. The Anaconda Python distribution, created by Continuum Analytics, uses MKL for speed-ups they have described as "up to 7 ×." There are also versions of R, including Revolution Analytics/Microsoft R Open, which use MKL to accelerate core capabilities.

The three Intel Performance Libraries are:

- Intel® Math Kernel Library (MKL) — mathematical functions including Basic Linear Algebra Subprograms (BLAS), fast Fourier Transform (FFT), equation solvers, etc.,

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00013-2

- Intel® Data Analytics Acceleration Library (DAAL) — functions to accelerate big data processing in a big data context (e.g., use with Spark, Hadoop, etc.), and
- Intel® Integrated Performance Primitives (IPP) — functions for image, media, communication and signal processing, data compression and encryption.

Libraries offer a very attractive option for using the power of Knights Landing, quickly and effectively. Using these libraries enables an application to be optimized for many different processor microarchitectures, including Intel® Xeon® processors and Intel Xeon Phi processors. For "offload" style programming, compiler support via pragmas (C/C++ OpenMP) or directives (Fortran OpenMP) allows code including library usage to be offloaded from a processor to either a Knights Landing coprocessor (e.g., a coprocessor card), or to a Knights Landing elsewhere in the system via offload-over-fabric (Chapter 18). Additionally, the OpenMP-threaded MKL supports an "Automatic Offload" (AO) capability for the offloading to be chosen within MKL itself. Fig. 13.1 summarizes these usage options for the libraries.

A particular optimized routine is generally only in one of the three libraries in order to avoid too much overlap in functionality, therefore the division between these libraries can be a little arbitrary and you are likely to use two or three libraries in many applications. For instance, most applications doing Big Data work will use both MKL and DAAL.

Model	Description
Native	Library functions called from within an application executing natively on an Intel Xeon Phi processor or coprocessor. This is simply "normal" usage as expected with any processor
Offload, Automatic (AO)	A special capability of MKL. Select MKL functions are automatically distributed to run across the processor(s) and Intel Xeon Phi coprocessor(s) when present including the automatic control of data movement to/from the coprocessor(s). This mode is enabled and more finely controlled where desirable with simple service functions (or environment variables) and supports calling from both C/C++ and Fortran
Offload, Compiler Assisted (CAO)	Library functions called from within compiler offload regions (see Chapter 18) run on Intel Xeon Phi target(s) when present (or on the host(s) otherwise). Compiler offload pragmas (C/C++) and directives (Fortran) control data movement to/from the target(s). This approach supports offloading regions of code potentially comprising of numerous calls to library functions, thereby amortizing the data movement over multiple computation operations

FIG. 13.1

Libraries can be used normal (native) or in offloading situations.

If an application uses any of these interfaces, or could use them, then we should definitely look at library solutions. The rest of this chapter provides further details on how to use Intel Performance Libraries on Knights Landing. For details on the functions themselves, within the libraries, you are encouraged to read the documentation (available online) to exploit the algorithms in each library and to learn their usage details and linking options (see the "For More Information" section at the end of this chapter).

INTEL MATH KERNEL LIBRARY OVERVIEW

The Intel Math Kernel Library (MKL) includes routines and functions optimized for Intel and compatible processor-based computers running Windows, OS X, and Linux-compatible operating systems. MKL has Fortran and C/C++ interfaces predefined, and is utilized by optimized Python implementations (MKL has been used to accelerate SciPy and NumPy in some Python distributions). MKL provides the acceleration of BLAS and more within several third-party math libraries. See the "For More Information" section at the end of this chapter for links. MKL takes advantage of vectorization and shared memory multiprocessing capabilities, and, for certain components, supports distributed memory parallelism using Message Passing Interface (MPI). MKL includes the following groups of routines:

- Basic Linear Algebra Subprograms (BLAS):
  - Level 1 BLAS: vector operations
  - Level 2 BLAS: matrix-vector operations
  - Level 3 BLAS: matrix-matrix operations
- Sparse BLAS Level 1, 2, and 3 (basic operations on sparse vectors and matrices)
- LAPACK routines for solving systems of linear equations, least squares problems, eigenvalue and singular value problems, Sylvester's equations, and auxiliary and utility LAPACK routines
- ScaLAPACK computational, driver, and auxiliary routines for solving systems of linear equations across a compute cluster
- PBLAS routines for distributed vector, matrix-vector, and matrix-matrix operation
- MKL PARDISO, direct sparse solver routine implementing LU factorization and Cholesky factorization for matrices stored in sparse data format
- Cluster Sparse Solver, distributed memory version of MKL PARDISO
- Extended Eigensolver for finding a subset of sparse or dense matrix-matrix spectrum and eigenvectors
- Deep neural network primitives, a set of highly optimized building blocks for convolutional neural networks
- Vector math functions for computing core mathematical functions on vector arguments
- Vector statistical functions for generating vectors of pseudorandom numbers with different types of statistical distributions and for performing convolution and

correlation computations. VSL also includes summary statistics functions. Used to accurately estimate, for example, central moments, skewness, kurtosis, variance, quantiles, and to compute variance-covariance matrices and correlation matrices
- FFT functions, providing fast computation of Discrete Fourier Transforms
- Cluster FFT functions functions to compute Discrete Fourier Transform on distributed memory systems using MPI protocol
- Tools for solving partial differential equations — trigonometric transform routines and Poisson solver
- Optimization Solver routines for solving nonlinear least squares problems through the Trust-Region algorithms and computing Jacobi matrix by central differences
- Data Fitting functions for spline-based approximation of functions, derivatives and integrals of functions, and cell search

The library functionality provides C and Fortran interfaces, and may be used from many high-level languages including Python, C#, and Java via native call mechanisms. Instructions and examples demonstrating use of MKL with managed runtime languages as well as recipes explaining use of MKL with popular open source packages are available in articles published at Intel Developer Zone (see the "For More Information" section at the end of this chapter).

MKL provides Intel® Thread Building Blocks (TBB) threading support in addition to Intel OpenMP threading (the TBB threaded capability was first introduced in 2015). The new Intel TBB threading layer allows seamless MKL integration to applications relying on Intel TBB technology for shared memory multithreading and helps to avoid issues like thread oversubscription leading to better application performance.

The Intel Math Kernel Library Reference Manual is the definitive resource for learning more about the many functions available in MKL (see the "For More Information" section at the end of this chapter). A very large number of the routines have industry standard interfaces that are used to access the Intel optimized implementations.

MKL provides both static and dynamic libraries for Knights Landing. To simplify linking, MKL offers a *Single Dynamic Library* interface. The MKL product includes a "Link Line Advisor" (available online) to help suggest the right linkage commands to meet your needs (see the "For More Information" section at the end of this chapter). The Advisor requests information about your system and on how you intend to use the library and then the tool automatically generates the appropriate link line for specified application configuration.

INTEL DATA ANALYTICS LIBRARY OVERVIEW

Intel DAAL is an open source library of optimized building blocks covering all stages of data analytics (Fig. 13.2): data acquisition from a data source, preprocessing, transformation, data mining, modeling, validation, and decision making. DAAL provides interfaces for C++, Java, and Python languages.

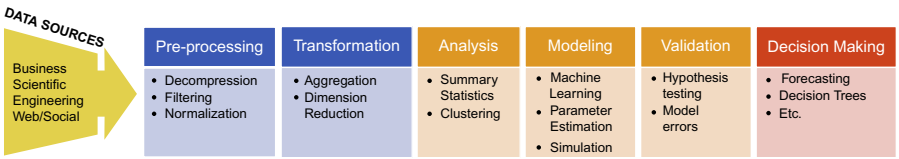


FIG. 13.2

DAAL covers all stages of data analytics.

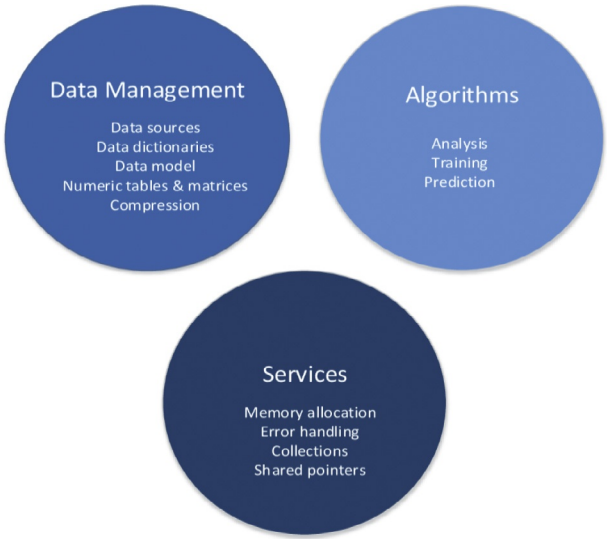


FIG. 13.3

DAAL structure.

The library consists of the following major components (Fig. 13.3):

- The Data Management component includes classes and utilities for data acquisition, initial preprocessing, and normalization, for data conversion into numeric formats done by one of the supported Data Sources, and for model representation.
- The Algorithms component consists of classes that implement algorithms for data analysis (data mining), and data modeling (training and prediction).
- The Services component includes classes and utilities used across Data Management and Algorithms components.

The Algorithms component consists of classes that implement algorithms for

- Data analysis (data mining)
  - Summary statistics: low order moments, variance-covariance and correlation matrix, quantiles

  - Distances: cosine and correlation matrix
  - Transformations: SVD, QR, Cholesky matrix decompositions
  - Principal component analysis
  - Association rules mining: a priori
  - Clustering: k-means, Expectation-Maximization for Gaussian Mixture Models
  - Outlier detection: univariate and multivariate algorithms
- Data modeling (training and prediction)
  - Regression: linear regression
  - Classification: Naïve Bayes, Support Vector Machine, Boosting algorithms
  - Recommendation: alternating least squares
  - Neural networks: building blocks for support of neural networks-based computations

The Algorithms support the following computation modes:

- Batch processing
- Online processing
- Distributed processing

In the *Batch processing* mode, the algorithm works with the entire data set to produce the final result. A more complex scenario occurs when the entire data set is not available at the moment or the data set does not fit into the device memory.

In the *Online processing* mode, the algorithm processes a data set in blocks streamed into device memory by doing incrementally updating partial results, which are finalized upon processing of the last data block.

In the *Distributed processing* mode, the algorithm operates on a data set distributed across several devices (compute nodes). The algorithm produces partial results on each node, which are finally merged into the final result on the master node.

Distributed algorithms in DAAL are abstracted from underlying cross-device communication technology, which enables use of the library in a variety of multi-device computing and data transfer scenarios. They include but are not limited to MPI-based cluster environments, Hadoop and Spark-based cluster environments, and low-level data exchange protocols.

For Knights Landing, the library provides the same feature set as on for other processors. DAAL relies on TBB for threading support.

TOGETHER: MKL AND DAAL

DAAL provides some algorithms such as matrix decompositions or covariance matrix which are already available in MKL. Both libraries support dense and sparse data types. At the same time, to choose the right library for specific application, the important differences between the libraries are provided below:

- Most of MKL is designed to support cases when all the data to be processed fits into memory at once. DAAL extends data processing to cases when the data is

distributed across computational nodes or all the data is not available and arrives in blocks.
- ScaLAPACK package available in MKL and supporting distributed processing of the data relies on MPI communication technology. DAAL abstracts from underlying cross-device communication technology, which enables use of the library in a variety of multidevice computing and data transfer scenarios including MPI, Spark, and low-level protocols.
- MKL supports processing of homogeneous data, for example double or single precision. DAAL extends this support to heterogeneous data and handles intermediate data conversion.

DAAL relies on building blocks from MKL and IPP to bring maximum performance gain on Intel architecture.

INTEL INTEGRATED PERFORMANCE PRIMITIVES LIBRARY OVERVIEW

The Intel IPP Library includes signal, image, video, and data processing functions optimized for Intel and compatible processor-based computers running Windows, OS X, Linux, Android, and VxWorks operating systems. IPP supplies interfaces for C. The primitives are a common interface for thousands of commonly used algorithms to help improve performance of multimedia, enterprise data, embedded, communications, and scientific/technical applications and are divided logically into domains (see Fig.13.4) for naming purposes.

Description	Domain Code	Header File	Prefix
Color conversion	CC	ippcc.h	ippi
String operations	CH	ippch.h	ipps
Core functions	CORE	ippcore.h	ipp
Cryptography	CP	ippcp.h	ipps
Computer vision	CV	ippcv.h	ippi
Data compression	DC	ippdc.h	ipps
Image processing	I	ippi.h	ippi
Signal processing	S	ipps.h	ipps
Vector math	VM	ippvm.h	ipps
Embedded functionality	E	ippe.h	ipps

FIG. 13.4

Relationship of IPP function names by IPP domains.

Intel® IPP Signal Processing domain:

- Signal Processing
- FFT, DFT, DCT, MDCT, Wavelet, Hilbert, Hartley, and Walsh-Hadamard Transforms
- Convolution, Cross-Correlation, Auto-Correlation, Conjugate
- Windowing, Jaehne/Tone/Triangle signal generation
- Digital Filtering
- Finite Impulse Response (FIR), Infinite Impulse Response, Single-Rate Adaptive FIR, Multi-Rate Adaptive FIR, Median Filter, Convolution and Correlation
- Coordinate Conversions (polarcartesian), Numeric Conversion (realcomplex), Emphasize, Nearest Neighbor, Threshold, etc.
- Statistics
- Mean, StdDev, NormDiff, Sum, MinMax

Intel® IPP Image Processing domain:

- Filters/Transforms
- Geometry transformations, such as resize/rotate
- Linear and nonlinear filtering operation on an image for edge detection, blurring, noise removal, etc.
- Linear transforms for 2D FFTs, DFTs, DCT
- Image statistics and analysis
- Computer Vision
- Background differencing, Feature Detection (Corner Detection, Canny Edge detection), Distance Transforms
- Image Gradients, Flood fill, Motion analysis, and Object Tracking
- Pyramids, Pattern recognition, Camera Calibration
- Canny, Optical Flow, Segmentation, Haar Classifiers
- Color Conversion
- Convert image/video color space formats: RGB, HSV, YUV, YCbCr
- Up/Down sampling
- Brightness and contrast adjustments

Intel® IPP Data Domains:

- Data Compression
- Huffman/Variable Length Coding (VLC)
- Lempel-Ziv-Storer-Szymanski (LZSS) PKzip
- Lempel-Ziv (lz77) Zlib, gzip
- Lempel-Ziv-Oberhumer (LZO) lzop
- Burrows-Wheeler Transform (BWT) bzip2
- Cryptography
- Symmetric cryptography (AES, TDES)
- One-way hash (SHA1, MD5)
- Public key cryptography (RSA)
- String Processing

- Find, Insert, Remove, Compare, Trim, Replace, Upper, Lower, Hash, Concatenate, Split
- Regular Expression Find/Replace

INTEL PERFORMANCE LIBRARIES AND INTEL COMPILERS

The libraries can be used with any x86/x86-64 compiler. This section focuses on special notes when using them with the Intel compilers.

To get maximal performance for an application on Knights Landing, or other processors with Intel AVX-512, use the -xMIC-AVX512 compiler switch. We can control what instructions are used when an application is run on a processor without AVX-512. For instance, in order to run an application to use Intel Advanced Vector Extensions (AVX) on processors where Intel AVX-512 is not available we can specify compiler options: -axMIC-AVX512,AVX (see the "For More Information" section at the end of this chapter has a link to "Intel® Compiler Options").

MKL AND INTEL COMPILERS

The Intel compilers conveniently have an option -mkl=lib which asks the compiler to link with MKL. The *lib* option indicates which part of the MKL library that the compiler should link. Possible values are:

- parallel: Tells the compiler to link using the threaded part of the MKL. This is the default if the option is specified with no lib.
- sequential: Tells the compiler to link using the nonthreaded part of the MKL.
- cluster: Tells the compiler to link using the cluster part and the sequential part of the MKL.

MKL supports two threading runtimes: Intel OpenMP, which is used in the majority of high-performance computing applications, and Intel TBB to provide seamless integration with applications using Intel TBB as the main threading model. The Intel compiler will automatically link the Intel TBB-based version of MKL when both -mkl and -tbb options are issued. If -qopenmp option is issued OpenMP based version of MKL will be used even if -tbb option is added. To understand how to link MKL in more complex scenarios, please refer to MKL Link Line Advisor (see the "For More Information" section at the end of this chapter).

DAAL AND INTEL COMPILERS

The Intel compilers conveniently have an option –daal=lib which asks the compiler to link with DAAL. The *lib* option indicates which part of the DAAL library that the compiler should link. Possible values are:

- parallel: Tells the compiler to link using the threaded part of the DAAL. This is the default if the option is specified with no lib.
- sequential: Tells the compiler to link using the nonthreaded part of the DAAL.

IPP AND INTEL COMPILERS

The Intel compilers conveniently have an option -ipp which asks the compiler to link with IPP.

NATIVE (DIRECT) LIBRARY USAGE

HIGH-BANDWIDTH MEMORY

One of the most important performance features of Knights Landing is the high-bandwidth memory which is on-package. This on-package high-bandwidth memory is of a particular type known as multi-channel dynamic random access memory (MCDRAM). This memory, and its usage, is covered in detail in Chapters 3 and 4. In Chapter 3, we explain how to allocate data in MCDRAM. Any function, including those in all three Intel Performance Libraries, can benefit from such allocations without requiring anything special in the libraries. Additionally, for the more complex functions in MKL and DAAL, the libraries have capabilities that can take special advantage of the MCDRAM. That support is explained in the next two sections. IPP functions do not directly manage MCDRAM since the routines in the IPP library are primitives. Instead, we can manage data placement outside the library and use the library to operate on the data. The library need not know that the data is in high-bandwidth MCDRAM memory, the advantages of such usage do not require any changes to the IPP library itself.

MATH KERNEL LIBRARY

The MKL memory manager detects the presence of MCDRAM in the system using libmemkind library (Chapter 3) and allows the user application to take advantage of the fast memory. All internal memory allocations and allocations made by user with `mkl_malloc` calls target MCDRAM by default. If allocation of memory to MCDRAM is not possible at the moment, MKL memory manager falls back to a regular system allocator and allocates memory in DDR.

The library provides controls to restrict the amount of high-bandwidth memory that it can use. To control the amount of MCDRAM available for MKL, do either of the following:

- Call `mkl_set_memory_limit (MKL_MEM_MCDRAM, <limit_in_mbytes>);`
- Set the environment variable:

```
export MKL_FAST_MEMORY_LIMIT="<limit_in_mbytes>"
```

The setting of the limit affects all MKL functions, including user-callable memory functions such as `mkl_malloc`. Therefore, if an application calls `mkl_malloc`, `mkl_calloc`, or `mkl_realloc`, which always tries to allocate memory to MCDRAM, make sure that the limit is sufficient.

If you replace MKL memory management functions with your own functions, MKL uses your functions and does not utilize the `libmemkind` library directly.

DATA ANALYTICS ACCELERATION LIBRARY

The DAAL library supports MCDRAM memory via its memory manager, which is similar to the one in MKL. The library allows controlling the amount of high-bandwidth memory that it can use via its service functionality.

```
services::Environment::getInstance()->setMemoryLimit ( MemoryType,
size_in_bytes );
```

The example in Fig. 13.5 shows how to configure MCDRAM size before the solution of the clustering problem with a DAAL version of a k-means algorithm which can increase algorithm performance.

INTEGRATED PERFORMANCE PRIMITIVES

IPP does not have its own memory manager and provides only memory allocation wrappers over system `malloc()` functions, therefore IPP does not provide any specific support for MCDRAM. The main aim of IPP memory allocation wrappers is to support in the most convenient way the data types used by IPP functions and provide

```
/* Use 50M of high bandwidth memory */
services::Environment::getInstance()->
   setMemoryLimit(daal::mcdram, 50000000 );
 /* Initialize FileDataSource to retrieve
    the input data from a .csv file */
 FileDataSource<CSVFeatureManager>
   dataSource(datasetFileName,
     DataSource::doAllocateNumericTable,
     DataSource::doDictionaryFromContext);

 /* Retrieve the data from the input file */
 dataSource.loadDataBlock();
 /* Generate initial clusters for the K-Means algorithm */
 kmeans::init::Batch<double,kmeans::init::randomDense>
      init(nClusters);
 init.input.set(kmeans::init::data, dataSource.getNumericTable());
 init.compute();
 services::SharedPtr<NumericTable>
      centroids = init.getResult()->get(kmeans::init::centroids);

 /* Create an algorithm object for the K-Means algorithm */
 kmeans::Batch<> algorithm(nClusters, nIterations);
 algorithm.input.set(kmeans::data,dataSource.getNumericTable());
 algorithm.input.set(kmeans::inputCentroids, centroids);
 algorithm.compute();

 /* Print the clusterization results */
 printNumericTable(algorithm.getResult()->get(kmeans::assignments),
                   "First 10 cluster assignments:", 10);
```

FIG. 13.5

Controlling high-bandwidth memory usage with DAAL, k-means example.

the most appropriate memory buffer alignment for the current architecture/platform. To utilize MCDRAM, we should allocate using other methods such as those discussed in Chapter 3.

OFFLOADING TO KNIGHTS LANDING WHILE USING A LIBRARY

Chapter 18 covers offloading over PCIe to a Knights Landing coprocessor or offloading over fabric to a Knights Landing processor. Compiler Assisted Offload (CAO) is the mechanism for "Offload over Fabric" which can apply to use of MKL, DAAL, and IPP, is also covered in more detail in Chapter 18. AO is a special feature of MKL which is covered in this chapter alongside some information on CAO with MKL.

AUTOMATIC OFFLOAD IN MKL

MKL provides a compatible experience for the Knights Landing coprocessor mode as with the original Intel Xeon Phi coprocessor. MKL supports two modes that allow offload computations to Intel Xeon Phi coprocessors: AO and CAO. AO mode is available with the OpenMP-threaded MKL but currently not the TBB-threaded MKL; CAO supports all versions of MKL. The AO functionality of MKL, discussed in the rest of this section, is focused on using an attached coprocessor and has not been extended or tuned to support to "Offload over Fabric" as of publication of this book. That could change in the future — you may want to consult documentation for new MKL releases.

AO allows application to take advantage of Intel Xeon Phi coprocessor with minimal changes to the code at expense of flexibility with data movement. Compiler-Assisted Offload mode allows to take full advantage of MKL performance on host and coprocessor sides and manages data movement explicitly.

In AO mode, supported for 64-bit binaries, MKL automatically detects the presence of coprocessors based on Intel MIC Architecture and automatically offloads computations that may benefit from additional computational resources available. This usage model enables calling MKL routines as we would normally do with minimal changes to an application. AO mode is available for a limited set of MKL functionality including BLAS and a subset of LAPACK functionality. Refer to MKL User Guide for the complete list of functions and problem sizes (see the "For More Information" section at the end of this chapter).

AO mode does not require any changes in how MKL is linked with the application. The only change needed to enable this mode is either the setting of an environment variable (export MKL_MIC_ENABLE=1) or making a function call (mkl_mic_enable()). Automatic offloading can be disabled by setting the environment variable to zero (MKL_MIC_ENABLE=0) or there is an mkl_mic_disable() function call. The function calls take priority over the environment variables when used.

CAO relies on the Intel compiler and its offload pragmas to manage the functions and data offloaded to a coprocessor. Unlike AO mode, this mode allows to run larger portions of the application on the coprocessor and reuse data moved to the coprocessor between function calls. To start using CAO mode the compiler needs to be instructed to interpret offload pragmas and link coprocessor libraries using -offload-attribute-target=mic command line option. For instance, to build simple parallel application for CAO mode on Linux platform with MKL use:

```
icc -qopenmp -mkl=parallel -offload-attribute-target=mic foo.c
```

For more complex usage scenarios, please refer to MKL Link Line Advisor, an online tool that helps to choose right set of libraries to link. All MKL functionality is available on both coprocessor and host. To call a function on a coprocessor, it's necessary to create an offload region and indicate which data should be transferred to and from coprocessor. For instance, multiplication of two dense matrices with help of the SGEMM function may look like that shown in Fig. 13.6.

MKL and the Intel Compiler provide debug mechanisms that allow checks if offload happened and to get profiling information for AO and CAO usage scenarios. The offload reporting mechanism can be enabled via the environment variable export OFFLOAD_REPORT = *level* or the MKL service function mkl_mic_set_offload_report(*level*). The *level* parameter can be set to 1, 2, or 3 to get different levels of detail.

MKL provides controls to manage what devices it is allowed to use, how much memory and computational cores it can utilize on each device via environment variables or service function calls. These controls help to use AO mode in scenarios when one or several coprocessors are shared between different MPI processes in the application. For such controls when we want them, in Figs. 13.7 and 13.8 enumerate the many additional controls are available for offloading. Functional interfaces take precedence over environment variables if both are used. The functions have definition files for Fortran (include: mkl.fi, module: mkl_service.mod) and C/C++ (include:

```
#pragma offload target(mic) \
in(transa, transb, N, alpha, beta) \
in(A:length(matrix_elements)) \
in(B:length(matrix_elements)) \
in(C:length(matrix_elements)) \
out(C:length(matrix_elements) alloc_if(0))
{
      sgemm(&transa, &transb, &N, &N, &N,
                  &alpha, A, &N, B, &N,
                  &beta, C, &N);

      }
```

FIG. 13.6

Offloading SGEMM (MKL).

Function {environment variable}	Description
mkl_mic_enable {MKL_MIC_ENABLE=1}	Enables AO mode
mkl_mic_disable {MKL_MIC_ENABLE=0}	Disables AO mode
mkl_mic_set_workdivision {MKL_MIC_WORKDIVISION or MKL_MIC_num_WORKDIVISION or MKL_HOST_WORKDIVISION}	For computations in the AO mode, sets the fraction of the work for the coprocessors, all or specified by num to do. Values of 0.0–1.0, or "MKL_MIC_AUTO_WORKDIVISION" to decide the best division of work at runtime. MKL interprets the values of these as guidance toward dividing work between coprocessors, but the library may choose a different work division. For LAPACK routines, setting the fraction of work to any value other than 0.0 enables the specified processor for AO mode but does not use the value specified to divide the workload
mkl_mic_get_workdivision	For computations in the AO mode, retrieves the fraction of the work for the specified target (processor or coprocessor) to do
mkl_mic_set_max_memory {MKL_MIC_MAX_MEMORY or MKL_MIC_num_MAX_MEMORY}	Sets the maximum amount of coprocessor memory reserved for AO computations. Can be set for all coprocessors or a specific num coprocessor. Memory size in kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T). For example, a valueof 4096M is the same as a value of 4G
mkl_mic_free_memory	Frees the coprocessor memory reserved for AO computations
mkl_mic_set_offload_report {OFFLOAD_REPORT}	The Intel compilers and MKL share the offload report capability, and reports will contain information about offload from both sources. Turns on/off reporting of AO profiling. Values of 0 (off), 1 (essential information), or 2 (everything/verbose)
mkl_mic_set_workdivision	For computations in the AO mode, sets the fraction of the work for the specified coprocessor or host CPU to do
mkl_mic_get_workdivision	For computations in the AO mode, retrieves the fraction of the work for the specified coprocessor or host CPU to do

FIG. 13.7

MKL support functions and environment variables useful for CAO and AO both.

(Continued)

`mkl_mic_set_max_memory`	Sets the maximum amount of coprocessor memory reserved for AO computations
`mkl_mic_free_memory`	Frees the coprocessor memory reserved for AO computations
`mkl_mic_register_memory {MKL_MIC_REGISTER_MEMORY}`	Enables/disables the mkl_malloc function running in AO mode to register allocated memory
`mkl_mic_set_device_num_threads {MIC_OMP_NUM_THREADS or MIC_num_OMP_NUM_THREADS}`	Sets the maximum number of OpenMP threads to use on an Intel Xeon Phi coprocessor for the AO computations
`mkl_mic_set_resource_limit {MKL_MIC_RESOURCE_LIMIT}`	For computations in the AO mode, sets the maximum fraction of available Intel Xeon Phi coprocessor computational resources (cores) that the calling process can use (0.0 and MKL_MPI_PPN are special values)
`mkl_mic_get_resource_limit`	For computations in the AO mode, retrieves the maximum fraction of available Intel Xeon Phi coprocessor computational resources (cores) that the calling process can use
`mkl_mic_set_flags / mkl_mic_get_flags`	Sets or retrieves flags to control the behavior of computations in the AO mode. (MKL_MIC_DEFAULT_FLAGS or MKL_MIC_DISABLE_HOST_FALLBACK) Specifies matrix size thresholds for ?GEMM computations in the AO mode
`mkl_mic_clear_status / mkl_mic_get_status`	For the AO mode, clears or gets the status (thread specific) of the latest call to an MKL function. Use clear before a call, and get after a call to MKL , all within the same thread
`MKL_MIC_THRESHOLDS_?GEMM`	Specifies matrix size thresholds for ?GEMM computations in the AO mode

FIG. 13.7, CONT'D

mkl.h). For more information, refer to the Reference Manual for Intel Math Kernel Library and the Intel® Compiler User and Reference Guides (see the "For More Information" section at the end of this chapter). There is also additional discussion in the chapter *Offload to Knights Landing* (Chapter 18).

DATA ANALYTICS ACCELERATION LIBRARY

DAAL does not have an AO mode. C++ DAAL supports Compiler-Assisted Offload (CAO) similarly to MKL and IPP.

The example in Fig. 13.9 shows use of DAAL Principal Component Analysis algorithm in the Compiler-Assisted Offload mode. Compiler-Assisted Offload is covered in detail in Chapter 18 (Offload Chapter).

INTEGRATED PERFORMANCE PRIMITIVES

IPP does not have an AO mode feature, but Compiler-Assisted Offload (CAO) works similarly to MKL and DAAL. Compiler-Assisted Offload is covered in detail in Chapter 18 (Offload Chapter).

Function {environment variable}	Description
mkl_mic_get_device_count	Returns the number of Intel Xeon Phi coprocessors on the system when called on processors.
mkl_mic_set_offload_report {OFFLOAD_REPORT}	The Intel compilers (CAO) and MKL(AO) share the offload report capability, and reports will contain information about offload from both sources. Turns on/off reporting of offload profiling. Values of 0 (off), 1 (essential information), or 2 (everything/verbose)
mkl_mic_get_device_count	Returns the number of Intel Xeon Phi coprocessors on the system when called on the host CPU
mkl_mic_get_meminfo	Retrieves the amount of total and free memory for the specified coprocessor or host CPU
mkl_mic_get_cpuinfo	Retrieves the number of cores, hardware threads, and frequency for the specified coprocessor or host CPU
MIC_LD_LIBRARY_PATH	Specifies the search path for coprocessor-side dynamic libraries

FIG. 13.8

MKL support functions and environment variables useful for AO mode.

```
/* Retrieve the data from the input file */
dataSource.loadDataBlock(nVectors);
    services::SharedPtr<NumericTable> dataset = dataSource.getNumericTable();
    services::SharedPtr<pca::Result> result;
#pragma offload target(mic:0) in(dataset), out (result)
{
    /* Create an algorithm for principal component analysis
       using the correlation method */
    pca::Batch<> algorithm;
    /* Set the algorithm input data */
    algorithm.input.set(pca::data, dataset);
    /* Compute results of the PCA algorithm */
    algorithm.compute();
    result = algorithm.getResult();
  }
  /* Print the results */
  printNumericTable(result->get(pca::eigenvalues), "Eigenvalues:");
```

FIG. 13.9

Using DAAL PCA algorithm in Compiler-Assisted Offload mode, PCA example.

PRECISION CHOICES AND VARIATIONS

Floating-point numbers have limited precision which give rise to question on both how we might speed up computation by selecting less precision or how we might make results more predictable. This section gives some of the most used options for both. The Intel compiler has many more controls for those with very precise needs, so the compiler documentation is worth reading through when we want much finer controls.

FAST TRANSCENDENTALS AND MATHEMATICS

Use of the AVX-512 Exponential & Reciprocal Instructions (EPI) on Knights Landing is default when using -fp-model fast (Windows /fp:fast). To selectively use the faster lower accuracy vectorizable math functions while also using -fp-model precise (Windows /fp:precise), we can specify:

- -fast-transcendentals for transcendentals such as sin, log, pow (Windows /Qfast-transcendentals)
- -no-prec-sqrt for square roots (Windows /Qprec-sqrt-)
- -no-prec-div for division (Windows /Qprec-div-)

The syntax of switches shown is for Linux and OS X, Windows syntax uses "/Q" instead of the leading hyphen. These are compiler options (Fortran or C/C++), and not an MKL library capability (libraries supplied by the Intel compiler including the math library (libm) and the short vector math library). Many algorithms find the performance of certain mathematics to be valuable and higher precision to be not required, hence the Intel compilers offers a rich set of options including controls over individual functions. Refer to the Intel Compiler User and Reference Guides for more details; see the "For More Information" section at the end of this chapter.

UNDERSTANDING THE POTENTIAL FOR FLOATING-POINT ARITHMETIC VARIATIONS

The floating-point model used by the Intel Compiler and its application to processors is described in the online article "Consistency of Floating-Point Results using the Intel Compilers" (see the "For More Information" section at the end of this chapter). For a suitable choice of settings, the compiler generates code that is fully compliant with the ANSI language standards and the IEEE-754 standard for binary floating-point arithmetic. Compiler options give the user control over the tradeoffs between optimizations for performance, accuracy, reproducibility of results, and strict conformance with these standards.

PERFORMANCE TIP FOR FASTER DYNAMIC LIBRARIES

If you are developing 64-bit applications on Linux, setting `LD_PREFER_MAP_32BIT_EXEC` is likely to improve performance. This is needed when we run an application. We set it on the machines where we run our program as it does not affect building an application. For instance, you could put the following in your `~/.bash_profile` file:

```
export LD_PREFER_MAP_32BIT_EXEC=
```

This works because for 64-bit applications, branch prediction performance on Silvermont or Knights Landing cores can be negatively impacted when the target of a branch is more than 4 GB away from the branch. It has been found on Linux that adding the `Prefer_MAP_32BIT_EXEC` bit so that `mmap` will try to map executable

pages with `MAP_32BIT` first. `Prefer_MAP_32BIT_EXEC` does reduce bits available for address space layout randomization, and can only be enabled by setting environment variable `LD_PREFER_MAP_32BIT_EXEC`. Anecdotal feedback indicates this can often give a boost of several percent in performance. This is a feature of glibc 2.23 and later.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this chapter.

- Intel Math Kernel Library Link Line Advisor, http://lotsofcores.com/Link-MKL.
- Intel® Compiler Options for Intel® SSE and Intel® AVX generation (SSE2, SSE3, SSSE3, ATOM_SSSE3, SSE4.1, SSE4.2, ATOM_SSE4.2, AVX, AVX2, AVX-512) and processor-specific optimizations, http://lotsofcores.com/AVX512options.
- Intel tools documentation including User Guides and Reference Manuals for each Intel Performance Library and the Intel compilers: https://software.intel.com/intel-parallel-studio-xe-support/documentation.
- Optimized library for small matrix-matrix multiplication (n<40), libxsmm, https://github.com/hfp/libxsmm
- Improve MKL Performance for Small Problems, https://software.intel.com/en-us/articles/improve-intel-mkl-performance-for-small-problems-the-use-of-mkl-direct-call.
- Consistency of Floating-Point Results using the Intel® Compiler, http://software.intel.com/articles/consistency-of-floating-point-results-using-the-intel-compiler/.
- libmemkind memory management library, https://github.com/memkind/memkind.
- Glibc 2.23 patch to add Prefer_MAP_32BIT_EXEC to improve branch prediction on Silvermont, https://sourceware.org/bugzilla/show_bug.cgi?id=19367.
- Rogue Wave IMSL Fortran Numerical Library, https://software.intel.com/rogue-wave-imsl-fortran-library.
- NAG Libraries for Intel Xeon Phi products, http://www.nag.com/downloads/xpdownloads.
- Python optimized with MKL, https://software.intel.com/articles/python-optimized.
- Anaconda Python, optimized with MKL, https://www.continuum.io/downloads.
- Microsoft R Open, optimized with MKL, http://lotsofcores.com/rev-mkl.
- Community Licenses (free to everyone) for the Intel Performance Libraries, http://software.intel.com/nest.

CHAPTER

14 Profiling and timing

INTRODUCTION TO KNIGHT LANDING TUNING

What is new with Knights Landing in this chapter?

New Event-Monitoring Registers. Most advice in line with Intel Xeon processors.

Gaining insight into what the hardware is doing can be priceless. We like to think of profiling, or performance monitoring tools, as flashlights in an otherwise dark interior of a computer system. It feels as though the more complex the internals of computers get, the darker they get without flashlights. With up to 72 cores, four threads per core, high-bandwidth memory, 512-bit vectors, in a system with potentially thousands of processors, tuning for Knights Landing is a great candidate for using such illuminating performance-monitoring tools.

When we think of systems with Intel® Xeon Phi processor (specifically the Knights Landing microarchitecture — Knights Landing), we can think about wanting insights into the activities on each thread, communication traffic between threads, and the thread mapping to each core so that the high core counts available on the platform can be effectively used for maximum performance/throughput.

For insight into the activities of a processor, Intel supports event-monitoring registers which include support for platform-monitoring events, uncore (aka untile) events, and core events. Using hardware performance counters instead of more intrusive techniques (such as the compiler profiling option `-pg` based on code instrumentation) is critical when dealing with high-performance programs. Intrusive methods can be highly misleading because of unexpected side effects of simple profiling on the apparent performance.

In this chapter, we will discuss these events and some proven formulas to compute commonly desired metrics. The premier tool for access to these counters is the Intel® VTune Amplifier XE product. Additionally the open source community has Performance Application Programming Interface (PAPI). PAPI provides a consistent interface and methodology for use of the performance counter hardware found in most major microprocessors including Knights Landing. PAPI is used by quite a number of open source tools (a list is available on the PAPI website, see the For More Information section at the end of this chapter) as well as some commercial products.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00014-4

For insight into MPI communications between ranks regardless of whether the rank is on a processor or coprocessor, we will introduce the Intel® Trace Analyzer and Collector (ITAC). Other opensource tools like Tau, HPCtoolkit will be briefly covered here.

One might wonder how to determine whether to tune at the node level with VTune, or at the cluster level with ITAC. The Intel® MPI Performance Snapshot (MPS) can answer this question. For motivational and more usage information on MPS and ITAC, see Chapter 15.

We will not cover the actual usage of VTune or other tools in this chapter, as that could easily occupy an entire book. Instead, we'll highlight the key things to entice you to learn more, and focus on things specific to profiling on Knights Landing. If you are already of a user of any of these tools, this will be sufficient to get an appreciation for the key Knights Landing-specific items. If you are not a user of these tools, we will give a feel for why you should consider becoming a user. There are online tutorials and documentation available for learning (see the For More Information section at the end of this chapter).

Software performance optimization can occur at many levels: application tuning, system tuning, operating system tuning, middleware tuning, and so on. Generally, a top-down approach is the most efficient: tuning of the system first, then optimizing the algorithms of the application, and then tuning at the microarchitecture level. System tuning, including tuning of the operating system, is normally done to remove hardware bottlenecks. Algorithmic tuning involves things like adding parallelism, tuning I/O, or choosing more efficient data structures or library routines. Algorithmic tuning generally relies on knowledge of the application's hotspots and familiarity with the source code, and aims to improve performance for the application in general. Tuning with a profiler can guide all these levels of optimization, but we would caution that a top-down approach is generally advisable, and therefore care should be taken to avoid optimizing at too low a level too soon. In other words, tune your kernels after you are sure that the kernels themselves are the right approaches from a "big picture" perspective.

EVENT-MONITORING REGISTERS

The event-monitoring registers on Knights Landing collect data for "activity we can count." These events will feel familiar to you if you know about event registers on other Intel processors with additional events which provide insights into a processor with more cores, more threads, and wider vectors than anything before it. One of the benefits of Knights Landing is this familiarity and the applicability of so many existing tools.

The events are useful for microarchitectural optimizations and how to identify where optimization work may be most impactful. Microarchitectural tuning relies on knowledge of how the application is executing on the hardware such as how

the pipelines, caches, and so forth are being utilized. Tuning at this level can be specific to the architecture and underlying hardware being used. For us to complete microarchitectural tuning, we need access to the real-time performance information gathered by the computing hardware itself while the application runs. This information is stored in a processing core's Performance Monitoring Unit (PMU), which can be programmed to count occurrences of particular events. VTune gives us the ability to both collect and view sampled data from Knights Landing. This section offers a framework for analyzing the event data collected from an application run on Knights Landing.

We will divide our discussion of the monitoring into two considerations: Efficiency Metrics and Potential Performance Issues.

LIST OF EVENTS USED IN THIS GUIDE

VTune uses the events listed in Fig. 14.1 for its predefined analysis types such as Advanced Hotspots, General Exploration, and Memory Access. It is also possible to create custom analysis types with a subset of these events. Although the events listed here are specific to Knights Landing, Intel architectures supported similar performance monitoring events across many generations of processors and someone familiar with the events on previous generations will be quite at home with this list. The ratios specified in the formulas to decide whether the events are signifying the potential existence of a performance problem is just an average across a large number of codes we ran on Knights Landing system — it is quite possible that an application can be out of these specified ranges and yet be quite efficient without any further opportunity for performance tuning.

EFFICIENCY METRICS

This section lists some general measures of efficiency that can help in evaluating when to optimize a particular piece of code. The section following this one will focus on a set of metrics that are valuable for application analysis. Along with each metric and its description are a formula for calculating the metric from available events, a threshold for determining when the value for a metric *may* indicate a performance problem, and some tuning suggestions.

There are several metrics that can be used to measure general efficiency on Knights Landing. You should look at these metrics first, to get an idea of how well the application is utilizing the resources available. These metrics (except where noted) can also be used to assess the impact of various optimizations as part of an iterative tuning process.

The formulas given for each metric are meant to be calculated at the function level (using the sum of samples from all hardware threads running). The Intel® VTune Amplifier XE interface performs this summation automatically if using

CPU_CLK_UNHALTED.THREAD	The number of cycles executed by the thread
CPU_CLK_UNHALTED.REF_TSC	The number of cycles executed by the thread at the reference frequency
INST_RETIRED.ANY	The number of instructions executed by the thread
UOPS_RETIRED.PACKED_SIMD	The number of Vector SSE/AVX operations executed
UOPS_RETIRED.SCALAR_SIMD	The number of Scalar SSE/AVX operations executed
MEM_UOPS_RETIRED.ALL_LOADS	The number of loads seen by a thread's L1 data cache
MEM_UOPS_RETIRED.ALL_STORES	The number of stores seen by a thread's L1 cache
MEM_UOPS_RETIRED.L1_MISS_LOADS	The number of loads that missed in L1 cache
MEM_UOPS_RETIRED.L2_MISS_LOADS	The number of loads that missed in L2 cache
MEM_UOPS_RETIRED.L2_HIT_LOADS	The number of loads that hit in L2 cache
MEM_UOPS_RETIRED.HITM	The number of loads retired that get the data from another core or from another module
MEM_UOPS_RETIRED.UTLB_MISS_LOADS	The number of L1 TLB misses for reads
MEM_UOPS_RETIRED.DTLB_MISS_LOADS	The number of L2 TLB misses for reads
UOPS_RETIRED.ALL	The number of operations executed
UOPS_RETIRED.MS	The number of operations executed out of MSROM
UNC_M_CAS_COUNT.RD	The number of reads from DDR
UNC_M_CAS_COUNT.WR	The number of writes to DDR
UNC_M_CAS_COUNT.ALL	The number of memory accesses to or from DDR
UNC_E_RPQ_INSERTS	The number of reads from MCDRAM
UNC_E_WPQ_INSERTS	The number of writes to MCDRAM

FIG. 14.1

Knights Landing events used in this guide.

the "Custom Analysis" Hardware Event-Based Sampling analysis type, and the "PMU events" tab with the "Function/Call stack" grouping. The summed values from this interface (per function) can be used to calculate the metrics in this guide. VTune will also calculate many of these metrics automatically when using one of the Advanced Hotspots, General Exploration, or Memory Access analysis types.

CYCLES PER INSTRUCTION

Cycles per instruction — description and usage

Cycles per instruction, or CPI, as defined in Fig. 14.2 is a metric that has been a part of the VTune interface for many years. It tells the average number of CPU cycles required to retire an instruction, and therefore is an indicator of how much latency in the system affected the running application. Since CPI is a ratio, it will be affected by either changes in the number of CPU cycles that an application takes (the numerator) or changes in the number of instructions executed (the denominator). For that reason, CPI is best used for comparison when only one part of the ratio is changing. For instance, changes might be made to a data structure in one part of the code that lower CPI in a (different) hotspot. "New" and "former" CPI could be compared for that hotspot as long as the code within it hasn't changed. The goal is to lower CPI, both in hotspots and for the application as a whole.

In order to make full use of the metric, it is important to understand how to interpret CPI when using multiple hardware threads. For analysis of Knights Landing performance, CPI can be analyzed in two ways: "per-core" or "per-thread." Each way of analyzing CPI can be useful. The per-thread analysis is the most straightforward. It is calculated from two events: `CPU_CLK_UNHALTED.THREAD` (also known as clock ticks or cycles) and `INST_RETIRED.ANY.CPU_CLK_UNHALTED.THREAD` counts ticks of the CPU core's clock when a thread is active. The other event used is `INST_RETIRED.ANY`, and this event is also counted at the thread level. On a sample, each thread executing on a core could have a different value for this event, depending on how many instructions from each thread have really been retired. Calculating CPI per thread is easy: it is just the result of dividing `CPU_CLK_UNHALTED.THREAD` by `INST_RETIRED.ANY`. For any given sample, this calculation will use the thread's value for clock ticks and an individual hardware thread's value for instructions executed. This calculation is typically done at the function level, using the sum of all samples for each function, and so will calculate an average CPI per hardware thread, averaged across all hardware threads running for the function.

CPI per core is fairly straightforward as shown in Fig. 14.3. To calculate an "Aggregate" CPI, or Average CPI per core, divide the sum of all core's threads `CPU_CLK_UNHALTED.THREAD` value by the sum of all the threads' `INST_RETIRED.ANY` values. For example, assume an application that is using one hardware thread per core on Knights Landing. One hot function in the application takes 1200 clock ticks to complete. During those 1200 cycles, the thread executed 600 instructions and hence the CPI of this thread is 2. But since the machine is capable of a CPI 0.5, the

Event	Meaning
`CPU_CLK_UNHALTED.THREAD`	The number of cycles executed per thread
`INST_RETIRED.ANY`	The number of instructions executed per thread

FIG. 14.2

CPI events.

Metric	Formula	Investigate if:
Average CPI per thread	CPU_CLK_UNHALTED.THREAD/ INST_RETIRED.ANY	>1, or generally increasing
Average CPI per core	(CPI per thread) / number of hardware threads used	>1, or generally increasing

FIG. 14.3

CPI formulas and thresholds.

thread is utilizing only ¼ of the capacity of the core. In such a situation consider adding another thread to the application so as to utilize the extra capacity of the core. Now assume that this application is parallelized and the two threads each gets 450 clocks and are able to retire 300 instructions each. Now the CPI of individual threads improved to 1.5 and the overall CPI of the core also improved to 1.5. Now we further parallelize this and add one more thread so that each thread is active for 200 clocks and executed 200 instructions each thereby improving the CPI to 1. We add one more thread to this and now each of the four threads is active for 75 clocks and retired 150 instructions, each achieving a CPI of 0.5. These performance numbers are summarized in Fig. 14.4.

This is a hypothetical example, and typically such scaling is not achievable in real world but illustrates how instructions from different hardware threads can be interleaved on a single core to utilize full capacity of the core. So the principle to be followed is to put multiple threads on a core only if one thread cannot fully utilize the core. If one thread is able to fully utilize the core and is able to achieve maximum CPI then it is better to map the threads to other cores. The availability of four hardware threads on Knights Landing can be useful for absorbing some of the latency of a workload's data access — while one hardware thread is waiting on data, another can be executing.

Fig. 14.5 shows the CPI per core for a real-workload run in our lab as the number of hardware threads/core is scaled from 1 to 4. For this application, the performance of the application was increasing with the addition of each thread, although the addition of the fourth thread did not add as much performance as did the second or third.

Number of Threads	Instructions	Clocks	CPI
1	600	1200	2
2	300	450	1.5
3	200	200	1
4	150	75	0.5

FIG. 14.4

CPI scaling across threads on a core.

Metric	1 hardware thread/core	2 hardware threads/core	3 hardware threads/core	4 hardware threads/core
CPI per core	5.24	4.40	3.73	3.43

FIG. 14.5

CPI example.

The data shows that the CPI per core is decreasing overall, as expected since each thread adds performance by effectively utilizing the spare capacity of the core. For this workload, the number of instructions executed was roughly constant across all the hardware thread configurations, so the CPI directly affected execution time. When CPI per core decreased, that translated to a reduction in total execution time for the application.

CPI — tuning suggestions

Any changes to an application will affect CPI, since it is likely that either the number of instructions executed or the time taken to complete them will change. The goal in general should be to reduce CPI per core (and therefore execution time), especially when compared to previous versions of the application. Most of the performance suggestions in each of the issues discussed later in the section "Potential Performance Issues" can be used to try to reduce CPI. Keep in mind that some beneficial optimizations, such as ones undertaken to increase Vectorization Intensity, may actually increase CPI because the amount of work done with a single instruction increases, and thus the number of instructions executed overall can decrease. CPI is most useful as a general comparison and efficiency metric rather than as a sole determinant of performance.

Knights Landing can support up to 288 threads (72 cores and four threads per core). If a program does not have significant sharing among threads and fewer than 288 threads, then the following scheme should be used for mapping threads to the cores to get best throughput and good CPI from each core and hardware thread. If there is significant sharing among the threads, then the cache locality advantage must be balanced against true parallelism advantage in deciding how threads will be mapped to the cores. Fig. 14.6 gives a suggestion on how to map threads to various cores depending on the overall number of threads in the system.

Number of Threads	Mapping
≤ 36 threads	1 thread per tile
≤ 72 threads	1 thread per core
≤ 144 threads	2 threads per core

FIG. 14.6

Thread to core mapping recommendation.

COMPUTE TO DATA ACCESS RATIO

Compute to Data Access Ratio — description and usage

The VPU is the vector processing unit in Knights Landing which is capable of performing multiple arithmetic operations in parallel on very long vector registers which are loaded from memory subsystem and a high-performance application makes effective use of the VPU. The events and formulas given in Figs. 14.7 and 14.8 provide a way to measure the computational density of an application, or how many computations it is performing on average for each piece of data loaded. The first, L1 Compute to Data Access Ratio, should be used to judge suitability of an application for running on Knights Landing. Applications that will perform well on Knights Landing should be vectorized, and ideally be able to perform multiple operations on the same pieces of data (or same cache lines). The L1 ratio calculates an average of the number of vectorized operations that occur for each memory access. Only vectorized operations, not including data operations, are included in the numerator by definition of the `UOPS_RETIRED.PACKED_SIMD` event. All demand loads are included in the denominator.

The threshold for the L1 metric is a guideline. Most codes that run well on Knights Landing should be able to achieve a ratio of computation to L1 access that is >1. This is similar to a 1:1 ratio, one data access for one computation, except that by vectorizing each computation should be operating on multiple elements at once. An application that cannot achieve a ratio above this threshold may not be computationally dense enough to fully utilize the capabilities of Knights Landing.

Computational density at the L1 level is critical. At the L2 level, it is an indicator of whether code is operating efficiently. Again, the threshold given is a guideline.

Event	Meaning
`UOPS_RETIRED.PACKED_SIMD`	The number of VPU operations executed by the thread
`MEM_UOPS_RETIRED.ALL_LOADS`	The number of loads seen by a thread's L1 data cache
`MEM_UOPS_RETIRED.L1_MISS_LOADS`	The number of loads that miss a thread's L1 cache

FIG. 14.7

Compute to Data Access Ratio events.

Metric	Formula	Investigate if:
L1 Compute to Data Access Ratio	`UOPS_RETIRED.PACKED_SIMD / MEM_UOPS_RETIRED.ALL_LOADS`	<1
L2 Compute to Data Access Ratio	`UOPS_RETIRED.PACKED_SIMD/ MEM_UOPS_RETIRED.L1_MISS_LOADS`	<100x L1 Compute to Data Access Ratio

FIG. 14.8

Compute to Data Access Ratio — formulas and thresholds.

For best performance, data should be accessed from L1. This doesn't mean that data cannot be streamed from memory — the high aggregate bandwidth on Knights Landing is advantageous for this. But, ideally, data should be streamed from memory into the caches using prefetches (hardware or software), and then should be available in L1 when the demand load occurs. This is even more important for Knights Landing than for processors with fewer cores and smaller vector units. Long data latency lessens the performance benefits of vectorization, which is one of the cornerstones of Knights Landing performance. The L2 Compute to Data Access Ratio shows the average number of L2 accesses that occurred for each vectorized operation. Applications that are able to block data for the L1 cache, or reduce data access in general, will have higher numbers for this ratio. As a baseline, the threshold of 100 × the L1 ratio has been used, meaning there should be roughly 1 L2 data access for every 100 L1 data accesses. Like the L1 metric, it includes only vectorized arithmetic operations (not including data movement) in the numerator.

The denominator for the L1 metric includes all *demand* loads. *Demand* loads and store counts do not include software or hardware prefetches in their counts. The denominator for the L2 metric is slightly more complicated — it uses all the demand data loads that missed L1 — only these will be requested from L2. It is strongly related to the L1 Hit Rate discussed later in this chapter.

Compute to Data Access Ratio — tuning suggestions

For the computational density metric, if the value is < 1, general tunings to reduce data access should be applied. This is best accomplished by trying to vectorize the load instructions. Removal of conditionals, initialization, or anything not needed in inner loops will help. Streamline data structures — align data and ensure the compiler is assuming alignment in generating loads and stores and wherever possible specify alias directives which lets compiler generate good vectorized code. For example, ensure that the compiler is not register spilling. Eliminate task or thread management overhead as much as possible.

For the L2 computational density metric, try to improve data locality for the L1 cache using techniques described later in this chapter. Restructuring code using techniques or pragmas from OpenMP (such as `#pragma simd`) can also enable the compiler to generate more efficient vectorized code, and can help improve both the L1 and L2 metrics.

POTENTIAL PERFORMANCE ISSUES

This section highlights several possible performance issues that can be detected using events. For each issue, the events needed are listed along with their descriptions. Each issue is identified using metrics and thresholds. Like the metrics given in the prior section, the formulas given for the metrics below are meant to be calculated at the function level (using the sum of samples from all hardware threads running and taking into account thread migration among various cores).

The value computed for each metric should then be compared to the threshold value. The thresholds given in this document are generally chosen conservatively. This means that an application is more likely to trigger the threshold criteria without having a problem than to have one of the given issues without triggering the threshold. The thresholds only indicate that you may want to investigate further. All of the metrics in this section are also designed to be used after the execution environment is fixed (will be held constant during tuning analysis work). Changes to the number of hardware threads or cores used may affect the predictability of these metrics.

GENERAL CACHE USAGE

General cache usage — description and usage

For applications running on Knights Landing, good data locality is critical for achieving their performance potential. In order to realize the benefit from vectorizing applications, the data must be accessible to be packed into VPU registers at as low a latency as possible. Otherwise, the time to pack the registers dominates the time to do the computation. Even though out of order execution and multithreading do hide some data access latency, it can still have a significant impact on performance. Therefore, improving data locality is one of the most worthwhile optimization efforts for Knights Landing. Both L1 and L2 locality are important. Program changes that result in data being accessed from local L2 cache as opposed to a remote cache or memory save at least 250 cycles of access time. Under load, the savings are even greater. Accessing data from L1 as opposed to L2 saves about 13 cycles.

Fig. 14.9 gives the list of events available that can monitor the activity at the caches and Fig. 14.10 gives the list of metrics that can be computed using these events to gauge the effectiveness of the cache. Traditionally, Hit Rate metrics

Event	Meaning
CPU_CLK_UNHALTED.THREAD	The number of cycles in which the core was executing
MEM_UOPS_RETIRED.ALL_STORES	The number of stores seen by a thread's L1 data cache
MEM_UOPS_RETIRED.ALL_LOADS	The number of loads seen by a thread's L1 data cache
MEM_UOPS_RETIRED.L1_MISS_LOADS	The number of loads that miss in the L1 cache
MEM_UOPS_RETIRED.L2_MISS_LOADS	The number of loads that miss in the L2 cache
MEM_UOPS_RETIRED.L2_HIT_LOADS	The number of loads that hit in the L2 cache
MEM_UOPS_RETIRED.HITM	The number of loads retired that get the data from another core or from another module

FIG. 14.9

General cache usage events.

Metric	Formula	Investigate if:
L1 Load Miss Rate	MEM_UOPS_RETIRED.L1_MISS_LOADS/ MEM_UOPS_RETIRED.ALL_LOADS	>0.1
L2 Hit Rate	MEM_UOPS_RETIRED.L2_HIT_LOADS/ (MEM_UOPS_RETIRED.L2_HIT_LOADS+ MEM_UOPS_RETIRED.L2_MISS_LOADS)	<0.8
L2 Hit Penalty	17*MEM_UOPS_RETIRED.L2_HIT_LOADS/ CPU_CLK_UNHALTED.THREAD	>0.15
L2 miss Penalty	230*MEM_UOPS_RETIRED.L2_MISS_LOADS/ CPU_CLK_UNHALTED.THREAD	>0.05
Contested accesses	MEM_UOPS_RETIRED.HITM/ MEM_UOPS_RETIRED.ALL_LOADS	>0.05

FIG. 14.10

General cache usage — formulas and thresholds.

indicate how well each level of cache is being used. It is normally calculated by dividing the number of hits by the total number of accesses for that level of cache. Hit rates also typically only apply to "demand" accesses. A demand access is an actual load issued by the application as opposed to a software or hardware prefetch. Knights Landing has events which compute the L1 and L2 cache misses directly for loads; there are no events that focus on stores except for MEM_UOPS_RETIRED.ALL_STORES. Stores typically are not an issue (unlike loads where computation grinds to a halt if not serviced immediately) since they are buffered and committed opportunistically.

The L1 load hit rate and L2 load hit rate metrics tell how effective L1/L2 caches are in capturing the data locality of the application. Typical expectation is that the L1 gets a hit rate of more than 90% and L2 gets a hit rate of more than 80% and if either of those is the case then one needs to take steps to improve them for getting better performance from the application running on Knights Landing.

The L2 hit penalty estimates the cost of loads that missed the L1 cache but hit in the L2 cache. If this number is more than 15% of the total cycles consumed by the application then it deserves attention. Some restructuring of the code or data structures could be done to improve the L1 hit rate instead of going to the L2 cache.

The L2 miss penalty estimates the cost of loads that have to go beyond L2 cache either into the MCDRAM or into the DDR. The expectation is that this cost will be $<5\%$ of the total time spent.

The contested accesses metric computed as MEM_UOPS_RETIRED.HITM/MEM_UOPS_RETIRED.ALL_LOADS can also be helpful for tuning data locality. A high value of this metric indicates that too much data was being accessed from other core's caches. Since remote cache accesses have high latency for memory accesses, they should be avoided if possible. This arises due to several reasons, including "false sharing" where two different items that happen to be in the same cache line are being read and written by two different cores, "true sharing" where global variables are shared across threads, and for variables used in synchronization primitives like locks.

General cache usage — tuning suggestions

All traditional techniques for increasing data locality apply to Knights Landing: cache blocking, software prefetching, data alignment, and using streaming stores can all help keep more data in cache. For issues with data residing in neighboring caches, using cache-aware data decomposition or private variables can help. Set associativity issues are another type of data locality problem that can be difficult to detect. If hit rates are low despite trying some of the above techniques to reduce them, conflict misses occurring from too many cache lines mapping to the same set may be the culprit. In Knights Landing, the L1 is 8-way and the L2 is 16-way set associative, many cache lines have to map to the same set for conflict misses to occur. Unfortunately, the specific type of miss caused by set associativity issues (conflict misses) cannot be separated from general misses detected by events. If set associativity issues are suspected, try padding data structures (while maintaining alignment) or changing the access stride.

To improve the cache sharing metric, one can avoid false sharing by restructuring the data structure and paying attention to how data elements are laid out in memory (also called structure splitting), or by changing the algorithm if possible by duplicating data elements or improving communication patterns among threads.

TLB MISSES

TLB misses — description and usage

Knights Landing has a two-level TLB.[1] The L1 TLB (also called the uTLB) can map 64 4KB pages, and the L1 data TLB (also called the DTLB) can map 256 4KB pages or 128 2MB pages or 16 1GB pages. Fig. 14.11 gives the list of events available for monitoring the activity at these TLBs and Fig. 14.12 gives the list of metrics that can be derived from these events that show the effectiveness of TLBs. The penalty for missing the uTLB is four cycles. The L2 instruction TLB (called the ITLB) can map 48 4KB pages. When the page size is larger than 4KB, the ITLB and uTLB still hold entries for only 4KB chunks of the larger page (also referred to as "fractured" entry).

Event	Meaning
MEM_UOPS_RETIRED.UTLB_MISS_LOADS	The number of L1 TLB misses for reads
MEM_UOPS_RETIRED.DTLB_MISS_LOADS	The number of L2 TLB misses for reads
MEM_UOPS_RETIRED.ALL_LOADS	The number of read operations

FIG. 14.11

TLB misses events.

[1]Translation look-aside buffer, hardware for accelerating translation of virtual addresses to physical addresses by caching translations in a buffer.

Metric	Formula	Investigate if:
L1 TLB miss ratio	MEM_UOPS_RETIRED.UTLB_MISS_LOADS/ MEM_UOPS_RETIRED.ALL_LOADS	>1%
L2 TLB miss ratio	MEM_UOPS_RETIRED.DTLB_MISS_LOADS/ MEM_UOPS_RETIRED.ALL_LOADS	>0.1%
L1 TLB misses per L2 TLB miss	MEM_UOPS_RETIRED.UTLB_MISS_LOADS/ MEM_UOPS_RETIRED.DTLB_MISS_LOADS	>1
L1 TLB miss overhead	4*MEM_UOPS_RETIRED.UTLB_MISS_LOADS/ CPU_CLK_UNHALTED.THREAD	>0.05
L2 TLB miss overhead	PAGE_WALKS.D_SIDE_CYCLES/ CPU_CLK_UNHALTED.THREAD	>0.05
ITLB miss overhead	PAGE_WALJS.I_SIDE_CYCLES/ CPU_CLK_UNHALTED.THREAD	>0,05

FIG. 14.12

TLB misses — formulas and thresholds.

The L2 TLB miss penalty is at least 100 clocks; it is very difficult to hide this latency with prefetches, so it is important to try to avoid L2 TLB misses in general. Using software prefetches for L2 TLB when the stride is large is useful since hardware prefetcher cannot prefetch beyond a 4KB boundary. L1 TLB misses that hit in the L2 TLB are of less concern.

Since there are 64 cache lines in a 4KB page, the L1 TLB miss ratio for sequential access to all the cache lines in a page is 1/64. Thus any significant L1 TLB miss ratio indicates lack of spatial locality; the program is not using all the data in the page. It may also indicate thrashing; if multiple pages are accessed in the same loop, the TLB associativity or capacity may not be sufficient to hold all the TLB entries. Similar comments apply to large pages and to the L2 TLB.

In order to decide whether L1 TLB miss is a cause for concern and should be improved, check if the ratio L1 TLB miss overhead is significant portion of the overall cycle count.

If the L1 to L2 TLB miss ratio is high, then there are many more L1 TLB misses than L2 TLB misses. This means that the L2 TLB has the capacity to hold the program's working set, and the program may benefit from large pages.

If the ITLB is causing performance issues as seen by a high ITLB miss penalty ratio, then it needs to be reduced through code layout optimizations. Profile guided optimization (PGO), in the compiler, may be useful to help improve ITLB performance.

TLB misses — tuning suggestions

For loops with multiple streams, it may be beneficial to split them into multiple loops to reduce TLB pressure (this may also help cache locality). When the addresses accessed in a loop differ by multiples of large powers of two, the effective size of the TLBs will be reduced because of associativity conflicts. Consider padding between arrays by one 4KB page.

If the L1 to L2 TLB miss ratio is high, then consider using large pages.

In general, any program transformation that improves spatial locality will benefit both cache utilization and TLB utilization. The TLB is just another kind of cache.

If ITLB misses are a problem, then the PGO feature of the compiler is a good option to use to reduce the ITLB misses. The compiler divides the functions and basic blocks into hot and cold regions and generates a layout to improve the code locality which in turn makes the ITLB and ICACHE hit rate better.

VPU USAGE

VPU usage — description and usage

We would like to be able to measure efficiency in terms of floating-point operations per second, as that can easily be compared to the peak floating-point performance of the machine. However, Knights Landing does not have events to count floating-point operations. An alternative is to measure the number of vector instructions executed. These vector instructions include instructions that perform floating-point operations, integer operation, instructions that load vector registers from memory and store them to memory, instructions to manipulate vector mask registers, and other special purpose instructions such as vector shuffle.

Vector operations that operate on full vectors use the hardware's "all-ones" mask register %k0 and in some cases where certain vector lanes need to shut off, the mask register contains the appropriate bit pattern. Unfortunately, none of the SIMD related metrics of Knights Landing can tell how many vector lanes are shut off. All these metrics have to be carefully used with the understanding that they do not capture the effect of vector mask registers.

A reasonable rule of thumb to see how well a loop is vectorized is to take the ratio of `UOPS_RETIRED.PACKED_SIMD` to the sum of `UOPS_RETIRED.SCALAR_SIMD and OPS_RETIRED.PACKED_SIMD` for a given loop. If this ratio is >0.5, then there is a good chance that the loop is well vectorized. If the ratio is <0.5, then the loop was not well vectorized. A quirkiness in this approach is that some vector instructions require 2 μops to execute and the packed loads and stores are NOT counted in the `UOPS_RETIRED.PACKED_SIMD` event. This method should be used in conjunction with the compiler's vectorization report to get a more complete understanding of the vectorization inside a loop. Figs. 14.13 and 14.14 list these counters and the rule of thumb for investigation.

Event	Meaning
`UOPS_RETIRED.SCALAR_SIMD`	The number of scalar SSE/AVX operations executed
`UOPS_RETIRED.PACKED_SIMD`	The number of vector SSE/AVX operations executed

FIG. 14.13

VPU usage events.

Metric	Formula	Investigate if:
Vector VPU Intensity	UOPS_RETIRED.PACKED_SIMD / (UOPS_RETIRED.PACKED_SIMD+ UOPS_RETIRED.SCALAR_SIMD)	<0.5

FIG. 14.14

VPU usage — formula and threshold.

Care should be taken when attempting to apply this method to larger pieces of code. Various vagaries in code generation and the fact that mask manipulation instructions count as vector instructions can skew the ratio and lead to incorrect conclusions.

VPU usage — tuning suggestions

If vectorization intensity is <0.5, the vector units of Knights Landing are not effectively being utilized in the application — typically indicated poor vectorization by the compiler due to some issues in the code. For example, lots of scalar operations or lots of gathers and scatters lower vectorization intensity. The compiler vectorization report should be examined and see if the code can be restructured to enable better vectorization. Examination of the vectorization report may provide insight into the problems. Problems are typically one or more of:

1. Unknown data dependences. `#pragma simd` and `#pragma ivdep` can be used to tell the compiler to ignore unknown dependences or to tell it that dependences are of a certain type, such as a reduction.
2. Non-unit-stride accesses. These can be due to indexing in multidimensional arrays, or due to accessing fields in arrays of structures. Loop interchange and data structure transformations can eliminate some of these.
3. True indirection (indexing an array with a subscript that is also an array element). These are typically algorithmic in nature and may require major data structure reorganization to eliminate.
4. Array alignment can play an important role in the decisions about vectorization and care must be taken to appropriately specify the array alignment directives.

MICROCODED VPU INSTRUCTIONS

Microcoded instruction intensity — description and usage

Knights Landing is optimized for floating-point and as a result it uses microcode to implement byte and word-level operations in AVX. Even though these instructions look like vector operations, they are implemented through microcode flows which are essentially small programs made up of a fairly large number of μops executed out of MSROM — in contrast native AVX instructions are typically converted into 1 μop and directly executed by the hardware without accessing the MSROM. Because of the access to the MSROM and the large number of μops introduced into

the execution pipeline, AVX instructions operating on byte and word lengths take much longer time to execute than their corresponding brethren operating on integers.

Fig. 14.15 gives the list of events available to monitor the operations executed and whether they are coming from MSROM or directly translated by hardware and Fig. 14.16 gives a formula to decide whether this is a potential performance problem.

Microcoded instruction usage — tuning suggestions

If a large number of operations (i.e., >20%) are being executed from the MSROM, there is a potential for improvement depending on why these MSROM μops are issued — if they happen to be because of microcoded AVX instructions in the program (can be verified by visual inspection of the program where the MSROM instructions are high) then they can potentially be replaced by AVX/SSE integer instructions (which have half the vector length but are much more efficient).

MEMORY BANDWIDTH

Memory bandwidth — description and usage

Knights Landing has two kinds of memory in addition to the L1 and L2 caches — DDR and MCDRAM. DDR is the traditional main memory but MCDRAM is quite unique to Knights Landing where it can be configured to be a third-level cache, or in flat mode where it is mapped to the physical address space or a hybrid where half is configured as cache and another half is configured in flat mode and mapped to physical address space. Figs. 14.17 and 14.18 provide the list of events available for monitoring various things going on with these memories and some formulas to compute the bandwidth of these memories.

Computing the bandwidth of DDR or MCDRAM in flat mode is fairly straightforward — multiply the number of reads/writes to the appropriate memory with the line size (in this case 64) and divide by the time.

Event	Meaning
UOPS_RETIRED.ALL	The number of total operations executed
UOPS_RETIRED.MS	The number of operations executed out the MSROM

FIG. 14.15

VPU usage events.

Metric	Formula	Investigate if:
Microcoded instruction intensity	UOPS_RETIRED.MS / UOPS_RETIRED.ALL	>0.2

FIG. 14.16

Microcoded instruction usage — formula and threshold.

Event	Meaning
UNC_M_CAS_COUNT.RD	The number of read operations to DDR
UNC_M_CAS_COUNT.WR	The number of write operations to DDR
UNC_M_CAS_COUNT.ALL	The number of accesses (read + write) to DDR
UNC_E_RPQ_INSERTS	The number of reads to MCDRAM
UNC_E_WPQ_INSERTS	The number of writes to MCDRAM
UNC_E_EDC_ACCESS.HIT_CLEAN	Number of access (read + streaming store) that hit in MCDRAM cache and the data is clean with respect to the DRAM
UNC_E_EDC_ACCESS.HIT_DIRTY	Number of access (read + streaming store) that hit in MCDRAM cache and the data is dirty with respect to the DRAM
UNC_E_EDC_ACCESS.MISS_CLEAN	Number of accesses (read + streaming store) that miss in MCDRAM cache and the data evicted is clean with respect to the DRAM
UNC_E_EDC_ACCESS.MISS_DIRTY	Number of accesses (read + streaming store) that miss in MCDRAM cache and the data evicted is dirty with respect to the DRAM
UNC_E_EDC_ACCESS.MISS_INVALID	Number of accesses (read + streaming store) that miss in MCDRAM cache and the data evicted is invalid with respect to the DRAM. This event counts cold misses in MCDRAM cache
CPU_CLK_UNHALTED.THREAD	The number of cycles

FIG. 14.17

Memory bandwidth — events used.

When MCDRAM is configured as cache, the bandwidth calculation is much more complex since evictions of cache lines (modified or unmodified) have to be taken into account and the streaming stores should be counted. Some of these events cannot be monitored precisely in Knights Landing and so the bandwidth calculations will be not be accurate but they can be used as close enough approximation.

Memory bandwidth — tuning suggestions

You will want to know how much memory bandwidth your application is using. If your data sets fit entirely in L2 cache, then the memory bandwidth numbers will be small. If the application uses a lot of memory bandwidth (e.g., by streaming through long vectors) then this method provides a way to estimate how much of the theoretical bandwidth is achieved.

In practice, achieved DDR bandwidth of 100 GB/s is near the maximum that an application is likely to see. If the achieved bandwidth is substantially less than this, it is probably due to poor spatial locality in the caches, possibly because of set associativity conflicts, or because of insufficient prefetching. In the extreme case (random access to memory), many TLB misses will be observed as well.

Metric	Formula	Investigate if:
DDR Read bandwidth (bytes/clock)	UNC_M_CAS_COUNT.RD*64/ CPU_CLK_UNHLATED.THREAD	Different from expected value
DDR Write bandwidth (bytes/clock)	UNC_M_CAS_COUNT.WR*64/ CPU_CLK_UNHALTED	Different from expected value
DDR Bandwidth (GB/s)	UNC_M_CAS_COUNT.ALL*64/ time-in-sec	<100 GB/s
MCDRAM Read bandwidth (bytes/clock) in flat mode	UNC_E_RPQ_INSERTS*64/ CPU_CLK_UNHALTED.THREAD	Different from expected value
MCDRAM Write bandwidth (bytes/clock) in flat mode	UNC_E_WPQ_INSERTS*64/ CPU_CLK_UNHALTED.THREAD	Different from expected value
MCDRAM bandwidth (GB/s) in flat mode	(UNC_E_RPQ_INSERTS+ UNC_E_WPQ_INSERTS)*64/ time-in-sec	<500 GB/s
Derived MCDRAM ReadBytes (in cache mode)	UNC_E_RPQ_INSERTS * 64 – UNC_E_EDC_ACCESS.MISS_DIRTY * 64 – UNC_E_EDC_ACCESS.MISS_CLEAN * 64	Different from expected value
Derived MCDRAM WriteBytes (in cache mode)	UNC_E_WPQ_INSERTS * 64 – UNC_M_CAS_COUNT.RD*64	Different from expected value
MCDRAM Only BW (GB/s)	(Derived MCDRAM readBytes + Derived MCDRAM writeBytes) / Time-in-sec	Different from expected value
Total BW	(Derived MCDRAM readBytes + Derived MCDRAM writeBytes) + UNC_M_CAS_COUNT.RD *64 + UNC_M_CAS_COUNT_WR*64) / Time-in-sec	Different from expected value

FIG. 14.18

Memory bandwidth — formulas and threshold.

MCDRAM is a very high bandwidth memory compared to DDR. Based on the needs of an application, placing data structures in MCDRAM can improve the performance of the application quite substantially. See Chapter 3 for much more about tuning applications for MCDRAM.

INTEL VTUNE AMPLIFIER XE PRODUCT

The most widely used tool for collecting and analyzing the event-monitoring registers on Knights Landing is the Intel VTune Amplifier XE. The current process for using VTune to collect and view data from a Knights Landing is detailed in several documents listed in the For More Information section at the end of this chapter. If you do not yet use VTune, you should consider learning it from one of the fine documents or tutorials online.

Data may need to be collected over multiple runs, and metrics will need to be calculated outside of VTune. Support within VTune for Knights Landing will certainly continue to evolve and improve.

Although looking at the individual counts of various events can be useful, in this document most events will be used within the context of metrics covered in earlier sections. The general method to follow for performance analysis with VTune is:

1. Select a hotspot (a function with a large percentage of the application's total computational cycles).
2. Evaluate the efficiency of that hotspot using the metrics from the prior section titled "Efficiency Metrics."
3. If inefficient, check each applicable metric from the prior section titled "Potential Performance Issues." If a value of a metric is below the suggested threshold, or unacceptable by other standards, use the additional information in that section to find and fix the problem.
4. Repeat until all significant hotspots have been evaluated.

When following this method, it is important to carefully select a representative workload. Many of the metrics involve collecting several events, and this may require running the workload multiple times to collect data. An ideal workload should have some steady state phase(s), where behavior is constant for a duration longer than the data collection interval. The workload should also give consistent, repeatable results, and be the only application consuming a significant portion of computational time during data collection. If the workload is being run multiple times to collect data, ensure that there are no warm-cache effects or other factors that affect performance. Finally, before beginning analysis, a sanity check with basic events is encouraged — ensure the event counts are constant run-to-run and fall within expectations.

VTune has some prepackaged analysis types which preselect a number of events to be monitored and apply the formulas specified in the earlier sections to report the derived metrics. The tool is also configured to visually highlight the metrics that are out of range.

An important capability of VTune is the ability to show the mapping of threads to cores and the ability to deduce the communication patterns among the threads and level of parallelism achieved by the application. Considering that Knights Landing has both multithreading (four hardware threads that time multiplex on a core sharing the resources) and multiple cores (where threads can truly execute in parallel at the same time), it is important to clearly understand how the cores are being used by the application to fully exploit the capabilities of Knights Landing. For more details see For More Information at the end of this chapter.

AVOID SIMPLE PROFILING

Instrumenting for profiling using the `-pg` option may diminish or stop optimizations, thereby reducing performance from what the real "release" build would create. Better profiling can be done less intrusively because the "release" build can be used with VTune.

PERFORMANCE APPLICATION PROGRAMMING INTERFACE

Another way to use the event-monitoring registers on Knights Landing is through PAPI and tools that use PAPI. ITAC supports PAPI as well. The PAPI API is a popular project that allows use of the performance counter hardware found in most major microprocessors including Knights Landing. See the For More Information section at the end of this chapter for more PAPI reference material.

MPI ANALYSIS: ITAC

MPI is a popular programming model for highly parallel programs and is used extensively on Knights Landing systems. ITAC is a graphical tool for understanding MPI application behavior, quickly finding bottlenecks, improving correctness, and achieving high performance for parallel cluster applications on Intel architecture.

ITAC is based on low-intrusion instrumentation and supports applications written in C/C++/Fortran. It has capabilities that allow the user to decipher the communication patterns among the threads, graphically visualize the performance and thread communication, and see if there are opportunities for improving overall performance. ITAC also includes Intel's MPI Correctness Checker that detects deadlocks, data corruption, and errors with MPI parameters, data types, buffers, communicators, point-to-point messages, and collective operations. The MPI Correctness Checker can scale to extremely large systems and detect errors even among a large number of processes. For more information regarding ITAC, see Chapter 15.

HPCTOOLKIT

HPCToolkit from Rice University is an integrated suite of tools for measurement and analysis of program performance and is quite popular in HPC community. It supports a number of systems across a variety of architectures which includes Knights Landing.

HPCToolkit uses statistical sampling of timers and hardware performance counters and collects accurate measurements of a program's work, resource consumption, and inefficiency and attributes them to the full calling context in which they occur. HPCToolkit works with multilingual, fully optimized applications that are statically or dynamically linked. Since HPCToolkit uses sampling, measurement has low overhead (1–5%) and scales to large parallel systems. HPCToolkit's presentation tools enable rapid analysis of a program's execution costs, inefficiency, and scaling characteristics both within and across nodes of a parallel system. HPCToolkit supports measurement and analysis of serial codes, threaded codes (e.g., pthreads, OpenMP), MPI, and hybrid (MPI+threads) parallel codes.

TUNING AND ANALYSIS UTILITIES

TAU (Tuning and Analysis Utilities) Performance System is a portable profiling and tracing toolkit for performance analysis of parallel programs written in FORTRAN, C, C++, UPC, Java, and Python from the University of Oregon. TAU is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements as well as event-based sampling. All C++ language features are supported including templates and namespaces. The API also provides selection of profiling groups for organizing and controlling instrumentation. The instrumentation can be inserted in the source code using an automatic instrumentor tool based on the Program Database Toolkit (PDT), dynamically using DyninstAPI, at runtime in the Java Virtual Machine, or manually using the instrumentation API.

TAU's profile visualization tool, paraprof, provides graphical displays of all the performance analysis results, in aggregate and single node/context/thread forms. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface.

TIMING

There are two ways to time code sections on Knights Landing: Linux system routines and the internal TSC (time stamp counter or cycle counter). The `gettimeofday(2)` or `clock_gettime(2)` method is the preferred mode of timing if a long running piece of code needs to be timed while the `RDTSC` instruction is preferred for timing a short running piece of code on a given core. Care must be taken with `RDTSC` to ensure it gets executed on the given core by pinning the thread to the corresponding core using taskset or some other mechanism.

SYSTEM ROUTINES

The Linux system routines `gettimeofday(2)` and `clock_gettime(2)` are good ways to measure time taken by a piece of code. They both use fast paths through the kernel so they are relatively inexpensive to use: `gettimeofday(2)` takes 157ns per call on a 1200 MHz preproduction Knights Landing, and `clock_gettime(2)` with the `CLOCK_MONITONIC` clock takes 181ns per call. There are no serialization issues when calling either of these routines by multiple threads at the same time.

TIME STAMP COUNTER

Each core has a 64-bit TSC that is incremented every reference clock cycle (the counter is at a fixed frequency regardless of the actual CPU frequency). The values of TSC on each core are synchronized, so it is possible to compare TSC values generated by different cores (e.g., if you want to know if one event happened before another in a parallel program). TSC can even be used to measure wall-clock time by dividing by the reference frequency; however, if accurate wall-clock time is needed, especially over a long interval, `gettimeofday(2)` should be used instead.

The Read Time-Stamp Counter instruction `RDTSC` loads the content of the core's time-stamp counter into the `EDX:EAX` registers. The Intel C/C++ compiler supports the _rdtsc() intrinsic to return an unsigned long containing the TSC value. The TSC instruction takes approximately 30 clock cycles (25 ns at 1200 MHz). Getting access to the __rdtsc() intrinsic from Fortran is done via use of call the C function as shown in Fig. 14.19.

```
program GotTime
use, intrinsic :: iso_c_binding
implicit none
interface
  function rdtsc () bind(c,name="__rdtsc")
  !DEC$ ATTRIBUTES KNOWN_INTRINSIC :: rdtsc
  import
  integer(C_INT64_T) :: rdtsc
  end function rdtsc
end interface
print *, rdtsc()
end program GotTime
```

FIG. 14.19

Calling __rdtsc from Fortran.

FREQUENCY VARIATION

Knights Landing may run at different frequencies depending on turbo settings, system power management, or the use of AVX-512 instructions. The actual frequency at which a given section of code runs can be determined by taking the ratio of the two events `CPU_CLK_UNHALTED.THREAD` and `CPU_CLK_UNHALTED. REF_TSC`. VTune will display this ratio when using many of the predefined analysis types. This ratio can be especially useful to determine if heavy AVX-512 usage is reducing the actual clock frequency. Note that the RDTSC instruction always returns the same value as `CPU_CLK_UNHALTED.REF_TSC` and is not affected by the actual core frequency.

SUMMARY

Profiling a program is a critical element in gaining accurate insights into how a program is running, and therefore in how to tune an application. It is important to use a top-down approach to optimization to avoid optimizing a small portion of a code embodied in a nonoptimal high-level approach. Some profiling tools can give insight based on event counters built into Knights Landing. Other profiling tools specialize in analysis of communication traffic when using MPI as covered in more depth in the next chapter. Both can yield critical insights that may lead to breakthroughs in performance of an application.

FOR MORE INFORMATION

- Compiler Methodology (including performance optimization) for Intel Many Integrated Core architecture, http://software.intel.com//articles/programming-and-compiling-for-intel-many-integrated-core-architecture.
- Intel VTune Amplifier XE, http://software.intel.com/intel-vtune-amplifier-xe/.
- Intel Xeon Phi developer portal, http://software.intel.com/mic-developer.
- Intel® 64 and IA-32 Architectures Software Developer Manuals, http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.
- PAPI Web site, http://icl.cs.utk.edu/papi/index.html.
- Trace Analyzer and Collector, http://software.intel.com/intel-trace-analyzer/.
- HPCToolkit, http://hpctoolkit.org/; downloads on https://github.com/HPCToolkit.
- TAU performance tool kit https://www.cs.uoregon.edu/research/tau/home.php.

CHAPTER

MPI 15

INTERNODE PARALLELISM

What is new with Knights Landing in this chapter?
Processor, Cluster Modes, Intel ® Omni-Path architecture

Most of this book focuses on intranode parallelism, while this chapter, Message Passing Interface (MPI), and Chapter 16, Partitioned Global Address Space (PGAS), focus on internode parallelism. Parallelism in large systems exist both intranode (message passing, threading and vectorization) as well as internode (message passing, distributed memory programming). Both forms of parallelism are important to harness in order to obtain high performance on systems today.

For parallel computing on any collection of processors with disjoint memory spaces, we have to use distributed memory parallelization techniques with explicit data transfers between different processors. An overwhelming majority of applications today use the MPI for this task.

MPI can be used for intranode parallelism, as well. The improved serial and network performance allow putting more than a single rank per shared memory node on Knights Landing.

MPI ON KNIGHTS LANDING

Because Intel® Xeon Phi™ processors (Knights Landing) are compatible with other Intel processors, like most topics in this book, we could just skip talking about MPI at all. After all, using MPI on Knights Landing is really the same as any processor at the same scale. We have chosen to focus on the challenges of using MPI at high levels of scale, including use in heterogeneous systems populated with a mix of Intel® Xeon® processors and Knights Landing processors. We will discuss tuning as well with references to the main tuning chapter, Chapter 14. See the For More Information section at the end of this chapter for several textbooks and learning references for MPI and to download the source code used in this chapter.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00015-6

MPI OVERVIEW

MPI is not a programming language. MPI defines a set of library routines that can be called from C and Fortran programs. MPI programs typically employ a single-program, multiple-data approach. Multiple instances, or MPI ranks, of the same program run concurrently. Each rank computes a different part of the larger problem and uses MPI to communicate data between ranks. From the perspective of an MPI programmer, ranks may run on the same node or different nodes; the communication path may be different, but that is transparent to the MPI program.

The first MPI standard was drafted in 1993 resulting in MPI-1.1 in Jun. 1994. MPI-1.1 followed in 1998. MPI-1.2 and MPI-2.0 were published together and maintained in parallel. MPI-1.3 and MPI-2.1 specifications were published in 2008. MPI-2.2 was published in 2009, MPI-3.0 in 2012, and MPI-3.1 in 2015.

The latest MPI standard includes point-to-point message passing, collective communications, group, communicator, and several other concepts. MPI is generally implemented as a library with more than 500 functions. It contains explicit message passing and collective operations. MPI also offers one-sided calls similar to PGAS libraries like OpenSHMEM (see Chapter 16).

MPI allows applications to distribute data among many nodes of a cluster. Distributed computing enables applications to scale upward, taking advantage of growing clusters to solve larger problems.

MPI PROGRAMMING EVOLVES

The programming style for MPI has changed over the years, which in turn has continued to spur the evolution of the MPI standard. Hybrid codes using MPI and OpenMP (or other threading model like TBB or pthreads) became more interesting for compute nodes with the growing number of cores in a common shared memory domain. Each MPI rank spawns several threads and we have `#ranks * #threads=#logical` processors. Logical processors can be the number of available cores or available cores times the number of hyper or hardware threads. Maximizing the number of threads and minimizing the number of MPI ranks will reduce the overhead that grows with the number of ranks. This overhead is due to the memory consumption of internal buffers and limited scaling potential of collective MPI operations.

The simplest example of hybrid programming (`MPI_THREAD_FUNNELED`) would be an application that only calls MPI routines outside OpenMP parallel regions on the master thread. This concept simply mimics the original MPI program but accelerates computation between MPI communication by using parallelism. The more involved technique uses MPI calls on different threads (`MPI_THREAD_MULTIPLE`). This can be beneficial for overlapping computation on some of the threads with MPI communications occurring between other threads. This technique does demand more consideration of synchronization issues among the threads.

Additionally, MPI offers one-sided operations `MPI_Put` and `MPI_Get` that can directly read from or write into the memory of another MPI rank. Using these routines will reduce the number of inherent synchronizations that will come with an `MPI_Send` or `MPI_Receive` and makes it easier to overlap communication and computation.

Another interesting new feature of the MPI-3 standard is a set of *Nonblocking Collectives*. Some collective operations behave like a barrier. All ranks have to wait until the slowest rank finishes. This waiting time can now be overlapped with useful work and the final synchronization of the collective can be done when it becomes necessary.

The subsequent sections will show how to run, analyze, and tune MPI programs for clusters of many core processors. The techniques presented enable developers to optimize MPI applications for many core processors, thereby improving performance and scaling ever higher. Later in this chapter, we will come back to discussing the evolution of MPI programming in the section *Recent Trends in MPI Coding*.

FUTURE OF MPI VERSUS OTHER INTERNODE TECHNIQUES

As application developers target exascale computing beyond the year 2020, researchers debate the extent that MPI can be ready for this task. MPI has some overhead in memory and performance that cannot be ignored for runs with more than 100K MPI ranks. PGAS libraries or compilers may be a serious competition for reaching this goal first (see Chapter 16). Such alternate approaches are bolstered by a trend to support more sophisticated capabilities in the hardware that connects nodes, commonly referred to as the fabric. The MPI standard started in an era of very simple fabrics that placed a lot of demands on processors themselves to implement functionality. Exactly what this shift in hardware capabilities means long term remains to be seen, but it certainly will give rise to capabilities in MPI and in alternative approaches both.

Regardless of new entrants, MPI will almost certainly continue to dominate internode programming because of its very widespread use, portability, and proven abilities. There are an enormous number of applications that have been ported to use MPI during the last 20 years and the standard continues to evolve to meet new challenges.

HOW TO RUN MPI APPLICATIONS

The techniques for running MPI programs will be explained using Intel MPI. Other MPI implementations use similar commands. Many MPI users simply run existing MPI codes without modification. But even when we do not modify the source code, we can dramatically influence the performance by changing several runtime parameters. The first option is the rank distribution on the compute nodes. The communication speed between different MPI ranks will be different depending on their

distance. We may distinguish between shared memory communication inside the same core, tile, quadrant, socket, and distributed memory communication between different processors. Even the network will behave in a nonhomogeneous manner depending on the number of switches a message has to pass.

In theory, we have to place our MPI ranks in a way that we use fast connections for the heavy traffic. In prior clusters, this means putting ranks with a lot of communication between them on the same shared memory compute node. The new Intel Omni-Path Host Fabric Interface (HFI) adaptors tip the scales by providing bandwidth that is much higher than the shared memory bandwidth. In this chapter, we focus on mapping ranks to compute nodes. Optimization techniques are in the tuning chapter (Chapter 14).

STARTUP OF PURE MPI PROGRAMS

The startup mechanisms of MPI applications are not part of the Standard. We demonstrate how to handle Intel MPI programs on Linux, but most other distributions have similar semantics and similar features. The basic way of executing an Intel MPI program is (with <N>: number of MPI ranks):

```
$ mpirun -n <N> ./prg.x
```

This will be sufficient for a shared memory node or a cluster with a standard batch management system. On Clusters without a batch system or when we want to overwrite the predefined settings, we have to define the hostnames where our program should run (without this definition all ranks will be started on the local node):

```
$ mpirun -f <hostfile> -n <N> ./prg.x
```

`<hostfile>` is a file that contains the name of a host node per line. Even more control can be gained by using a `machinefile`:

```
$ mpirun -machinefile <machines> ./prg.x
```

Please note that there is no number of ranks defined. This is because we have to define the number of ranks per host inside the `machinefile` as shown in Fig. 15.1.

On a batch system, hostfile and machine file settings are ignored by default! The batch system definitions may be overwritten by using this variable:

```
$ export I_MPI_JOB_RESPECT_PROCESS_PLACEMENT=0
```

```
Host1:<N1>
Host2:<N2>
…
# ranks = N1+N2+…Nh, Nh:ranks on the last host
```

FIG. 15.1

Define the number of ranks per host inside the machinefile. Blank lines and lines starting with # are ignored.

If the amount of work is evenly distributed over ranks, the `machinefile` provides a simple mechanism to cope with heterogeneous architectures by placing fewer ranks on slower systems. Doing so can aid with load balancing. Load balancing is a critical topic for any MPI application. However, it will not be discussed in great detail here.

Another way of coping with homogeneous clusters could be done by using a wrapper script instead of the executable on the `mpirun` command line. The script can select actions including which binary to run. Inside such a script we could set environment variables for each branch independently. Environment variables may be set in the calling shell and they will be exported to all ranks. The script has also the advantage that we can make calls to `ulimit` in each branch. For Knights Landing the value of `uname -m` will be `x86_64`, so this is not a way to distinguish from other processors as it has been with Knights Corner. Instead, we may use the kernel name or any other environment variable identifying Knights Landing here.

We may also set environment variables in the `mpirun` command line instead of the Linux shell. This has the advantage of having their use better documented and keeping the Linux shell free of many variables that might influence later runs without intention:

```
$ mpirun -genv ENV_VARIABLE <VALUE> ...
```

STARTUP OF HYBRID MPI/OpenMP PROGRAMS

An important environment variable for hybrid MPI/OpenMP programs will be the number of OpenMP threads. By default it will use all available threads including hardware threads. This can already be the optimal setting, but we should also test settings with fewer threads per core. For Knights Landing, the key choice is the number of threads per core. We will show how to adjust this number with environment variables. The number of threads variable can be used for this, but we will present more generic solutions later.

Intel MPI has by default a pinning strategy for MPI Ranks. This strategy may change in the future; it is important to check whether the current pinning is as we expect it. We will learn in a later section, *Analyzing MPI Application Runs*, how to do this check.

The default pinning strategy will define nonoverlapping *pin domains* for each MPI rank. Each pin domain is a collection of logical processors. Threads that are spawned from this rank will be able to move inside the domain because the OpenMP library does not impose any restriction. On Knights Landing, we also have pin domains, but the behavior inside a domain is different because the OpenMP library pins threads by default to logical processors using the scatter strategy that will be explained later.

The natural choice for the Sub NUMA Cluster Mode with four Quadrants (SNC-4, see Chapters 3 and 4) would be four MPI ranks per Knights Landing making each Quadrant a *pin domain*.

The thread pinning inside a pin domain is completely analogous to the thread pinning on a shared memory node. Only the number of logical processors is lower.

Sharing a core on different pin domains should normally be avoided, but there can be odd distributions where this can happen. Early testing has shown that sharing of tiles results in a potential performance penalty! In the following discussion, we will assume that physical cores are not shared among different pin domains. For each domain, we may choose to have 1–4 threads per core on Knights Landing and 1–2 threads on an Intel Xeon processor. We will also present the different thread pinning strategies and how to archive them by setting environment variables. Let <NC> be the number of physical cores for our Knights Landing. The number of logical cores is <NLC> = 4*<NC>. The enumeration of logical cores is done round robin across cores:

Cores	0				1			…	NC-1			
Logical	0	NC	2NC	3NC	1	NC+1	2NC+1	…	NC-1	2NC-1	3NC-1	4NC-1

As an example on a 72 core Knights Landing:

Cores	0				1				…	71			
Logical	0	72	144	216	1	73	145	217	…	71	143	215	287

Default Knights Landing thread pinning—cluster mode: All-to-All

An Intel MPI application with <R> ranks per node will by default split the set of cores into <R> pin domains. The size of each pin domain is <NC>/<R> == <DC> cores—provided that <R> divides <NC> without rest. We should take into account that 2 cores are building a tile and share a L2 cache. Because of that it is also beneficial to keep all tiles in the same domain (<DC> should be an even number). The default number of threads per Domain (without setting additional environment variables) is <DLC> = 4*<DC> using all available logical processors.

<t> (t=1-4) threads per core are set by:

```
$ export KMP_PLACE_THREADS=<t>T
```

Or by setting the number of threads explicitly:

```
$ export OMP_NUM_THREADS=<DC>*<t>
```

The first solution does not need to compute the domain size and will be easier to realize. Both solutions work because of the implicit affinity `scatter` on Knights Landing.

Consider a 72 core Knights Landing node with 4 MPI ranks as an example. The next table shows the thread pinning for the third domain spanning 18 cores from core #36 to core #53. Without setting of `KMP_PLACE_THREADS` or `OMP_NUM_THREADS` we get 72 threads on each domain:

Core	36				37				38			…	53			
Logical	36	108	180	252	37	109	181	253	38	110	182	…	53	125	197	269
Thread	0	18	36	54	1	19	37	55	2	20	38	…	17	35	53	71

If we set `KMP_PLACE_THREADS=2T` or `OMP_NUM_THREADS=36` we get two threads per core:

Core	36				37				38			…	53			
Logical	36	108	180	252	37	109	181	253	38	110	182	…	53	125	197	269
Thread	0	18			1	19			2	20		…	17	35		

The default is equivalent to the following environment setting:

```
$ export KMP_AFFINITY=scatter
```

NOTE: The core numbering may be different from system to system depending on the Bios Version!

Compact thread pinning on Knights Landing—cluster mode: All-to-All

The prior pinning strategy gives us a round robin scattering across the cores. We might consider a strategy where consecutive threads are placed on consecutive logical processors, and this will improve data locality which in turn helps construct such as OpenMP loops:

```
$ export KMP_AFFINITY=compact
```

By using compact affinity, cores are filled first with four threads per core, before the next core is used. Taking our prior example:

Core	36				37				38			…	53			
Logical	36	108	180	252	37	109	181	253	38	110	182	…	53	125	197	269
Thread	0	1	2	3	4	5	6	7	8	9	10	…	68	69	70	71

This makes only sense if OMP_NUM_THREADS is not set or set to the number of logical cores (=72).

If we intend to use less than four threads per core with the compact affinity, we need an additional environment variable:

```
$ export KMP_PLACE_THREADS=<t>T   with t=1,2,3,4
```

With two threads per core we have for our example:

```
$ export KMP_PLACE_THREADS=2T
```

Core	36				37				38			…	53			
Logical	36	108	180	252	37	109	181	253	38	110	182	…	53	125	197	269
Thread	0	1			2	3			4	5		…	34	35		

Sub NUMA clustering mode (SNC)

Pinning strategies are basically the same as the prior shown, but the choice of cores per domain is different due to the membership of each core to one of the four NUMA domains. This mapping of cores to NUMA domains can be determined by executing `numactl -H` on the command line. Intel MPI respects NUMA domains and in the case of 4 MPI ranks each pin domain is equivalent to a NUMA domain.

Taking the prior example with 4 MPI ranks on a 72 core Knights Landing, we observe for the third domain and default affinity:

Core	4				5				10			…	65			
Logical	4	76	148	252	5	77	149	221	10	82	154	226	…	65	137	209	281
Thread	0	18	36	54	1	19	37	55	2	20	38	56	…	17	35	53	71

Core and logical processor enumeration may be subject to changes and could differ from this (preproduction system) example! The command `cpuinfo` that is part of the Intel MPI distribution shows more info about core and thread layout.

STARTUP OF MPI PROGRAMS USING OFFLOAD

A Knights Landing coprocessor can be used exactly as the original Intel Xeon Phi coprocessor, Knights Corner, for offloading computation tasks to the card. The setting of variables is quite similar to the previous hybrid configuration when we just offload from a single MPI rank to each attached card as shown in Fig. 15.2.

For jobs with two or more ranks offloading to a single card, we must make sure that the offloaded threads do not interfere. With no extra settings both ranks will offload their threads to the same subset of cores. To avoid this interference we need to use different offload thread pinning on each MPI Rank. We can do this by means of a wrapper script. In this wrapper script, we need to identify the exact rank number. The rank number is stored in the environment variable `PMI_RANK`. With that information,

```
$ export MIC_ENV_PREFIX=MIC
$ export MIC_KMP_AFFINITY=compact
$ export MIC_KMP_PLACE_THREADS=<t>T
$ export MIC_OMP_NUM_THREADS=<MTH>

$ mpirun -n <N> -ppn 1 ./offload_app
```

FIG. 15.2

Simple single rank configuration for an offload enabled app.

we can quite elegantly set the correct thread pinning for an offload program. The wrapper script in Fig. 15.3 manages the thread distribution on the coprocessor:

For a job with 2 MPI ranks per node and 2 (=<t>) threads per core on the coprocessor (72 cores available) with MIC_OMP_NUM_THREADS=8 and compact affinity we get:

```
MPI rank 0: MIC_KMP_PLACE_THREADS=4C@0x2T
      (==[0-3,72-75])
MPI rank 1: MIC_KMP_PLACE_THREADS=4C@4x2T
      (== [4-7,76-79])
```

ANALYZING MPI APPLICATION RUNS

It is always good practice to check whether the MPI program does really do what we intend it to do! We can start by setting the `I_MPI_DEBUG` variable, that is always good practice. Setting this variable to a positive integer will give information about the Intel MPI settings used for this run. This information is printed to `stdout` or to a file (or one file per rank) defined by another variable `(I_MPI_DEBUG_OUTPUT)`.

After verifying that the configuration is correct, we may start gathering statistics. This can be done by setting the variable `I_MPI_STATS` to a positive integer. The output will show some statistics about the sent data volumes and the used MPI functions.

More useful statistics can be gathered with the MPI Performance Snapshot (MPS). This tool is part of the Intel® Trace Analyzer and Collector (ITAC) package. It provides timing information for each MPI call used by a program and prints extensive statistics. It also provides OpenMP load balancing statistics if Intel compilers were used.

ITAC has a more detailed collection capability to obtain information about every MPI call. Once this data has been collected, the Analyzer component allows a developer to visualize communications within their program on a timeline of all MPI ranks. Due to the level of detail collected, large, realistic runs typically require additional data filtering in order to keep the analysis manageable.

Intel VTune Amplifier can also be used analyzing MPI ranks and their threads. The OpenMP analysis, inside the VTune tool, is vital for identifying and analyzing load balance and performance issues at a thread level in hybrid OpenMP MPI jobs.

```
THREADS_PER_CORE=<t>
CORES = $(( (MIC_OMP_NUM_THREADS+<t>-1)/<t> ))
OFFSET=$(( CORES*PMI_RANK ))
export MIC_KMP_PLACE_THREADS=${CORES}C@${OFFSET}x<t>T

./my_app
```

FIG. 15.3

Wrapper script to be used as the new executable.

DEBUG INFORMATION

Debug information is in fact a verbose output of several important settings that Intel MPI sets for the next MPI run.

```
$ export I_MPI_DEBUG=5
```

Setting the variable to 5 will give some useful information such as:

- The chosen or set fabric like shm:tmi or shm:ofi
- The psm2 Provider for Intel Omni-Path network
- The pin domains for each MPI rank
- The MPI environment variables set by the user

Fig. 15.4 shows an example of the Debug information for an 18 rank run on a single Knights Landing node.

Setting the variable to 6 will give additional information about the default values for collective algorithm settings.

```
[0] MPI startup(): Multi-threaded optimized library
[0] MPI startup(): shm data transfer mode
…
[17] MPI startup(): shm data transfer mode
[0] MPI startup(): Rank    Pid      Node name  Pin cpu
[0] MPI startup(): 0       80679    eke044\
{0,1,2,3,72,73,74,75,144,145,146,147,216,217,218,219}
[0] MPI startup(): 1       80680    eke044\
{4,5,6,7,76,77,78,79,148,149,150,151,220,221,222,223}
[0] MPI startup(): 2       80681    eke044\
{8,9,10,11,80,81,82,83,152,153,154,155,224,225,226,227}
…
[0] MPI startup(): 17      80696    eke044\
{68,69,70,71,140,141,142,143,212,213,214,215,284,285,286,287
[0] MPI startup(): I_MPI_DEBUG=5
[0] MPI startup(): I_MPI_INFO_NUMA_NODE_DIST=10,31,31,10
[0] MPI startup(): I_MPI_INFO_NUMA_NODE_MAP=hfi1_0:0,qib0:0
[0] MPI startup(): I_MPI_INFO_NUMA_NODE_NUM=2
[0] MPI startup(): I_MPI_PIN_MAPPING=18:0 0,1 4,2 \
  8,3 12,4 16,5 20,6 24,7 28,8 32,9 36,10 40,11 \
  44,12 48,13 52,14 56,15 60,16 64,17 68
```

FIG. 15.4

Sample debug output. Not all lines are shown because of the amount of data. A "\" indicates a line break. Some of the listed environment variables are set automatically by Intel® MPI and may not be listed in the reference guide.

Verbose information for the OpenMP thread mapping in each pin domain is created, by adding the flag "verbose," to the `KMP_AFFINITY` variable.

MPI STATISTICS

MPI statistics can be gathered by setting the variable I_MPI_STATS. The output gives a first impression about the amount of data that is sent during the whole MPI run:

```
$ export I_MPI_STATS=<l> # <l> = 1-4,10,20
```

The integer <l> controls the output: from amount of data sent by each process up to additional timing information for all MPI functions. After setting the environment variable and running the program, a file `stats.txt` will be generated. The variable `I_MPI_STATS_SCOPE` is useful to select the interesting information. Additional information can be found in the Intel MPI reference guide.

INTEL MPI PERFORMANCE SNAPSHOT

Intel® MPI Performance Snapshot (MPS) has fast become the starting point for tuning a hybrid program because it gives a high level picture of activity that quickly helps discern whether tuning MPI or tuning OpenMP, or other node level parallelism, is the best next step in tuning an application. MPS is very useful for complete profiles of runs with 10K+ MPI ranks. Most of the information in MPS comes from the MPI statistics previously mentioned, but it is in a more human readable form. Currently the output comprises the accumulated timing and volume statistics for all MPI functions and load balance statistics for MPI and OpenMP. Performance and hardware counters can also be collected using PAPI or VTune. To activate MPS, the ITAC source script or the MPS source script must be sourced:

```
$ mpirun -mps -n <N> ./prg.x
```

After running the program with the additional–mps flag, two new files will be generated: stats*.txt and app_stats*.txt. MPS can then be used to analyze the data collected (use a unique ID, e.g., a time stamp):

```
$ mps -s app_mpsID.txt statsID.txt
```

This will print statistics on `stdout`. Several options are possible and can be shown by calling MPS without parameters. The additional options can be used to tailor the output to our needs, including generating an HTML report. It is also possible to evaluate performance counters from VTune. This configuration can done when setting up the MPS environment.

Please note that the program does not have to be recompiled. Fig. 15.5 shows an example of the summary output of MPS presented in HTML format. The output may be customized by different options and level of detail. Fig. 15.6 shows a sample output containing VTune counters:

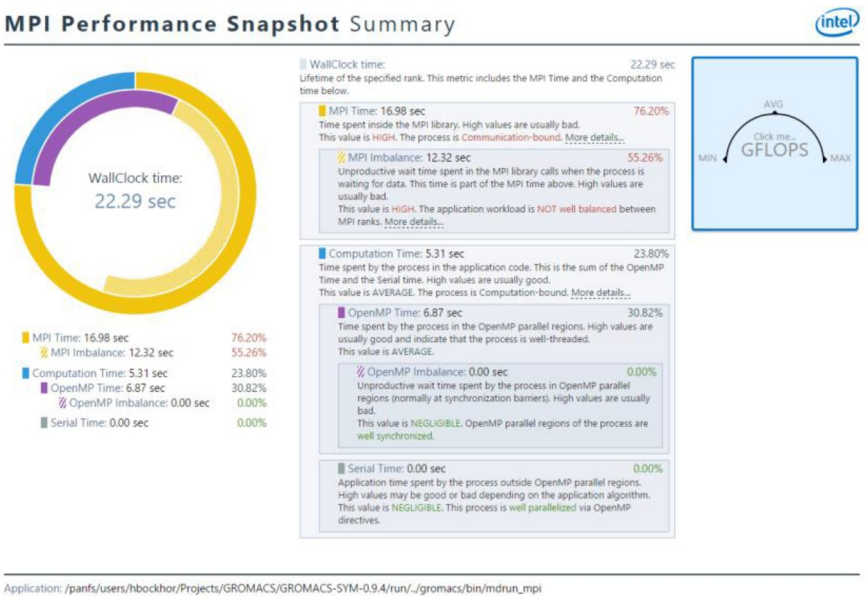


FIG. 15.5

Summary output of MPS presented in HTML format.

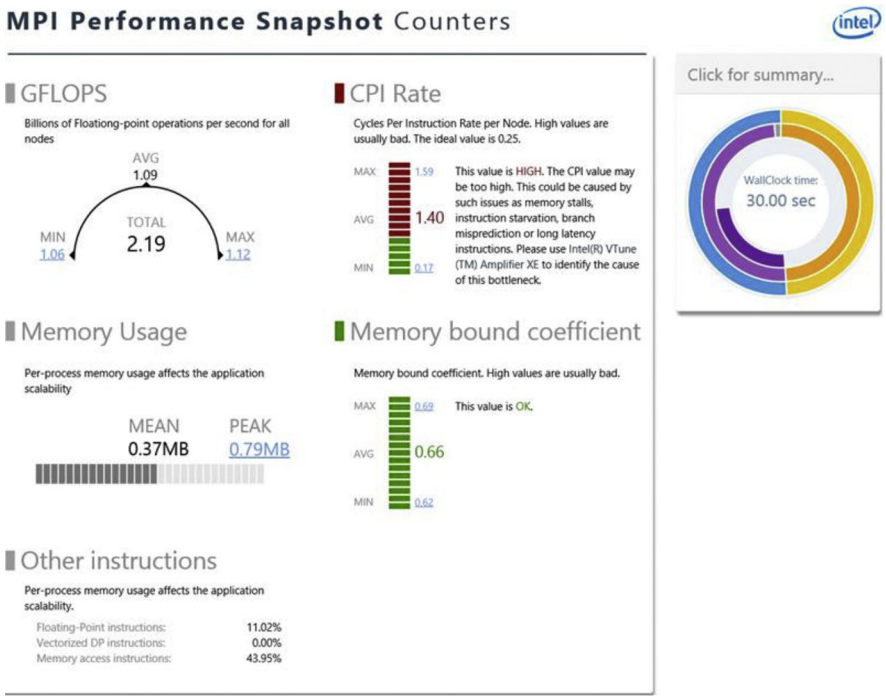


FIG. 15.6

VTune and PAPI counters evaluated.

Note Intel tools activation is done by a predefined environment script. Each Intel tool uses such a script. For MPS we use:

```
$ source $VT_ROOT/bin/mpsvars.sh
```

`VT_ROOT` points to the base directory of ITAC.

INTEL TRACE ANALYZER AND COLLECTOR

The Intel Trace Analyzer and Collector (ITAC) tool provides the most detailed MPI analysis. The complete timeline of an MPI program can be visualized including messages and collective operations. The MPI tracing can be activated with a simple switch to `mpirun`.

```
$ mpirun -trace .... prg.x
```

The resulting trace file will be named `prg.x.stf` if `prg.x` is the program name. It is highly recommended to set an additional environment variable:

```
$ export VT_LOGFILE_FORMAT=STFSINGLE
```

If this variable is not set there will be many additional files generated. This mechanism was originally designed for better scalability, but usually it is sufficient to just generate the single trace file which is much more convenient, especially for transferring it to a laptop!

ITAC will provide the most complete MPI analysis, but the trace file size will become unmanageable even for a moderate number of ranks and time intervals. Here we have to apply filtering options like filtering time intervals or functions or ranks. MPS will provide a good starting point for deciding which details we should investigate and which we can ignore.

Fig. 15.7 is a snapshot of the Poisson Solver that we will use later as a test example. A single iteration is zoomed out and function names have been instrumented. The collective operation `MPI_Allreduce` works like a barrier at its end. We will later see how to use Non-blocking collectives and better overlapping of computation and communication.

INTEL VTune ANALYZER

Intel® VTune™ Analyzer (VTune) is discussed in Chapter 14. Here we will briefly discuss the use in MPI programs and how to use the command line interface of VTune.

NOTE: A sample command line can be generated from within the GUI for following identical experiments without using the GUI.

Besides learning how to use VTune for MPI programs and collecting counter information, we need VTune for a decent analysis of the OpenMP part of Hybrid applications.

FIG. 15.7

Snapshot of the Poisson solver that we will use later as test example.

```
$ mpirun -n <N> \
-gtool "amplxe-cl -collect <mode> \
-r <resultdir>:<rank[,...]>" ./prg.x
```

The `gtool` parameter inserts the tool to be used. After the colon we may list ranks that will execute the tool selectively. Even more convenient than the additional flag is the environment variable `I_MPI_GTOOL`. It can be used as the flag making it unnecessary to touch the `mpirun` command line.

The ranks that are interesting for further analysis can be selected by investigating a previous MPS or ITAC analysis.

There are further Intel tools that can be used in combination with MPI programs. Correctness Checker is part of the ITAC package and can be used like ITAC by applying the `-check_mpi` flag instead of –trace and Intel® Inspector can be used like VTune with the `inpexe-cl` command line. These tools will check on coding errors like race conditions and wrong function parameters.

TUNING OF MPI APPLICATIONS

Before starting to tune a program it is obvious that we need a good analysis and an efficient starting point. The very first step is to check for the speedup and parallel efficiency of the program. If the efficiency drops below 50% it means that it is

wasting half of the available cycles due to sub optimal scaling. We may even encounter overall wall clock time increases above a certain number of ranks.

When we see the need for optimization, it is crucial to determine the reason for the suboptimal performance. For this case, we can take an advanced feature of ITAC called the idealizer. The idealizer takes a previously generated ITAC trace and simulates an ideal network while keeping the non MPI timings as they are. This ideal network is sending messages with infinite speed and zero latency. The internal times of MPI are also shrunk to zero. The remaining MPI time reflects the algorithmic deficits of the implementation like synchronization points and imperfect load balancing. See For More Information at the end of this chapter: "Overviews of dynamic load balancing."

A total program time simulated under the ideal network that is not much different from the measured time shows that we have to work on the MPI algorithm before we plan a network upgrade. If the total program time under the ideal network simulator is significantly less than the actual program time, we know that network performance is a major bottleneck. This can be addressed by upgrading the network or better adapting jobs to the existing network:

- Optimize the rank to node placement. Bulk transfer should take the fast connection
- Changing default behavior of Intel MPI message passing. (e.g., `I_MPI_EAGER_THRESHOLD`) by environment variables
- Adapting the algorithms of collective operations

PROCESS PLACEMENT

Rank to node process placement is necessary because the MPI launcher will by default place the first PPN (Processes Per Node) ranks on the first node, the next PPN ranks on the second node, and so forth. In a 2 or higher dimensional grid, it is very likely that too many neighbor ranks will be placed on different nodes and have to communicate through the slower inter node communication. We present this issue by showing a two dimensional process grid of a Poisson solver running on 16 MPI ranks with 4 ranks per node as shown in Fig. 15.8.

The upper left square shows the process grid with the default mapping of ranks and nodes. The upper right square shows an optimized rank distribution that minimizes the boundaries and communication between nodes.

For larger clusters, the linear distribution will have inner nodes with 10 exchanges to neighbor nodes while the 2×2 sub-grids have only 8 exchanges. This is just a small benefit but in the case of 16 ranks per node and linear mapping we get 34=2*16+2 exchanges versus 4*4=16 exchanges for 4×4 sub-grids. This ratio will grow with the number of processes per node. The mapping can be done by a `machinefile`. In the prior simple case, it will be lines as shown in Fig. 15.9.

We could have also used a Cartesian communicator provided by the MPI standard. This example is rather taken as an illustrative example. In more unstructured examples we could also benefit from a non-default ranks to nodes mapping but it is more involved to determine.

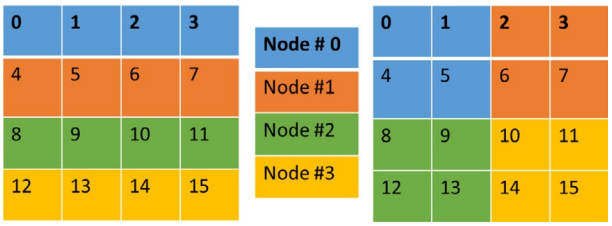


FIG. 15.8

Placing the processes running on the same node in a square sub grid will reduce the number of internode communications.

```
Node0:2
Node1:2
Node0:2
Node1:2
Node2:2
Node3:2
Node2:2
Node3:2
```

FIG. 15.9

This machinefile will generate the rank to node mapping for the example shown in Fig. 15.8 on the right side.

MPI RUNTIME SETTINGS

The Intel® MPI user and reference guide show a lot of environment variables that influence internal message passing settings. Default settings are tuned for most known hardware. Many tradeoffs have to be considered for these default settings, and we may see some situations where changing these parameters results in better performance. The most basic decision is the network fabric. The default used to be `I_MPI_FABRICS=shm:dapl`. This choice will lead to the use of shared memory for intranode communication and the Direct Access Programming Library* (DAPL*) layer for internode communication. In most cases the DAPL choice means using Infiniband. The Intel Omni-Path fabric (Chapter 5) was especially designed for HPC usage. It should be the best choice for supporting the integrated network adapters of Knights Landing. A good MPI setting for using Intel Omni-Path selection on Knights Landing is the OpenFabrics* Interface (OFI*):

```
$ export I_MPI_FABRICS=shm:ofi
$ export I_MPI_OFI_LIBRARY=<path>/libfabric.so
```

The `LD_LIBRARY_PATH` should also include the directory containing the Performance Scaled Messaging (PSM2) library `libpsm2.so`. These settings may change with new Intel® MPI releases. Consider OFI, for performance, if it is supported even if it is not set as default.

One of the most prominent candidates for fine-tuning is the environment variable `I_MPI_EAGER_THRESHOLD`. It determines the switching point for eager sends (small messages) and a rendezvous protocol for larger messages. The eager send is usually faster but it consumes more memory for buffering. For programs with a lot of messages slightly above the threshold it makes sense to raise the threshold. Defaults are documented inside the reference manual but may also depend on the platform. For Intel Omni-Path networks, `I_MPI_EAGER_THRESHOLD` is currently ignored but there is a similar variable `PSM2_MQ_RNDV_HFI_THRESH` provided by the PSM2 library.

MPI COLLECTIVE ALGORITHMS

Each collective is implemented with a number of algorithms that show different performance depending on the data sizes and the number of ranks. Intel MPI allows a fine tuned selection of algorithms for each data size and rank number by using the `I_MPI_ADJUST_<COLLECTIVE>` variable. The Intel MPI reference guide will have more details.

MPITUNE TOOL

The Intel MPI package contains a tool named `mpitune` that will test these parameters systematically. There is a mode for generic cluster optimization using the Intel MPI Benchmarks (`IMB`) as a test program and also a mode for testing an application. These tests will identify and save configuration values that may improve performance significantly beyond default values.

HETEROGENEOUS CLUSTERS

A homogeneous cluster using all Knights Landing processors will deliver its performance potential for applications with high vectorization efficiency and high thread scalability. But a real world legacy application might contain parts that may not apply to these preconditions. Intel Xeon processors are likely to be much more appropriate for sections of applications with low potential for vectorization and limited thread scalability. Building a cluster, which has both Intel Xeon processors and Intel Xeon Phi products, would seem to have the best of both worlds. While this concept is quite easy to comprehend, it demands sophisticated strategies for software design and load balancing. Clusters, based on Intel Xeon processors, equipped with Intel Xeon Phi coprocessors have been early examples of such heterogeneous clusters for some years now.

Such clusters are capable of running MPI programs in *symmetric mode* using all processors and coprocessors as nodes of a cluster. They are also able to run MPI programs on host processors with additional offload to the coprocessors. The symmetric MPI mode often suffered from low message passing performance between the host and coprocessor or even worse between coprocessors on different hosts. All communication had to go through the host and its PCIe bus. Especially collective operations like `MPI_Alltoall` performed suboptimally in this model.

Before Knights Landing existed, a different approach of heterogeneous clusters was realized in the Dynamical Exascale Entry Platform (DEEP) project cofunded by the European Commission and conducted by the Forschungszentrum Jülich JSC and Intel (see "For More Information" at the end of this chapter). This pioneering work has already supplied recipes and other learnings that immediately apply to machines which use both Intel Xeon processors and Knights Landing. The idea for DEEP was to combine an Intel Xeon processor based cluster with a *booster cluster* consisting of coprocessor nodes. The advantage of this approach is that the coprocessors have their own fast network without any bypass. The realization with coprocessors was difficult because a customized node had to be designed for an Intel Xeon Phi coprocessor based cluster. Fig. 15.10 shows the schema of the Deep cluster. A low scaling sub cluster consists of Intel Xeon processors and Intel Xeon Phi coprocessors with a fast Infiniband interconnect.

For Knights Landing, it should be easier to build and program a cluster consisting of both Intel Xeon processors and Knights Landing processors. The Intel® Omni-Path fabric (Chapter 5) is designed to work with both architectures and MPI usage will be very similar to the symmetric mode. One way to execute applications on heterogeneous networks would be:

1. Define a machine file containing host names in the intended order with one host per line and the number of ranks on this host. The same host may be used several times if an interleaved rank distribution is needed.
2. Define a wrapper script for the application with one branch per architecture. In this script, different environments for each architecture like the number of threads can be used.

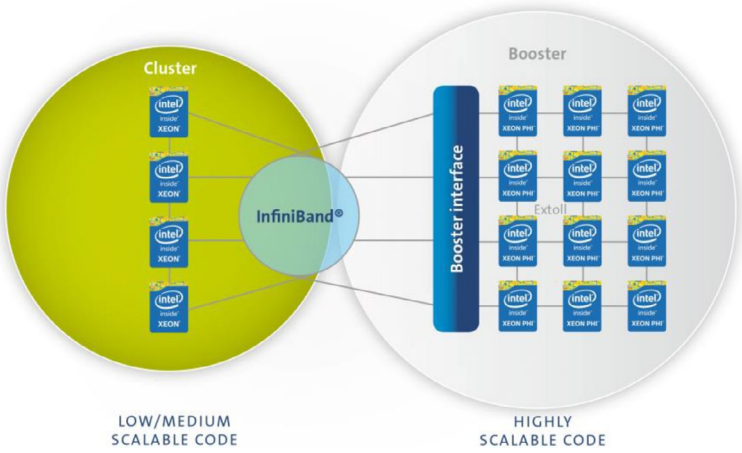


FIG. 15.10

Schema of the DEEP cluster.

In a batch environment the `machinefile` may be generated by a shell script that transforms the hostfile delivered by the batch manager into a machine file.

The next question is if we already have applications that could make use of such a heterogeneous environment? The Gromacs molecular dynamics application is a good example of such a program. For *Particle Mesh Ewald (PME) workloads*, the application can be divided into two parts:

1. The particle-particle part directly calculates the interaction of pairs of particles. This part is highly vectorized by using intrinsics. OpenMP scalability is also very good for this part which makes it ideal for running on Knights Landing.
2. The PME part calculates long-range forces in Fourier Space. The Fourier Transforms usually scale very well, also, but for distributed applications the necessary transpose operation carried out by `MPI_Alltoall` will scale much worse than the next neighbor exchanges in the PP part, which may favor running on Intel Xeon processors.

A recipe for running these workloads as described already exists in the Intel Developer Zone (see For More Information).

RECENT TRENDS IN MPI CODING

MPI programming methods are changing due to new functions available in the MPI standard as well as new network and computational capabilities. Hybrid MPI/OpenMP combinations became interesting with the first multiprocessor clusters. In the beginning, the coexistence of MPI communication and threads was not defined. But this was not really a problem because MPI communication and threading were strictly separated. Later, the MPI distributions became thread safe, enabling developers to mix the two parallel techniques for more interesting programs.

SIMPLE HYBRID CODING

The Poisson problem is a simple partial differential equation. More advanced problems often have similar communication patterns called next neighbor halo exchange. This makes it an ideal training example that is discussed in the *Using MPI* book referenced at the end of this chapter. Even the naïve approach shows that hybrid programming is outperforms pure MPI. The Poisson solver works on a rectangular grid of points. In the parallel version, each MPI rank gets a rectangular subset of the grid depending on the number of ranks assigned to rows and columns.

On the highest level of the implementation, shown in Fig. 15.11, we see an iteration loop consisting of a copy routine saving the grid points from the previous iteration for calculation of the residuum at the end of each iteration. The main computation is done inside the red and black routines. The red routine updates every

```
for(it=0; it < maxiter; it++)
{
  // copy last iteration state to backup array x_cur
  copy(&p,&gr);
  // RED update
  red(&p,&gr);
  // Boundary exchange (MPI communication)
  exchange(&p,&gr);
  // BLACK update
  black(&p, &gr);
  // Boundary exchange (MPI communication)
  exchange(&p,&gr);
  // Residuum (MPI global sum)
  resid = residuum(&p, &gr);
  if(resid < eps ) break;
} // iteration ends
```

FIG. 15.11

Poisson solver iteration loop. Compare this to the previous ITAC picture in Fig. 15.7.

second grid point in a two-dimensional grid. The black routine updates the interleaving points. After each red and black update a communication routine exchanges the boundaries. This methodology ensures perfect parallelism. The residuum measures the difference between the previous and the current solution. The main loops are additionally parallelized with OpenMP. Once the residuum falls under a user defined eps value, the program ends and prints out the final residuum and some performance characteristics like GFlops/sec, time, # number of threads per rank, and number of iterations. The red black methodology also ensures that results for different numbers of threads and MPI ranks are identical!

The exchange routine sends the boundary rows and columns in all four horizontal and vertical directions. The vertical part of the exchange routine (the horizontal part is similar) is shown in Fig. 15.12. Please note that the matching sends and receives have the same tags (100,200). The remote ranks are named gr->up and gr->down. For the boundaries without further up and down processors, these simply have the value MPI_PROC_NULL and therefore the call will be ignored.

OVERLAP OF COMMUNICATION AND COMPUTATION

In the implementation of the exchange routine shown in Fig. 15.12, sends and receives are located in a single routine. There is practically no potential for overlapping the communication with computation and hide the latency. This is not a serious problem when all ranks are running in lockstep. But in reality, small jitter in the OS might slow down the computation randomly and generate small load

```
// ---------  vertical exchange ---- //
  // receive row from above
  MPI_Irecv(gr->x_new[gr->lrow+1],gr>lcol+2,
            MPI_DOUBLE,gr>up,
            100,MPI_COMM_WORLD,&vert_req[1]);
  // send down first row
  MPI_Isend(gr->x_new[1],gr->lcol+2,
            MPI_DOUBLE,gr>down,
            100,MPI_COMM_WORLD,&vert_req[0]);
 // receive from below
  MPI_Irecv(gr->x_new[0],gr->lcol+2,
            MPI_DOUBLE,gr>down,
            200,MPI_COMM_WORLD,&vert_req[3]);
 // send up last row
  MPI_Isend(gr->x_new[gr->lrow],gr->lcol+2,
            MPI_DOUBLE,gr->up,
            200,MPI_COMM_WORLD,&vert_req[2]);
  MPI_Waitall(4, vert_req, vert_status);
```

FIG. 15.12

This is just the vertical part of the exchange routine (the horizontal part is similar).

imbalances. In these situations, it is good to have receives called earlier than the sends. The code from the main iteration loop, shown in Fig. 15.13, may now be transformed by splitting the exchange routine into a routine that starts the receives before the update happens; `exchange_start_recv` and a routine for sends and waits `exchange_send_wait`. This should remove the wait time in cases where receive is late due to OS jitter or systematic load imbalances as seen in Fig. 15.14. This is just a small step. It could be even thought of something like splitting the red (and the black) routine into a boundary update like `red_boundary` and a routine for the inner grid points `red_inner`. Finally, we can then write the code block as shown in Fig. 15.15.

We start sending the boundary before the inner points are computed in Fig. 15.15. This is why we calculate it first and then immediately send it. All communication should now overlap with computation of the inner points. Finally we have to make sure that all updated boundaries were received: `exchange_wait`.

In theory, this methodology should give us a perfect overlap if computation time of the inner points is not too short. This becomes even more important for solvers that use higher order stencils. These solvers have to communicate much more data for boundaries of more than just a single row/column width.

The reality might look different because for larger messages, the immediate sends may need a handshake from the matching receives first before they start. However, there is a variable of Intel MPI that lets it spawn an additional helper thread for

```
// RED update
  red(&p,&gr);
// Boundary exchange (MPI communication)
   exchange(&p,&gr);
```

FIG. 15.13

Exchange code, version one. All exchange of the boundaries is done inside exchange.

```
// start receive of exchange before the update
 exchange_start_recv(&p,&gr);
// RED update
 red(&p,&gr);
// Boundary exchange (MPI communication)
 exchange_send_wait(&p,&gr);
```

FIG. 15.14

Exchange code, version two, reduces wait time by starting the Irecvs early.

```
// start receive of exchange before the update
 exchange_start_recv(&p,&gr);
// RED boundary update
 red_boundary(&p,&gr);
// start to send boundary before working on inner points
 exchange_start_send(&p,&gr);
// RED inner point update
 red_inner(&p,&gr);
 // Finish boundary exchange (MPI communication)
 exchange_wait(&p,&gr);
```

FIG. 15.15

Exchange code, version three, overlap boundary communication with calculation of inner points.

```
$ export I_MPI_ASYNC_PROGRESS_PIN = <id1,[id2,…idN]>
$ export I_MPI_ASYNC_PROGRESS_PIN = auto
```

FIG. 15.16

Pinning the helper thread to logical cores (id1,…) or using a default pinning mechanism.

handling MPI calls asynchronously: `I_MPI_ASYNC_PROGRESS=1`. But in many hybrid MPI/OpenMP programs, we see a performance decrease when using this variable because the helper threads may interfere with the OpenMP threads.

Intel MPI supports helper thread pinning as shown in Fig. 15.16.

MPI CALLS ON THREADS

The best way to avoid interference of MPI communication with computation is achieved by calling MPI functions on additional threads. This was not possible for a long time because of the lack of thread safe implementations but for Intel MPI it is now safe to use this methodology! The most consequential usage of this feature is demonstrated by the Swift code from Durham University. This code was improved as part of the Intel Parallel Compute Centers (IPCC) program. The whole application is divided into tasks that are scheduled during run time to MPI ranks and threads spawned from these ranks. In this framework, MPI use is just another task that is scheduled to one of the threads (see For More Information for references to this work).

In our simple Poisson example, we separate MPI communication and computation by reserving one thread (0 = Master thread) for MPI and do the computation in parallel to MPI on the other threads. This could be either done manually by using the thread ids or by introduction of nested parallelism. The manual solution has the advantage of having full control, but the nested solution is easier to implement and does not need so many changes to the previous version. In the Poisson Solver, we simply placed the whole iteration loop as shown in Fig. 15.11 plus the changes for the overlap into a parallel region. This is also an improvement from the OpenMP perspective because of the potential to remove some of the implied synchronizations in Fig. 15.17.

NONBLOCKING COLLECTIVES

Collectives like the `MPI_Allreduce`, shown above, often act like a barrier synchronization and have the same negative impact on performance. Every fluctuation in run time on a certain rank will slow down the whole application because all ranks have to wait on the slowest rank. This could be avoided by finalizing the collective at a later point in program execution when all ranks have done their necessary internal message passing. The idea is quite similar to the above described overlap of communication and computation. The Nonblocking `MPI_Iallreduce` will just initiate the collective but it will also define an MPI request that must be finished by an `MPI_Wait` later. For having a reasonable overlap we may change the position of the copy routine inside the loop shown in Figs. 15.11 and 15.17. Without any change of algorithm we may just move the copy routine to the end of the loop. For the first iteration, we may just add another copy before the iteration loop. The transformation of the global sum portion is shown in Fig. 15.18.

ONE-SIDED SHARED MEMORY CALLS

One-sided routines are discussed in Chapter 16 (MPI-3 RMA). We could have also generated a version using MPI-3 shared memory programming as discussed in the book High Performance Parallelism Pearls (see section For More Information). The exchange routine can be optimized by using the shared memory MPI-3 semantics

```
#pragma omp parallel
{
  int it;
  int me = omp_get_thread_num();

  for(it=0; it < maxiter; it++)
  {
    if(me == 0)
      exchange_start_recv(&p,&gr); // prepare to receive
    else
      copy(&p,&gr);

#pragma omp barrier
    // reset resid after first barrier
    if(me == 1) resid = 0.0;

// RED boundary update
   if(me == 0)
   {  red_boundary(&p,&gr);
      exchange_start_send(&p,&gr);
   }
   //wait on boundary (for BLACK) and main RED update
    if(me==0)
       exchange_wait(&p,&gr);
   else
       red(&p,&gr);
 # pragma omp barrier
// BLACK update like the red update
…
// Residuum routine only for OMP part of reduction
  if(me)
   residdum(&p, &gr, &resid);
# pragma omp barrier
  if(me==0)   // MPI global sum
  {
   resid_tmp = resid;
MPI_Allreduce(&resid_tmp,&resid,1,MPI_DOUBLE,MPI_SUM,
                  MPI_COMM_WORLD);
  }
#pragma omp barrier
#pragma omp flush(resid)
 if(resid < eps ) break;
}}
```

FIG. 15.17

Iteration loop for the MPI on threads version.

for ranks that are located on the same compute node. Newer versions of Intel MPI use Knights Landing MCDRAM for improving the performance of these operations.

PUTTING IT ALL TOGETHER

In this last section, we will show some benchmarking results of the Poisson example showing the benefits of the tuning strategies described earlier. We used a preproduction Knights Landing processor with 64 cores. Fig. 15.19 shows the speedup of different implementations on a single Knights Landing. The Baseline (Speedup = 1) is defined as the performance for pure MPI with an MPI rank on each available core. The other versions are all hybrid MPI/OpenMP, and we pick the best combination

```
MPI_Request iall_request;
MPI_Status  iall_status
// part before global sum
…
if(me==0)   // MPI global sum
{
  resid_tmp = resid;
MPI_Iallreduce(&resid_tmp,&resid,1,MPI_DOUBLE,MPI_SUM,
                  MPI_COMM_WORLD, &iall_request );
  exchange_start_recv(&p,&gr); // for next iteration
  MPI_Wait(&iall_request,&iall_status)
 }
// move copy to this position - overlap with allreduce
if(me)
      copy(&p,&gr);

#pragma omp barrier
#pragma omp flush(resid)
if(resid < eps ) break;
}}
```

FIG. 15.18

Global sum implemented with nonblocking Iallreduce.

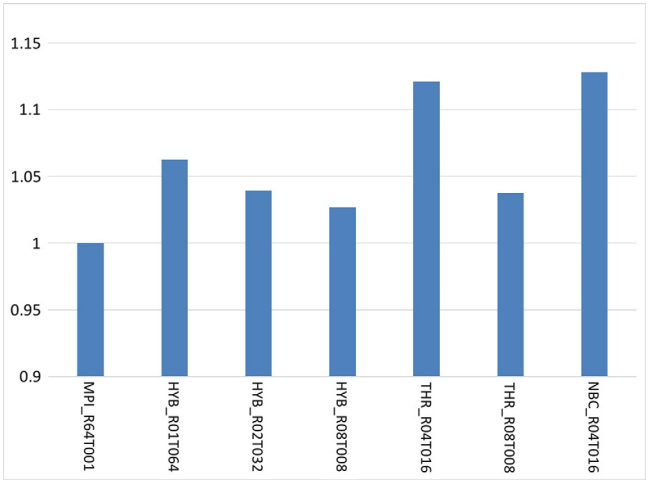


FIG. 15.19

Single node Poisson speedup, calculated vs. single node pure MPI version with 64 ranks and a single thread per rank (`MPI_R64T001`). HYB = naïve hybrid version Fig. 15.13, THR = MPI on threads version Fig. 15.17. NBC = final version with a nonblocking Iallreduce Fig. 15.18.

of MPI rank number and OpenMP thread number. Each combination was tested three times and the best numbers were taken. The computation grid is 3000×3000 causing bandwidth limitations for the single node results and showing some cache effects for multi node runs. The different versions are the naïve implementation in pure MPI mode

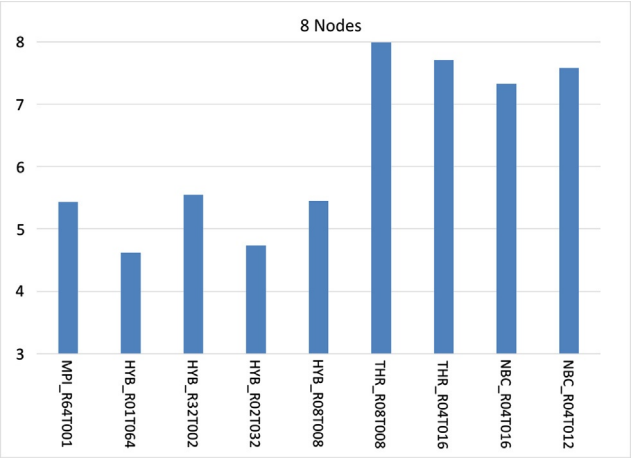


FIG. 15.20

Comparison of different MPI ranks/OpenMP threads combinations on 8 nodes. The speedup is calculated vs. the single node pure MPI result.

with 64 ranks and a single thread (`MPI_R64T001`) and best hybrid mode (`HYB_R01T64`), the thread MPI version (prefix THR) shown in Fig. 15.17, the thread MPI version with additional non-blocking collective (prefix NBC) shown in Fig. 15.18.

Fig. 15.20 shows the 8 node results. The versions are labeled by their configuration per node. The Pure MPI version (`MPI_R64T001`) is on 8 nodes equal to $64 \times 8 = 512$ ranks. The figure clearly shows that the MPI on threads versions (THR, NBC) outperform the pure MPI and the naïve hybrid versions. There seems to be additional potential for the NBC version because the R04T012 version outperforms the R04T016 version that makes use of all available cores. These optimizations will be performed and published in later blogs and articles.

Fig. 15.21 shows the speedup of these different versions compared to the single node pure MPI performance measured for the base version. Single node results may not reflect the situation of a cluster run because we are using shared memory communication and the effect of random OS jitter grows with the number of nodes.

The results show that performance of the simple naïve implementation can be optimized by applying methodologies for overlapping computation and communication. The strategy of latency hiding is as old as message passing itself but new implementations of MPI libraries make it possible to call MPI functions on threads and minimize the interference of the MPI progress engine with the computation.

On Knights Landing, we see more reproducible results with less fluctuation in timing when using the Intel Omni-Path fabric (Chapter 5) because it was designed especially for MPI usage. Another feature that is crucial for optimization is the fact that the Intel Omni-Path architecture uses processor cycles. For compute bound application it may be beneficial to put MPI threads on separate cores.

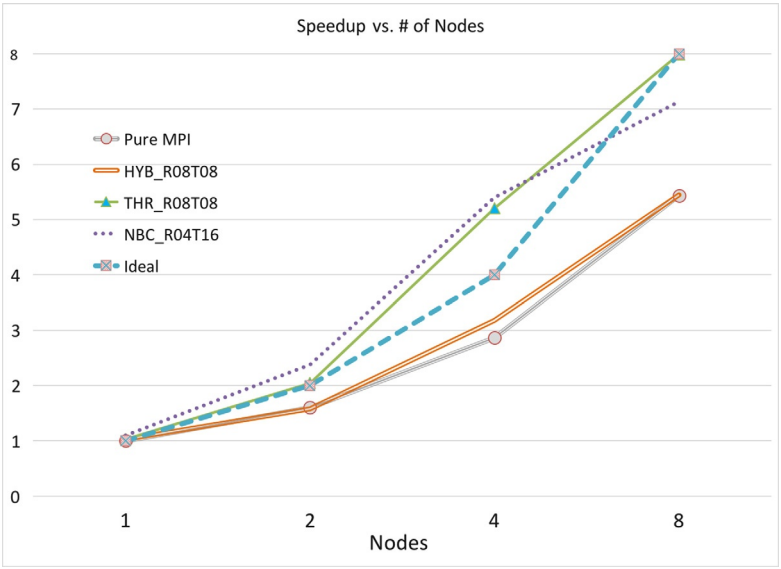


FIG. 15.21

Scaling of different implementations vs. number of Knight Landings. For 1–4 nodes the scaling is super linear due to strong scaling. The application is bandwidth bound for 1,2 nodes and becomes compute bound for 4, 8 nodes. Thread MPI (THR) version and the optimization with an additional non blocking collective (NBC) deliver by far the best results.

SUMMARY

The usage of MPI on Knights Landing is essentially the same as on Intel Xeon processor based systems. However, the characteristics of MPI performance may differ considerably. Pure MPI solutions may show good performance on Knights Landing, as well, but hybrid MPI/OpenMP solutions show some advantage even for naïve hybrid implementations. Part of this is due to the slower serial performance of Knights Landing compared to Intel Xeon processors. The Intel Omni-Path network depends on using processor cycles for achieving its best performance. As a result, it makes sense to think of techniques for separating MPI usage from the main computation by using MPI on separate threads running on dedicated cores. Nonblocking Collectives like `MPI_Iallreduce` provide a good way of gaining a further overlap of computation and communication.

FOR MORE INFORMATION

The MPI Forum (http://www.mpi-forum.org) develops and maintains the MPI standard. MPI-3, the latest major version of the standard, was ratified in Sep. 2012. Support for MPI-3 will start to appear in implementations in late 2012 and continue into 2013. None of the examples in this chapter use MPI-3 specific functionality.

A short list of popular and commonly available MPI implementations:

- Intel® MPI Library: http://www.intel.com/go/mpi
- MPICH2: http://www.mcs.anl.gov/research/projects/mpich2
- MVAPICH: http://mvapich.cse.ohio-state.edu
- OpenMPI: http://www.open-mpi.org

Excellent resources for more information on MPI programming:

- William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd ed. Cambridge, MA: MIT Press, 1999.
- Peter S. Pacheco. Parallel Programming with MPI. San Francisco, CA: Morgan Kaufman, 1996.
- *MPI-3 Shared Memory Programming Introduction*, High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, Chapter 16, Mikhail Brinskiy and Mark Lubin and James Dinan.

Some References mentioned in this Chapter:

- Jim Jeffers and James Reinders. Intel® Xeon Phi™ Coprocessor, High-Performance Programming. Chapter 12, MPI
- IPCC: http://software.intel.com/ipcc
- Swift Astro Physics code using the QuickSched library: http://www.swiftsim.com
- MPI Tutorial with examples from this Book, http://lotsofcores.com
- Mark Lubin and Heinrich Bockhorst. MPI Application Tune-Up, Four Steps to Performance. http://goparallel.sourceforge.net/wp-content/uploads/2015/02/MPI-Application-Tune-Up-r5.pdf
- GROMACS recipe for symmetric Intel® MPI using PME workloads: https://software.intel.com/en-us/articles/gromacs-recipe-for-symmetric-intel-mpi-using-pme-workloads
- Intel® MPI Library: Compatibility among Intel® Xeon® processors, Intel® Xeon Phi™ Coprocessors and Intel® Xeon Phi™ Processors x200, Loc Nguyen, http://lotsofcores.com/mpi200.

To learn more about the DEEP project, co-funded by the European Commission and conducted by the Forschungszentrum Jülich, refer to:

- http://www.deep-project.eu
- Eicker, N., Lippert, T., Moschny, T., Suarez, E., *The DEEP Project An alternative approach to heterogeneous cluster-computing in the many-core era. Concurrency Computat.: Pract. Exper., DOI: 10.1002/cpe.3562.*

Hands-on uses of dynamic load balancing with detailed discussions, source code sharing, and results with processors and Intel Xeon Phi coprocessors:

- Gilles Civario and Michael Lysaght. "Dynamic Load Balancing Using OpenMP 4.0." Chapter 11 of *High Performance Parallelism Pearls Volume One (2015)*.
- Vadim Karpusenko. "Heterogeneous MPI application optimization with ITAC." Chapter 25 of *High Performance Parallelism Pearls Volume One (2015)*.
- Ashraf Bhuiyan and Perri Needham and Ross C. Walker. "Amber PME Molecular Dynamics Optimization." Chapter 6 of *High Performance Parallelism Pearls Volume Two (2015)*.
- Enda O'Brien. "Coarse-Grained OpenMP for Scalable Hybrid Parallelism." Chapter 17 of *High Performance Parallelism Pearls Volume Two (2015)*.

Overviews of dynamic load balancing:

- Aaron Becker, Gengbin Zheng, and Laxmikant Kale. Load Balancing, Distributed Memory. *Encyclopedia of Parallel Computing*, David Padua, Ed., New York, NY: Springer-Verlag, 2011.
- Bruce Hendrickson and Karen Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering* 184(2–4):485–500, 2000.
- Vipin Kumar, Ananth Y. Grama, and Vempaty Nageshwara Rao. Scalable Load Balancing Techniques for Parallel Computers. *Journal of Parallel and Distributed Computing* 22(1):60–79, 1994.

CHAPTER

PGAS programming models

16

In this chapter, we take a look at Partitioned Global Address Space (PGAS) programming models, which scale across cores and nodes while preserving a shared memory-like programming model. While Knights Landing will be programmed mostly with MPI, OpenMP, and TBB, utilizing PGAS models will be increasingly important in the future. In this chapter, we show that PGAS is an effective programming model for the large number of cores on a Knights Landing.

What is new with Knights Landing in this chapter?

PGAS programming models are a good match for scaling across the many cores of Knights Landing and are likely to increase in popularity in upcoming years.

PGAS programming models provide a parallel programming environment where data at all locations in a parallel computing system can be made accessible for reads, writes, and atomic updates, even when data is located within a remote node. PGAS models logically partition data addresses based on its physical location, and we can query the location of a particular element to optimize data access costs. Implementations of PGAS concepts as both languages and libraries have been successful in a range of application domains, particularly those that access large data sets in irregular patterns or where asynchronous communication is important.

TO SHARE OR NOT TO SHARE

Multithreaded programming models like OpenMP are *shared-memory* programming models, wherein data is visible to all processing elements unless specifically allocated privately. A *processing element* (PE) refers to a thread or process that is executing its own stream of instructions; MPI programmers refer to PEs as *ranks*. This model is convenient, but may make it difficult to write correct programs, because every datum is potentially subject to race conditions and concurrency bugs. Furthermore, on modern systems, nonuniform memory access (NUMA) latencies lead to performance artifacts when data is allocated to threads with incorrect affinity. NUMA stands for nonuniform memory access, which results from hierarchical memories, including caches and multiple memory controllers. Such data locality concerns are challenging to address through most shared memory models because of a

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00016-8

combination of dynamic factors including forking and joining threads as well as dynamic scheduling. Thus, multithreaded programming models make writing simple parallel constructs easy, but we often find that more explicit control over locality and data movement is needed for complex programs and for achieving high parallel efficiency.

Programming models such as MPI (Message Passing Interface), that were designed to support distributed systems, have private data for every PE, and explicit communication is required to exchange data between PEs. For example, typical MPI programs pass messages via *Send* and *Receive* to exchange data between pairs of PEs, and use collective generalizations thereof (e.g., *Broadcast*) when many-to-many patterns are required. In this model, the sending and receiving PEs each provide information about the memory that is involved, the communication partner, and the communication context; matching operations must implicitly synchronize before the receiver can use the data. The need to describe data movement explicitly is both a boon and a burden on us as programmers. Incorrect MPI programs may deadlock although it is straightforward to reason about both performance and correctness in this two-sided (i.e., sender- and receiver-oriented) communication model.

PGAS programming models support a view of memory that has aspects of both shared and private data. The model supports a global address space (GAS) that is similar to shared memory, as it allows for data to be configured such that any PE can access it directly. However, data is updated in-place rather than relying on coherent replicas and data consistency typically follows a more relaxed model that may require the programmer to "publish" values to make them visible to others. In some models, this direct access is facilitated through the programming language by referencing or assigning to data in the GAS, while in others, it is facilitated through calls to one-sided (i.e., initiator-oriented) read, write, and update functions.

The GAS is partitioned such that each PE owns a portion of the data, and this portion is said to have affinity to the PE that owns it. In Fig. 16.1, the data *A* is part of the GAS and can be accessed by any thread, while *B* and *C* are private data. In most PGAS models, one must explicitly indicate the PE to which the desired data has affinity, such that the data locality information is clearly conveyed within the source code. For example, the array access `B = A[me];` involves data with local affinity (to my PE "me"), while `B = A[me+1];` requires reading a value of *A* with affinity to a different PE, potentially across a network or in a different NUMA domain.

The explicit partitioning of data across PEs makes it easier for us to reason about locality, which is essential to performance, both within a node (due to NUMA) and between nodes (because network transactions behave differently than local memory access).

Another important characteristic of PGAS is that applications always run in parallel. There is an implicit forking of PEs at programming launch and an implicit join at program termination, but from our perspective, the PEs persist throughout the lifetime of the application. Such persistence is not strictly true if one considers Chapel or

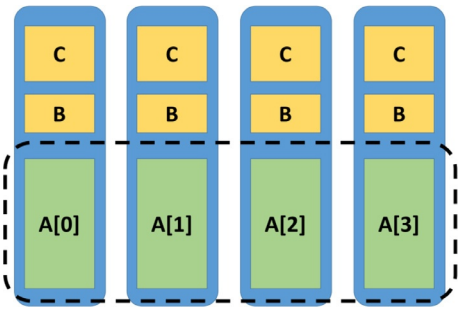


FIG. 16.1

In the PGAS programming model, both private and shared (globally addressable) data can be allocated. Here the objects *A* on each PE can be accessed from other PEs, whereas each copy of *B* and *C* cannot. In most PGAS implementations, there is a special memory allocator for globally addressable data.

HPX, both of which support the PGAS concept, but have a more dynamic execution model. This is in contrast to multithreading, wherein there can be numerous instances of fork and join within a single program. While multithreading does not *require* us to fork and join threads more than once, in practice, the majority of applications will fork and join many times. This can lead to a noticeable overhead and also limit scalability as serial sections and fork-join overheads can account for a larger fraction of the execution time with increasing scale.

While advocates of multithreading often place blame for fork-join overheads and serialization on programmers, this is not entirely accurate, because one of the consequences of multithreaded programs existing within a single address space is that it is very difficult to avoid race conditions. In a program with a single fork and join, every access to mutable data that is not explicitly made private must be performed through an atomic operation or protected by a critical section or other synchronization primitive. This includes protecting all library calls that are not reentrant, or by replacing such calls with reentrant alternatives, e.g., rand() becomes rand_r(), where the latter uses thread-private state. As many programmers lack total knowledge of potential data races in their applications as well as all of its dependencies, the simple solution is to restrict multithreading to code that is known to be thread-safe, and to fork and join just for this epoch.

Because PGAS data is private by default, it does not have the same synchronization challenges. Each PE lives in its own address space and can call library functions that are not reentrant, as the mutable data is not shared between PEs. Only when the programmer performs unsafe accesses to explicitly shared data in the globally addressable space will a potential race arise, which localizes the problem of debugging to the code that the programmer is touching. Thus, PGAS has many of the advantages of shared-memory when it comes to directly accessing distributed, shared data, while at the same time not suffering from several challenging pitfalls of multithreading/shared-memory related to fork-join/serialization overheads and race

conditions. These benefits derive from the PGAS *private by default*, *shared when requested* data model, versus the multithreading *shared by default*, *private when requested* data model.

WHY USE PGAS ON KNIGHTS LANDING?

Knights Landing has a large number of cores within a single coherent domain, which makes global addressability trivial. However, since exploiting locality is critical for performance on any multicore architecture, the explicit data partitioning implied by the PGAS model makes it easier to optimize for data locality, for example, by blocking data accesses to exploit L2 caches, as there will be no false-sharing between data owned by different PEs.

As explained in the previous section, multithreaded programming can lead to a large number of fork-join epochs. The cost of fork and join can be significant and increase with the number of threads. Even for highly optimized systems where this cost has been improved upon, serial sections represent an impediment to efficient scaling (due to Amdahl's Law), as the serial section performance is not improved by utilizing additional cores. Thus, overheads from frequent fork-join and scalability limitations from serial sections can become serious problems on a processor capable of running hundreds of threads.

Compared to the previous generation Intel® Xeon Phi™ product, Knights Corner, Knights Landing has substantial advantages from a PGAS perspective. Knights Corner was a coprocessor attached to a host processor via a PCI bus, which meant that moving data between Knights Corner coprocessors on different nodes entailed traversal of the PCI bus on both ends, in addition to the network traversal between nodes. In contrast, Knights Landing does not require a separate host, meaning that communication between different Knights Landing processors incurs no additional data movement cost relative to an Intel® Xeon® processor. Knights Landing has the same ability to exploit fast networks like Intel® Omni-Path Fabric (Chapter 5), Mellanox InfiniBand, or the Aries network found in Cray XC supercomputers. Some Knights Landing processors will feature integrated, on-package, fabric support.

Relative to message passing, PGAS communication, that is, moving data using explicit or implicit one-sided communication operations, is potentially less processor-intensive. This is because the message-passing model, at least as implemented in MPI, utilizes a more complex protocol that is typically supported through software processing to match send and receive operations. In contrast, PGAS one-sided read and write (a.k.a. *get* and *put*) operations have all the information necessary at the initiating PE to write directly to application buffers and generally do not require message ordering. When coupled with a network that supports Remote Direct Memory Access (RDMA), such as the Aries network, Knights Landing will likely delivery lower latency and higher message-rate with Put and Get operations than with Send and Recv. The importance of reducing software processing in network communication is particularly critical with the Knights Landing processor because of the trade-offs favoring total compute capability over single-threaded performance.

PROGRAMMING WITH PGAS

PGAS programming models are not new, although they may be unfamiliar to many programmers due to their historical connection with special purpose supercomputers. The first well-known PGAS model was SHMEM, which was developed for the Cray T3D supercomputer and has since been renewed through a community effort to create a modern OpenSHMEM specification. While OpenSHMEM is a library that can be used from C or Fortran applications, other PGAS models are implemented as language features and require compiler support. Unified Parallel C (UPC) and Fortran coarrays are two such popular examples. The important features of these models are demonstrated through examples in the following sections.

OpenSHMEM

Because OpenSHMEM (Fig. 16.2) is implemented as a library, writes to globally addressable data are performed with *put* functions, rather than simple assignments, as we will see in a language implementation. The explicit nature of OpenSHMEM—particularly that put operations have the type encoded in the function name—may be aesthetically unpleasing to some, but it enables a very high-performance implementation because each operation can be mapped directly to hardware in a platform that supports global addressing or RDMA, because put calls are mapped directly to remote stores or network operations.

OpenSHMEM is a relatively restrictive programming model because of the desire to map its functionality directly to hardware. One important limitation is that all globally addressable data must be *symmetric*, which means that the same global variables or data buffers are allocated by all PEs. Symmetric memory is allocated through a symmetric heap allocator that is invoked collectively (i.e., from all PEs) and symmetrically (with the same arguments on all PEs). Any static data is also guaranteed to be symmetric. This ensures that the layout of remotely accessible memory is the same at

```
#include <shmem.h>
int main(void) {
  shmem_init();
  if (num_pes()<2) shmem_global_exit(1);
  /* allocate from the global heap */
  int * A = shmem_malloc(sizeof(int));
  int B = 134;
  /* store local B at PE 0 into A at PE 1 */
  if (my_pe()==0) shmem_int_put(A,&B,1,1);
  /* global synchronization of execution and data */
  shmem_barrier_all();
  /* observe the result of the store */
  if (my_pe()==1) printf("A@1=%d\n",*A);
  shmem_free(A);
  shmem_finalize();
  return 0;
}
```

FIG. 16.2

A simple OpenSHMEM program, written according to the OpenSHMEM 1.2 specification.

all PEs and enables efficient implementations. It does, however, limit the programmer's ability to use subsets of processes with divergent control flow, which limits the types of applications that can be expressed in OpenSHMEM.

UPC

Unified Parallel C (UPC) is an extension to C99. The most important language extension is the *shared* type qualifier. Data objects that are declared with the shared qualifier are accessible by all threads, even if those threads are running on multiple nodes. An optional layout qualifier can also be provided as part of the shared array type to indicate how the elements of the array are distributed across PEs. Since processor load and store instructions cannot access data stored on remote nodes, UPC requires a runtime that handles remote memory access via the appropriate network operations.

UPC differs from OpenSHMEM in a number of ways, not the least of which is that, because UPC is a language and has compiler support, the assignment operator (=) can be used to perform remote memory access. Pointers to shared data can also themselves be shared, allowing the programmer to create distributed, shared linked data structures (e.g., lists, trees, or graphs). However, because compilers may not always recognize bulk data transfers, UPC provides functions (upc_memput, upc_memget, upc_memcpy) that explicitly copy data in and out of globally addressable memory. Another key difference is that UPC can allocate globally addressable data in a nonsymmetric and noncollective manner, which increases the flexibility of the model and can help to enable alternatives to the conventional bulk-synchronization style of parallelism.

Fig. 16.3 shows a simple UPC program. We use the collective function upc_all_free instead of the local function upc_free to match the other PGAS models, which only have collective deallocation.

```
#include <upc.h>
int main(void) {
  if (THREADS<2) upc_global_exit(1);
  /* allocate from the shared heap */
  shared int * A = upc_all_alloc(THREADS,sizeof(int));
  int B = 134;
  /* store local B at PE 0 into A at PE 1 */
  if (MYTHREAD==0) A[1] = B;
  /* global synchronization of execution and data */
  upc_barrier;
  /* observe the result of the store */
  if (MYTHREAD==1) printf("A@1=%d\n",A[1]);
  upc_free(A);
  return 0;
}
```

FIG. 16.3

A simple UPC program (written per UPC v1.3 specification).

```
program main
implicit none
integer, allocatable :: A(:)[:]
integer :: B
if (num_images()<2) call abort;
! allocate from the shared heap
allocate(A(1)[*])
B = 134;
! store local B at PE 0 into A at PE 1
if (this_image().eq.0) A(1)[1] = B;
! global synchronization of execution and data
sync all
! observe the result of the store
if (this_image().eq.1) print*,'A@1=',A(1)[1]
deallocate(A)
end program main
```

FIG. 16.4

A simple Fortran program (written per Fortran 2008 specification).

FORTRAN COARRAYS

The concept of Fortran coarrays (Fig. 16.4) was developed as an extension to Fortran 95 and has been standardized in Fortran 2008. Thus, the current version of Fortran is itself a PGAS language, although this is not widely known. An optional codimension attribute can be added to Fortran arrays, allowing remote access to the array instances across all *images*, or PEs, executing the program. When referencing a coarray, an additional codimension is specified using square brackets to indicate the image in which the array locations will be accessed. In comparison with UPC, specifying globally addressable data in Fortran coarray is restrictive; however, this restricted model was carefully chosen to support the bulk of applications, while making compiler and runtime optimizations tractable. Fortran programmers are nothing if not obsessed with performance!

MPI-3 RMA

The MPI community first introduced one-sided communication, also known as Remote Memory Access (RMA), in the MPI 2.0 standard. MPI RMA defines a library PGAS model with functions for exposing memory for remote access through RMA *windows*, for reading and writing globally addressable data, and for synchronizing accesses across PEs. The MPI-2 RMA interface was designed to be maximally portable, even to systems without a fully coherent memory subsystem. However, several critical issues were identified with MPI-2 RMA, including restrictive synchronization semantics, lack of atomic operations, and a restrictive memory model. To address these issues, the MPI Forum revised RMA in MPI 3.0, adding new atomic operations, synchronization methods, methods for allocating and exposing remotely accessible memory, a new memory model for cache-coherent architectures, and several other features.

Unfortunately, the MPI-3 RMA interface remains complex, particularly due to the large number of features, which are aimed at supporting a wider range of usages than

most PGAS models. One way to mitigate the complexity of MPI-3 RMA is to use a higher-level interface such as Global Arrays, OpenSHMEM, or Fortran coarrays, as there is an implementation of each of these that uses MPI-3 RMA under the hood. However, if one intends to use MPI-3 RMA directly, it is useful to remember that MPI aims to provide programmers with the necessary and sufficient building blocks for designing complex parallel applications, which sometimes means providing more functions (and function arguments) than any one programmer or application needs. Just as with MPI message passing programs, there is a set of approximately 10 functions that are sufficient to write a wide range of PGAS-style programs in MPI-3; most of these functions are used in the example program in Fig. 16.5 (important missing functions include MPI_Get, MPI_Fetch_and_op and MPI_Compare_and_swap).

PGAS VERSUS MULTITHREADING

OpenMP is a simple, evolutionary programming model that can be used to transform existing serial or MPI-only programs to use multicore systems without increasing the memory footprint of the application at the same rate as multiprocessing (e.g., MPI) approaches. This is important for Knights Landing, as its number of cores per node is increased significantly relative to Intel Xeon processors, while the amount of memory remains comparable.

```
#include <mpi.h>
int main(void) {
  MPI_Init(NULL,NULL);
  int me,np;
  MPI_Comm_rank(MPI_COMM_WORLD,&me);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  if (np<2) MPI_Abort(MPI_COMM_WORLD,1);
  /* allocate from the shared heap */
  int * Abuf;
  MPI_Win Awin;
  MPI_Win_allocate(sizeof(int),sizeof(int),MPI_INFO_NULL,
                   MPI_COMM_WORLD,&Abuf,&Awin);
  MPI_Win_lock_all(MPI_MODE_NOCHECK,Awin);
  int B = 134;
  /* store local B at PE 0 into A at PE 1 */
  if (me==0) {
    MPI_Put(&B,1,MPI_INT,1,0,1,MPI_INT,Awin);
    MPI_Win_flush(1,Awin);
  }
  /* global synchronization of PEs */
  MPI_Barrier(MPI_COMM_WORLD);
  /* observe the result of the store */
  if (me==1) printf("A@1=%d\n",*Abuf);
  MPI_Win_unlock_all(Awin);
  MPI_Win_free(&Awin);
  MPI_Finalize();
  return 0;
}
```

FIG. 16.5

A simple MPI RMA program (written per MPI-3.1 specification). MPI_MODE_NOCHECK implies that only one synchronization motif will be used in a given epoch, which is true in PGAS motifs.

In spite of its popularity, crafting highly scalable OpenMP code poses significant challenges. At scales of one hundred or more threads in a NUMA system, simple approaches to OpenMP parallelization can encounter synchronization, serialization, and load imbalance bottlenecks requiring advanced tuning that is neither simple nor evolutionary. In these scenarios, scalable OpenMP requires careful management of shared and private data, which means code changes in the initialization of program state, not just in kernels. Furthermore, data must be initialized with the right affinity, which requires the programmer to understand and account for how the operating system virtual memory system maps allocations (e.g., Linux first-touch page faulting). Given that significant code changes may be necessary for scaling, it is worthwhile to evaluate how OpenMP compares to PGAS from the standpoint of supporting the programming features that enable high degrees of scaling.

Let us consider the toy program used to demonstrate the syntax of various PGAS models from earlier in this chapter, but written in naïve OpenMP (see Fig. 16.6). As with many codes where OpenMP is added after the initial design, this program allocates memory in the serial portion of the program. If it were initialized there, the memory would have affinity to the main thread. Of course, in this toy program, all of the data fits into a single memory page. However, if we extrapolate to a larger scale with more data, all of those pages will have affinity to the main thread, reducing available bandwidth, because a single memory controller manages all memory traffic, and incurs additional NUMA penalties.

Associating the shared data allocation with each thread in a NUMA-aware fashion is much more complicated. Modifying the OpenMP program to allocate memory with affinity to each thread adds significant complexity (see Fig. 16.7). While allocating memory in a serial region and faulting it in a parallel region will usually impart the right affinity, the safest method is to allocate data private to each thread. However, for that data to be accessible by the other threads, as it would be in a PGAS program, the pointers must be shared. Thus, one ends up allocating a vector of pointers, one for

```
#include <omp.h>
int main(void) {
  int np = omp_get_max_threads();
  if (np<2) exit(1);
  /* allocate data from the heap */
  int * A = malloc(np*sizeof(int));
  #pragma omp parallel shared(A)
  {
    int me = omp_get_thread_num();
    int B = 134;
    /* store local B at PE 0 into A at PE 1 */
    if (me==0) A[1] = B;
    /* global synchronization of execution and data */
    #pragma omp barrier
    /* observe the result of the store */
    if (me==1) printf("A@1=%d\n",A[1]);
  }
  free(A);
  return 0;
}
```

FIG. 16.6

The OpenMP program that is roughly equivalent to the template used for the PGAS examples.

```
#include <omp.h>
int main(void) {
  int np = omp_get_max_threads();
  if (np<2) exit(1);
  /* allocate shared pointers */
  int ** A = malloc(np*sizeof(int*));
  #pragma omp parallel shared(A)
  {
    int me = omp_get_thread_num();
    /* allocate per-thread data */
    A[me] = malloc(sizeof(int));
    #pragma omp barrier
    int B = 134;
    /* store local B at PE 0 into A at PE 1 */
    if (me==0) A[1][0] = B;
    /* global synchronization of execution and data */
    #pragma omp barrier
    /* observe the result of the store */
    if (me==1) printf("A@1=%d\n",A[1][0]);
    free(A[me]);
  }
  free(A);
  return 0;
}
```

FIG. 16.7

This OpenMP program is closer to the PGAS semantic, due to thread-local allocation of data.

each thread, to store the location of each thread's data. While this is easy to do in such a simple program, it may be difficult to reorganize performance-critical data structures in a large scientific application to fit this pattern. Multithreaded computation is sometimes far away from the allocation of the associated data, and in some cases, the allocation is done in a different programming language than the computation, making it more difficult to replace one array with a vector of arrays.

The more general form of NUMA-aware allocation for OpenMP programs is shown in Fig. 16.8. Unfortunately, OpenMP 4.5 does not provide a memory allocator except for the target features (these are primarily aimed at offloading computation to a coprocessor), so the user must create their own. This motif may not be practical for all applications.

In contrast to OpenMP, the PGAS has no trouble allocating shared data across threads with affinity, since allocators found in OpenSHMEM, UPC, Fortran 2008, and MPI-3 RMA all do this in a single function call. Furthermore, the NUMA-adverse allocation of shared data in serial regions is harder to realize in PGAS, due to the absence of a fork-join execution model.

PERFORMANCE EVALUATION

The most important differences between PGAS, multithreading, and message passing are semantic, and one should choose the model that is the best match for the algorithm to be implemented. However, it is useful to compare performance across

```
#include <omp.h>
void ** ompx_calloc(size_t bytes)
{
  int np = omp_get_max_threads();
  void ** ptrs = malloc(np*sizeof(void*));
  #pragma omp parallel shared(ptrs)
  {
    int me = omp_get_thread_num();
    ptrs[me] = malloc(bytes);
    memset(ptrs[me],0,bytes);
  }
  return ptrs;
}
void ompx_free(void ** ptrs)
{
  #pragma omp parallel shared(ptrs)
  {
    int me = omp_get_thread_num();
    free(ptrs[me]);
  }
  free(ptrs);
}
int main(int argc, char* argv[]) {
  int n = (argc>1) ? atoi(argv[1]) : 1<<20;
  int np = omp_get_max_threads();
  if (np<2) exit(1);
  int ** A = (int**)ompx_calloc(n*sizeof(int));
  #pragma omp parallel shared(A)
  {
     /* threaded computation */
  }
  ompx_free((void**)A);
  return 0;
}
```

FIG. 16.8

This OpenMP program is closer to the PGAS semantic, due to thread-local allocation of data. However, because OpenMP does not guarantee that thread affinity is preserved across parallel regions, ompx_calloc may not achieve the desired result.

models in order to understand the trade-offs between performance and programmability. Figs. 16.9 and 16.10 show the performance of Transpose from the Parallel Research Kernels (PRK) on a dual-socket Intel E5-2699v3 system and a 68-core KNL B0 system, respectively. The PRK contain implementations of important kernels, or application patterns, in a wide range of programming models. The performance data shown was obtained using Intrepid GCC UPC 5.2.0 on Intel Xeon processors and Berkeley UPC 2.22.0 with the shared-memory conduit (denoted UPC in the figures) and OSHMPI (OpenSHMEM over MPI-3, with shared-memory optimizations) on Knights Landing for OpenSHMEM (denoted SHMEM in the figures). All models used the Intel C/C++ 16.0 compilers and Intel MPI 5.1.1. Berkeley UPC 2.22.0 with the Intel compiler produced similar results to GCC UPC on the Intel Xeon processors; compiler optimizations do not appear to be a significant source of performance differences.

PRK Transpose is written to avoid OpenMP runtime overheads and allocates memory in a NUMA-aware fashion; it does not suffer from the pitfalls noted previously. As

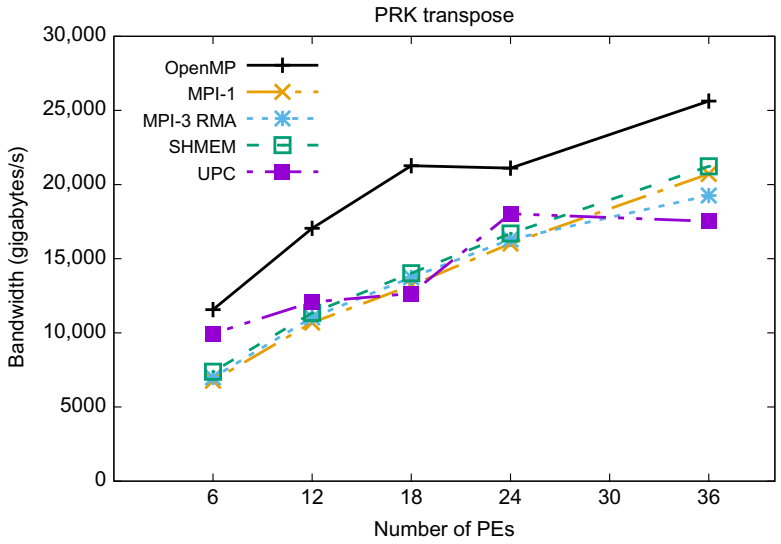


FIG. 16.9

Performance comparison of different models using Transpose from the Parallel Research Kernels (PRK) on an Intel Xeon processor (Haswell). These results were obtained with a matrix of dimension 11520, a tilesize of 32, and 10 iterations.

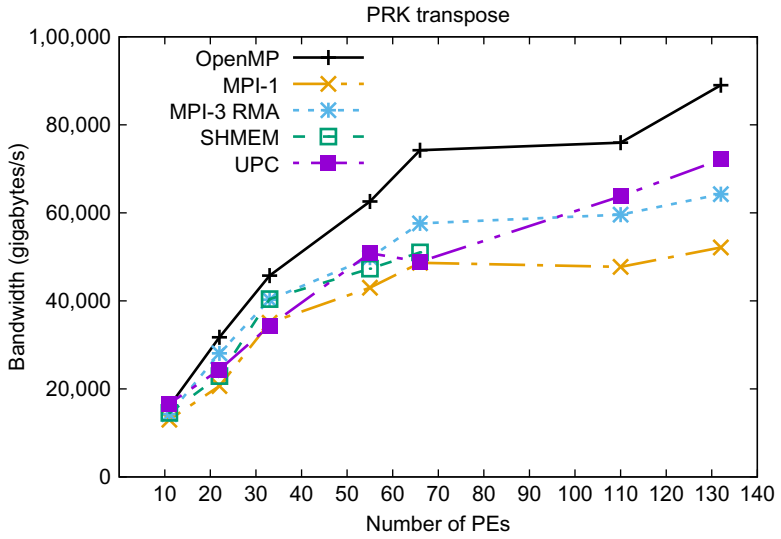


FIG. 16.10

Performance comparison of different models using Transpose from the Parallel Research Kernels (PRK) on Knights Landing. These results were obtained with a matrix of dimension 11520, a tilesize of 32, and 10 iterations.These results were obtained with a matrix of dimension 21120, a tilesize of 32, and 10 iterations. All data was allocated in MCDRAM using numactl –m 1. SHMEM performance drops with more than one PE per core because of an apparent implementation defect.

one might expect, it performs the best within a shared memory environment, since it does no unnecessary memory copying. UPC, OpenSHMEM (SHMEM in the figures), and both MPI implementations use an extra copy, since this is essential for the distributed memory implementation (PRK Transpose performs $B += A^T$; without the accumulation, the UPC implementation could avoid the copy). The distributed memory implementations of the PRK perform roughly the same, or lower than, OpenMP due to the additional data movements. Of course, when running across multiple nodes, these copies are required; in this context, OpenMP can only be used in conjunction with MPI or another distributed model.

Because the PRK transpose is relatively sensitive to runtime parameters, these results are only illustrative of possible performance with different models on Intel Xeon and Xeon Phi processors, but should not be treated as benchmarks. Please download the PRK and perform experiments with a range of problem sizes and runtime parameters to get a more complete understanding of the capability of these models on different platforms.

Fortran coarray results are not included here because there is not a widely available optimized implementation. However, the Cray Fortran compiler delivers excellent performance with coarray programs (including PRK transpose) for those that have access to such systems.

BEYOND PGAS

So far, we have focused on traditional PGAS models, which have a static execution model (fixed number of PEs throughout the lifetime of the application). However, globally addressable data is not limited to this execution model. As two examples, Chapel and HPX both provide many of the features of PGAS from a data perspective, while supporting more dynamic execution models that allow the number of PEs to vary. This flexibility can be exploited to implement dynamic load balancing or express parallel algorithms that have varying amounts of concurrency, such as the multigrid method. On the other hand, dynamic execution models often possess some of the pitfalls of fork-join execution; thus, we must be careful to avoid bottlenecks in the spawning of PEs (often implicitly, hidden behind task spawning).

While it is unclear if dynamic execution models are appropriate for a large fraction of high-performance computing workloads, Knights Landing provides a useful platform for evaluating them, due to its general purpose nature and substantial hardware parallelism.

SUMMARY

In this chapter, we provided an overview of the PGAS parallel programming paradigm and demonstrated a number of different implementations, including both libraries (OpenSHMEM and MPI-3) and languages (UPC and Fortran 2008). We contrasted the PGAS model with MPI and OpenMP, focusing on the challenges

associated with fork-join execution and NUMA-aware memory allocation. We also showed that PGAS programming models deliver results similar to MPI-1 for shared-memory, although OpenMP is the clear winner over both MPI and PGAS when written carefully to avoid both fork-join and NUMA overheads. Of course, when scaling to multiple nodes, OpenMP alone is inadequate, and one must hybridize it with MPI or another model, which can introduce significant complexity.

While PGAS programming models have been around for more than two decades, recent hardware trends—both multicore processors and networks that support direct access to remote memory—suggest that they will be supported more efficiently in upcoming platforms. Furthermore, now that the PGAS concept is present in the MPI-3 standard, software support for PGAS programming is available on essentially every platform, rather than requiring a combination of special hardware and software. Finally, the long-standing question—"which is better: MPI or PGAS?"—has been answered decisively: *MPI is PGAS*.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this chapter.

- Using Advanced MPI: Modern Features of the Message-Passing Interface, by William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk.
- *UPC*: *Distributed Shared-Memory Programming*, by Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick.
- Programming Models for Parallel Computing, edited by Pavan Balaji.
- The OpenSHMEM website, http://openshmem.org/.
- The GCC Coarray wiki, https://gcc.gnu.org/wiki/Coarray, and references therein.
- *Implementing OpenSHMEM using MPI-3 one-sided communication*, by Jeff R. Hammond, Sayan Ghosh, and Barbara M. Chapman, http://rd.springer.com/chapter/10.1007%2F978-3-319-05215-1_4.
- *MPI-3 Shared Memory Programming Introduction*, High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, Chapter 16, Mikhail Brinskiy and Mark Lubin and James Dinan.
- Parallel Research Kernels, https://github.com/ParRes/Kernels.
- Chapel, http://chapel.cray.com/.
- The STE||AR Group: HPX, http://stellar.cct.lsu.edu/projects/hpx/.
- Download the code from this, and other chapters, http://lotsofcores.com.

CHAPTER

Software-defined visualization

17

Software-defined visualization (SDVis) on Knights Landing highlights how and why visualization of large data sets is best done on processors. This chapter highlights three key open-source libraries that are fundamental for SDVis work (i.e., OpenSWR, Embree, and OSPRay). We can use these libraries *now* to benefit from the SDVis capabilities of Knights Landing.

What is new with Knights Landing in this chapter?

Large System Memory, MCDRAM, AVX-512 and Intel Omni-Path fabic enables efficient rendering for highly scalable visualization.

In this chapter, we discuss how Knights Landing processors deliver high-performance, interactive, and high-fidelity 3D image rendering. Fig. 17.1 shows one example of how high-fidelity SDVis solutions can aid data analysis. The dataset (over 1.2 Terabytes of data) was rendered all in software live at the SC'15 conference on a Knights Landing preproduction cluster, interactively, at about 15 frames per second.

We will discuss the motivation to pursue SDVis solutions and then describe the SDVis architecture. Throughout the chapter, we stress the Knights Landing architecture elements that exemplify how very large, complex scientific visualization (SciVis) workloads can best be managed to maximize insight. Noting that the SDVis

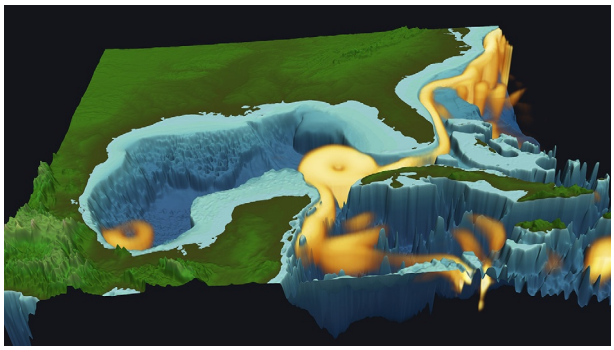


FIG. 17.1

Software-defined visualization of ocean climate on the U.S. Southeastern seaboard with data from MPAS-Ocean simulation. The highlighted portion near the coasts represents the ocean kinetic energy (i.e., vortex currents). Visualization application: Kitware ParaView 5.0.

Data Source: MPAS-Ocean/ACME, U.S. Dept. of Energy; visualization: TACC, UT-CAT.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00017-X

on Knights Landing is rapidly expanding and evolving, will provide online resources at http://www.lotsofcores.com/KNLSDVis that dive deeper, provide downloadable example code, and will let you study how experts in graphics and parallel programming write high-performance code for Knights Landing.

MOTIVATION FOR SOFTWARE-DEFINED VISUALIZATION

It is no secret that high-performance computing data centers continue to generate and ingest an ever-growing amount of data to enable scientists and engineers to gain deeper insight into a wide array of physical and theoretical phenomena. Processing capability and performance continues to increase significantly every few years. This continually increasing performance, usually through added parallelism in the form of cores and vector units, allows researchers to use finer detail, and more data variables, to improve the accuracy of insights. Increased data quantity maps to improved quality (or fidelity) for the analysis, and therefore, new and more complete discoveries. Much of this book is about harnessing that increased processing capability to generate these growing data sets as efficiently as possible through exploiting the incredible parallel processing capabilities of Knights Landing.

One might ask: Once these parallel improvement techniques are applied to codes such as LAMMPS, SeisSol, and QCD (see Chapters 20, 21, and 26, respectively) and they go into production for users, is the job done? The answer is no: Not for the user, scientist, or engineer whose job it is to interpret and analyze the results which can be embedded in many gigabytes, terabytes, or even petabytes of data. One key method to more quickly understand data is using visually-based data analysis, known as visualization. There are a number of both broadly applicable and domain-specific software tools for this purpose including Kitware's ParaView application built upon the Visualization ToolKit (VTK), and the open-source project VisIt for a wide range of domains, as well as tools like Visual Molecular Dynamics (VMD) and VAPOR, for domains such as life sciences and weather analysis. These tools have been written primarily targeting off the shelf graphics cards (GPUs) to process (aka render) and manipulate the data in 2D and 3D.

However, in recent years, many of these tools have begun to face key challenges to keep up with the massive data growth HPC systems enable, making it much more difficult to deliver the level of performance needed to best analyze very large "Big Data" datasets. Several key factors have arisen that conspire to make GPU-based solutions evermore challenging, inefficient, and uneconomical:

- Memory sizes needed to efficiently hold and render data at its highest available quality and resolution often exceed today's typical GPU memory capacity (2 GB up to 24 GB for the highest end devices).
- I/O speeds are not keeping pace with data size growth making it important to minimize data movement to maintain optimal, interactive performance levels including attached devices on the PCIe bus.
- High-fidelity rendering techniques such as ray tracing that can provide enhanced visual cues in complex data models often map better to highly programmable

parallel CPU architectures than today's DirectX and OpenGL traditional rasterization focused GPU architectures (Fig. 17.2).

- Data center power consumption is a growing constraint; discrete GPUs add hundreds of watts per node.
- Today's and future highly parallel many-core Knights Landing processors and multicore Intel® Xeon® Processors provide the fundamental parallelism performance levels needed to efficiently and competitively execute highly parallel graphics workloads.
- Rapid innovation in graphics processing and rendering techniques make the flexibility in highly parallel processors a strong choice allowing new state of the art solutions and performance improvements to a very wide range of HPC data centers from mature systems to those recently deployed.

Motivated by growing concerns from the described challenges, in 2013, Intel and several partners in the Visualization community embarked on an initiative dubbed SDVis. This effort helps maximize the ability to achieve the increased insight promised by evermore capable HPC systems and their associated large data workloads by

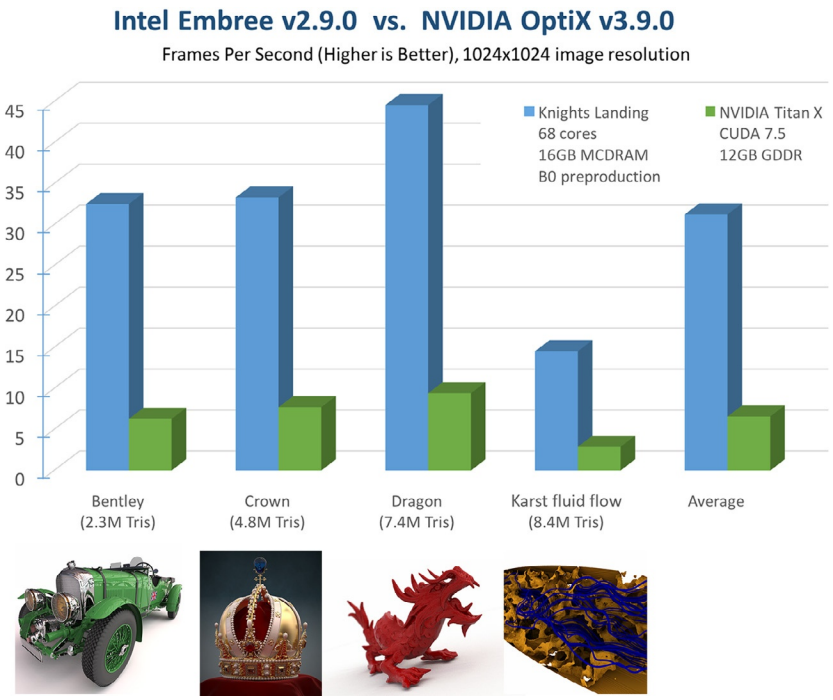


FIG. 17.2

Embree ray tracing kernel library performance on a preproduction Knights Landing 68 core processor (1.3 GHz, using cache and quadrant modes) vs NVidia OptiX ray tracing library performance on NVidia GeForce Titan X GPU in an Intel Xeon Processor E5 v3 2×18 core host platform.

removing roadblocks for large-scale visualization. Furthermore, it is speeding efforts to modernize visualization workflows to match the pace of industry hardware and software innovation.

These visualization specialists also recognized the Knights Landing processor is emerging as an inflection point for the initiative because Knights Landing brings together the following exemplary features that alleviate many challenges making it a very strong platform for SDVis including:

- *Large main memory capacity*: Up to 400 GB for direct, high-fidelity rendering of large data models without data movement.
- *MCDRAM*: High-bandwidth memory with up to 16 GB capacity cache to maximize locality of reference for large data models, image compositing operations, shared data thread scaling, and vector compute.
- *Many cores (up to 72) & threads (up to 288)*: Enables scalable parallel rendering algorithms plus memory latency hiding for ray tracing hierarchical data structure access patterns and graphics primitive processing.
- *AVX-512*: Data parallel processing to support vectorized stages of the OpenSWR OpenGL rasterization pipeline, multiple object ray intersection, coherent ray packet, and efficient ray stream tracing.
- *Efficient power consumption*: High performance per watt platform; Eliminates the need for peripheral PCIe devices for visualization.
- *Intel® Omni-Path fabric*: Low latency/high-bandwidth fabric maximizes very large scale multinode distributed rendering and compositing to enabling scalable resolution, time series animations and scalable interactive frame rates.

The remainder of this chapter will describe the architecture and key technical components of SDVis and how they take advantage of the features of Knights Landing processors. The Intel Xeon Processor platform shares many of Knights Landing's benefits, including large memory, caches, vector instructions, and general programmability, also making them strong platforms for SDVis. As seen in the prior "Pearls" books and throughout Section III of this book, applications and libraries derive benefits when run on any modern parallel processor. SDVis is no different in that respect.

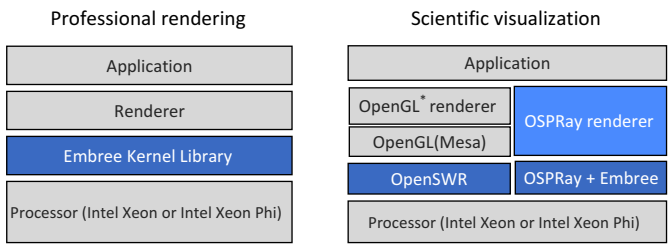


FIG. 17.3

Software architecture for two prominent SDVis use cases: *Left*: Professional rendering (Photorealistic animation, Visual effects, and Industrial design); *Right*: Scientific visualization of large-scale simulated and/or sensor-captured data for visual analysis and scientific insight.

SOFTWARE-DEFINED VISUALIZATION ARCHITECTURE

SDVis components have been designed to support two primary usages: (1) Professional rendering and (2) Scientific visualization. Their placement in the overall rendering solution for each usage is depicted in Fig. 17.3. Professional rendering includes visual special effects and animated movies from studios such as DreamWorks Animation, Pixar, and Walt Disney Studios looking for the highest quality rendered imagery with the desired degree of realism (i.e., up to photorealistic). To achieve that level of quality, professional rendering applications predominantly use ray traced rendering solutions that model the physics and characteristics of light in a given scene for each rendered frame. Professional rendering also encompasses industrial design applications from such companies as AutoDesk and Dassault for automobiles, architecture, advertising, room interiors, and other products for ray traced photorealistic rendering. This enables the designers, executives, and consumers to make decisions on the optimal product design and preferred look for expensive to produce and purchase items. The left side of Fig. 17.3 shows the most prevalent software architecture for professional rendering solutions using the software rendering stack including the Intel developed Embree kernel library. Embree is deployed in well over 50 ISV and in-house rendering solutions, including Pixar's Renderman and DreamWorks Animation Apollo platform. Embree provides an API with low-level functions using highly parallel optimized code (multithreaded and vectorized) for common ray tracing operations.

Embree allows the application and renderer developers to focus on the best means to deliver their unique domain's interface and rendering (3D modeling, textures, materials, shading, color, etc.) requirements calling Embree to provide the performance needed to provide the required image quality with the minimal rendering time.

The right side of Fig. 17.3 shows the SDVis software architecture. Most prevalent SciVis applications such as ParaView, VisIt, VMD, and CEI EnSight have traditionally utilized the standard OpenGL rasterization pipeline for rendering (box marked "OpenGL Renderer"), and for ease of integration, this legacy usage has informed the SciVis rendering software architecture. For software-only OpenGL rendering, the open-source Mesa (www.mesa3d.org) community project has been widely adopted. As will be described in more detail in the upcoming sections, the OpenSWR SDVis component depicted under the OpenGL (Mesa) block is a high-performance rasterizer optimized for scientific visual data that seamlessly fits within the Mesa gallium driver architecture. OpenSWR is a freely available subcomponent upstreamed and available from the Mesa project. Drop-in use of Mesa with OpenSWR without needing to modify applications is enabled through standard OS library and driver loading mechanisms.

The far right side Fig. 17.3 shows the ray tracing components in the SciVis architecture. The "OSPRay Renderer" is a code module that maps 3D object primitives (volumes, triangles, spheres, etc.) and associated elements such as materials, colors, etc. to the OSPRay library interface (box marked "OSPRay+Embree"). OSPRay is a ray tracing rendering infrastructure and API conceptually at a similar software stack level as OpenGL. OSPRay targets straightforward integration into classes of applications that have extensible rendering architectures via plug-ins or are internally

modular enough to allow direct integration of a rendering engine as an additional output path for geometry and volumetric objects. OSPRay utilizes the Embree high-performance ray tracing kernel library. Next, we will describe OpenSWR, Embree, and OSPRay and how they are implemented to best use highly parallel processors such as Knights Landing.

OpenSWR: OpenGL RASTER-GRAPHICS SOFTWARE RENDERING

OpenSWR (Open SoftWare Rasterizer) has been developed as a Mesa (www.mesa3d.org) integrated OpenGL "driver" solution that enables Knights Landing and Intel Xeon processors to be the graphics rendering processor for OpenGL compliant applications. In that context, the method to use OpenSWR is simply to use the standard OpenGL API and use the platform's operating system method for installing and selecting Mesa with OpenSWR as the active OpenGL driver. Fig. 17.4 shows how OpenSWR fits into the Mesa OpenGL architecture.

As of this writing, several Linux-based Supercomputing clusters including TACC's Stampede, Maverick, and LoneStar 5 supercomputers provide OpenSWR-based Mesa packages that are selectable through the job manager. Planned deployments on the Department of Energy Trinity and Cori Supercomputers that utilize Knights Landing processors should occur during 2016. OpenSWR can utilize all the cores and threads available as well as runtime selection of AVX through AVX-512

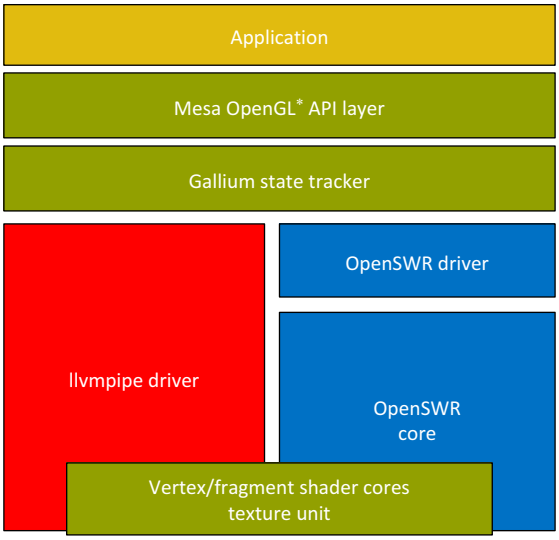


FIG. 17.4

Comparison of the llvmpipe and OpenSWR software architecture, within the Mesa open-source software stack.

vector processing based on the processor architecture. OpenSWR initial development (as of this writing) has been focused on maximizing the OpenGL software rendering performance for the most prevalent HPC visualization applications including ParaView, VTK, VisIt, VMD, and CEI's EnSight. Other applications and broader OpenGL version support are envisioned for the future.

A goal of OpenSWR is to provide significantly higher performance than the "llvmpipe driver" depicted on the left of Fig. 17.4. llvmpipe has been the de facto non-GPU rendering path in Mesa and has been used in prior visualization tool implementations when no GPU solution is available, or the visualization use case requires it. Early results for the OpenSWR implementation are showing performance versus llvmpipe scaling on the magnitude of number of cores ($\sim 30\times$ to $50\times +$ performance (see www.sdvis.org)) for the targeted SciVis applications mentioned. This is generally due to the highly optimized vector and thread implementations provided in OpenSWR, and the llvmpipe limitation of having a single-threaded geometry front-end. OpenSWR has shown strong linear scaling characteristics with increasing cores and benefits from increasing vector width, both of which should translate well to Knights Landing. Please check out http://www.lotsofcores.com/KNLSDVis for the latest performance information.

OpenSWR is a tile-based immediate mode renderer with a sort-free threading model which is arranged as a ring of queues. Fig. 17.5 depicts the control flow. Each entry in the ring represents a draw context that contains all of the draw state and work

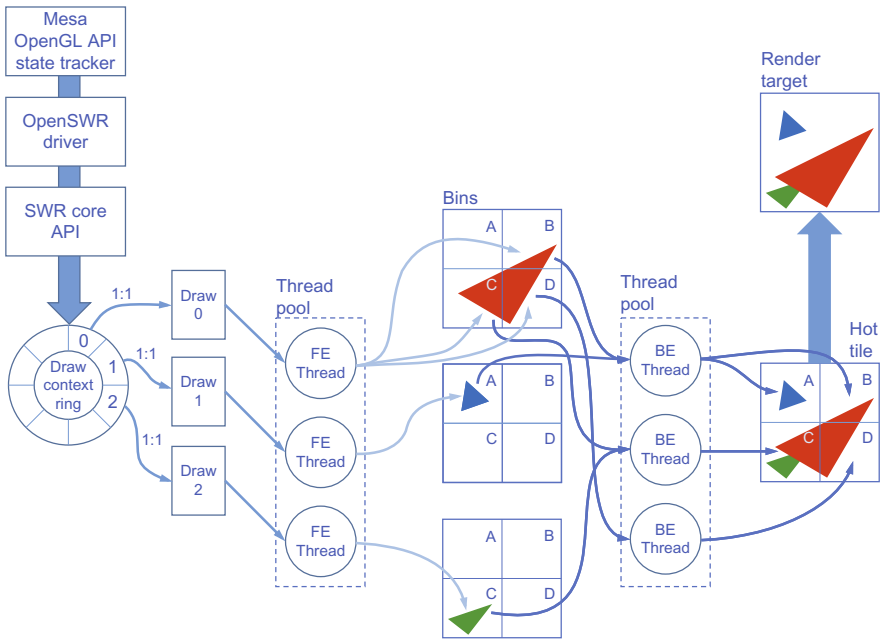


FIG. 17.5

OpenSWR rendering control flow.

queues. An API thread sets up each draw context and worker threads will execute both the frontend (vertex/geometry processing) and backend (fragment) work as required. The ring allows for backend threads to pull work in order. Large draws are split into chunks to allow vertex processing to happen in parallel, with the backend work pickup preserving draw ordering.

The OpenSWR pipeline uses just-in-time compiled code for the fetch shader that does vertex attribute gathering and AOS to SOA conversions, the vertex shader and fragment shaders, streamout, and fragment blending. All of these stages are vectorized, as are the fixed-function units of primitive assembly, transform and clipping, binning, rasterization, and the output merger.

EMBREE: HIGH-PERFORMANCE RAY TRACING KERNEL LIBRARY

Embree is discussed in High-Performance Parallelism Pearls Volume 1, Chapter 21. It is also well described in a SIGGRAPH 2014 paper and a companion tutorial presentation (see *For More Information* at the end of this chapter). Embree is a popular open-source project and includes source and build scripts for Linux, Microsoft Windows, and OS X, plus sample rendering code at http://embree.github.io. We will give a very brief recap of Embree in this section, discuss several features added to Embree since the prior publications, and then focus on early performance for the first ports of Embree to preproduction Knights Landing.

Embree is a collection of high-performance ray tracing kernels akin to Intel's Math Kernel Library (Chapter 13), but for the most prevalent ray tracing algorithms and methods. Graphics software developers use Embree and its API to improve the performance of rendering applications on current and future Intel® Architecture processors, including Knights Landing, with support for SSE, AVX, AVX2, AVX-512, and the first-generation Intel Xeon Phi coprocessor IMCI vector instructions. Embree is developed by experienced graphics and parallel code optimization engineers at Intel. Embree performs automatic runtime code selection to choose the ray traversal and build algorithms that best match the available instruction set.

Embree supports parallel applications written with the Intel® SPMD program compiler (ISPC, https://ispc.github.io) by also providing an ISPC interface to its ray tracing algorithms. This makes it possible to write a renderer application in ISPC that leverages current and future instruction sets without any code change. ISPC also supports runtime code selection using the best code path for your application, while Embree selects the optimal code path for the ray tracing algorithms. The OSPRay rendering infrastructure described in the next section is an ISPC-based renderer that uses the Embree API.

Since prior publications about Embree, it has continued to evolve with optimized support for new types of complex geometric primitives and features to support both optimal memory utilization and rendering performance for the latest photorealistic effects required by movie studios, industrial CAE applications, and special effects. Embree now implements two types of hair/fur rendering-focused primitives based on

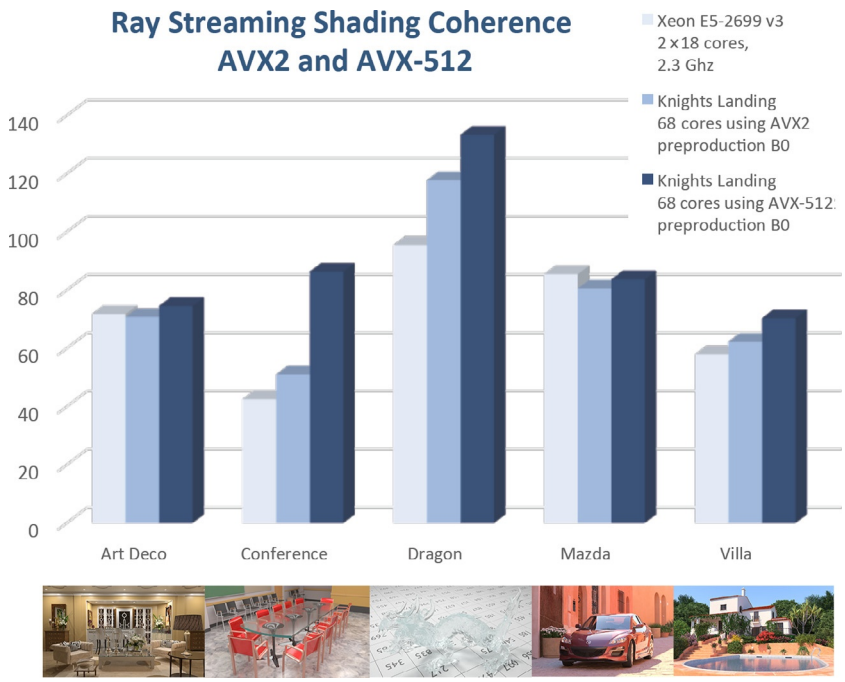


FIG. 17.6

Knights Landing 68 core processor (1.3 GHz, using cache and quadrant modes) AVX-512 vector instructions vs AVX2 on both Intel Xeon E5 v3 2×18 core processor and Knights Landing. Conference scene using compute-bound procedural textures shows almost $2\times$ performance improvement as expected. Other scenes have memory-bound texture fetch processing but still show benefits on Knights Landing AVX-512.

(1) Bezier curves and (2) connected line segment definitions. Embree also has added robust Catmull-Clark subdivision surface support. Both the hair rendering and subdivision surface solutions had detailed design and usage descriptions in published papers (see *For More Information*). Most recently ray streaming support that also enables highly efficient coherent ray shading solutions through algorithms that can take groups of rays, extract *coherence* (rays close in proximity that mostly intersect the same scene objects) to maximize vector utilization that is particularly beneficial on Knights Landing with its wide AVX-512 vector capability. Fig. 17.6 shows the benefits of Knights Landing using AVX2 and AVX-512 vs a 2×18 core Intel Xeon E5 v3 dual socket processor running this new shading coherence code. Furthermore, it shows the benefits of AVX-512 as well. As is shown, AVX-512 on Knights Landing provides the maximum performance on all the workloads. Four of the five workloads use memory-based texture fetches making its processing more memory bound limiting the full AVX-512 benefit. The "Conference" scene uses a compute-bound procedural texture method that shows the close to theoretical $2\times$ full benefit of AVX-512.

Optimizing for Knights Landing and AVX-512 is relatively new as of publication; for the latest information, please use the links in *For More Information*.

OSPRay: SCALABLE RAY TRACING FRAMEWORK

OSPRay (Open, Scalable, Portable Ray tracing) is a rendering framework which uses ray tracing as the underlying technique to render images. It is currently primarily targeted at enabling high-performance and improved image fidelity for SciVis applications through the 3D visibility data structures (hierarchical object tree) mechanisms and lighting/shading benefits inherent in ray tracing applications; since based on the Embree kernel library, OSPRay also can provide up to photorealistic rendering capability (not typically needed for SciVis applications). A key capability built into OSPRay is the ability to *scale* its rendering capability from a single node or workstation to a full supercomputing cluster with thousands of nodes including a mix of Knights Landing and Intel Xeon processors and without requiring any GPUs to perform rendering. This scaling can be done without the master application's (e.g., ParaView) need to itself run on multiple nodes. The end display mechanism can be a workstation, laptop, or Display Wall. The OSPRay API and infrastructure (based on MPI) can optimally manage the rendering tasks. Since OSPRay provides a new API requiring integration into HPC applications and run-time environments, we will do a deeper dive into OSPRay's architecture and its use of Knights Landing's vector, thread, and cluster (Intel Omni-Path fabric, Chapter 5) level parallelism in the following sections.

OSPRay API OVERVIEW

Renderers contain several components which are coordinated to carry out ray generation, ray intersection, and shading operations. To implement these operations, OSPRay categorizes "actors" as depicted in Fig. 17.7. Each actor type shares a common role in rendering while having different implementations. OSPRay expresses the relationships between actors using an object-oriented programming model—each "role" is an abstract base type which is extended to accomplish specific goals. Client applications to OSPRay only track handles to the generic base types through the use of factory functions. This keeps applications from needing to understand the implementation details of specific objects—only the role of the object needs to be known.

For the purposes of this chapter, we will describe some of the key actors in OSPRay and not comprehensively discuss all of them. The object types readers should be aware of are as follows:

- *Frame buffers*. The result of a rendering pass in OSPRay is an image, which a frame buffer stores. This is typically a 2D array of pixel colors, which can be stored in a variety of pixel data formats. While frame buffers are generally not a

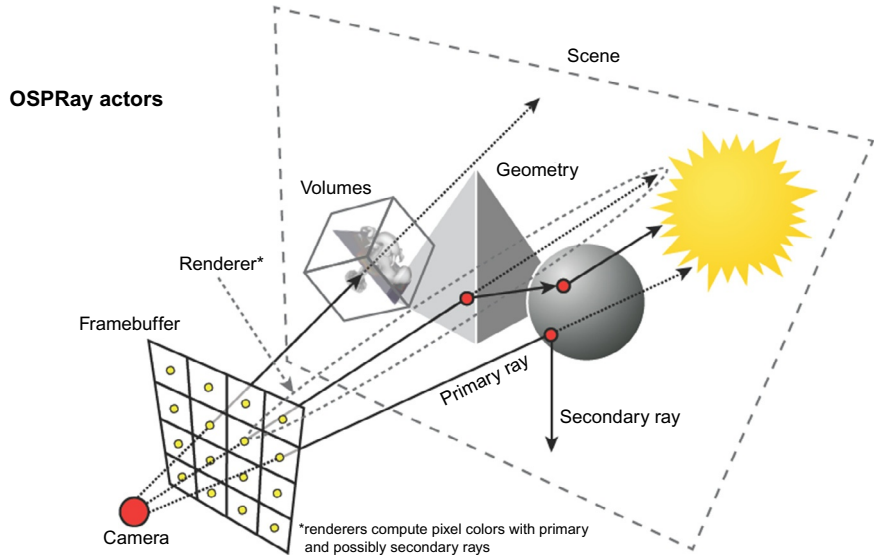


FIG. 17.7

OSPRay "Actors" in its conceptual object-oriented API and model.

complex object, they are critical to the coordination of rendering passes, such as pixel color accumulation, sub-pixel sampling, and tile compositing.

- *Cameras*. Every image rendered occurs from a particular location in the 3D scene. Cameras take location and directional information to generate primary rays which map to a particular pixel in the frame buffer. The most common example of a camera is a perspective camera, where primary rays share a common origin and are traced through individual pixels in the image plane and handed to the renderer to intersect and shade.
- *Geometry*. Cameras are aimed at virtual objects in a scene, which are comprised of geometric shapes which form visible objects. Geometries represent surfaces with which rays can be intersected. While triangles are a very common geometric primitive, any mathematical ray-primitive intersection can create a geometric surface: spheres, cones, planes, cylinders, b-splines, etc.
- *Volumes*. Similar to geometry, volumes are data (typically a 3D grid) which rays interact with for renderers to shade. However, volumes use sampling instead of intersection for renderers to shade. For example, a ray passing through a volume would sample the volume's data at various intervals along the ray. Then, a transfer function would be applied to the samples to map values found in the volume to colors a renderer can incorporate in shading operations.
- *Models*. A model represents a collection point for one or more geometry and volume objects to represent a scene. Models are the natural top-level object to build acceleration structures for accelerating ray intersections.

- *Renderers*. All shading operations are coordinated by renderers—an input primary ray from the camera is used to interact with the scene, resulting in a pixel color stored in the frame buffer. Renderers intersect the primary ray with the scene, which may result in more rays being generated to compute visibility, transparency, lighting, and volumetric contributions to the final pixel color. Renderers vary greatly in complexity, but generally have ways of trading image quality for performance—for example, using progressive refinement with less samples per frame time to increase per-frame rendering performance.

In addition to defining actors, the OSPRay API relies upon an abstract device abstraction to map the API to different hardware configurations. As stated before, objects are created through the use of factory functions, allowing the API to easily adapt to many types of execution environments. During initialization, OSPRay takes a configuration hint as to the backend the application prefers, setting up the appropriate device implementation for OSPRay API calls. Our current implementation has three devices available to applications:

- *LocalDevice*. The default device OSPRay uses is the local device, which uses the same CPU resources the calling application is using. All API calls and data share memory on the local machine, which could be a laptop, workstation, or single compute node where the application is distributed.
- *COIDevice*. Intel Xeon Phi coprocessors (see Chapter 18) are supported through the coprocessor offload infrastructure (COI) library which enables offload of computation to coprocessors on a local machine. OSPRay is able to use COI to offload OSPRay rendering computations to any available coprocessors, freeing up the host CPU for the calling application.
- *MPIDevice*. Finally, OSPRay can take advantage of HPC multinode Knights Landing or other processor nodes through the use of MPI and high-speed networking such as Intel Omni-Path fabric. Similar to the COI device, the MPI device offloads rendering operations to any available compute nodes provided in the MPI launch, where rank 0 collects final frame buffer information for the calling application. It is important to note that while the OSPRay API backend runs distributed, the coordination of rendering on multiple compute nodes is transparent to the calling application—the use of the API is identical to other devices and the application itself only needs to run on a single machine.

Client applications which use OSPRay rely upon properties and commit semantics to correctly construct, configure, and connect related objects together to render images. Applications track handles which they receive from OSPRay factory functions, where specific properties and interfaces are not known at compile time. In order to set parameters and data for these objects, the API allows clients to set abstract named properties on an object instance. Internally, every OSPRay object contains a list of parameters clients have set, regardless of device type, which are then utilized appropriately when an object is committed—making the object and its parameters available for use by the API.

For example, to set the number of ambient occlusion (AO) rays an instance of an AO renderer will use per frame would be set with the following function call:

```
ospSet1i(aoRendererInstance, "aoSamplesPerFrame", 4);
```

Internally, the object would store the named-value pair of {"aoSamplesPerFrame", 4}. Then to make the parameter visible and used by the object, clients would commit the renderer with the following call:

```
ospCommit(aoRendererInstance);
```

The commit semantics OSPRay expresses have different meanings for different object types and device implementations, enabling applications to control when heavy weight computations should occur. For example, when a model is committed, an acceleration structure build is triggered which could take long enough for an application to need control when it occurs. Additionally, committing an object makes the object visible and valid within the API—for offload devices, data may need to be broadcast to all hardware used by OSPRay, such as multiple coprocessors or compute nodes. Our implementation assumes that all objects are individually committed. In other words, no commit dependencies are handled automatically, which greatly simplifies the implementation and expectations of how the OSPRay API is used.

FORMULATING A HIGH-PERFORMANCE IMPLEMENTATION

High-performance code on Knights Landing systems centers around expressing parallelism. Specifically, data parallelism in throughput sensitive calculations is critical to getting performance out of modern CPUs. There are three levels of parallelism we examine on Knights Landing for OSPRay: vectorization, multithreading, and node-parallelism. We will discuss each type of parallelism in order from lowest to highest levels of parallelism.

Vector parallelism

The lowest level of parallelism we can exploit is vector parallelism through the use of the new AVX-512 vector instructions found on Knights Landing processors. OSPRay typically works with single-precision data allowing 16 simultaneous operations; if not vectorized you will not approach the available performance provided!

OSPRay utilizes the Intel SPMD Compiler (ISPC) to vectorize various parts of the code, while leveraging Embree (which also supports ISPC) to provide well vectorized ray intersection kernels. OSPRay uses ISPC to vectorize all shading operations—while rendering a particular frame, all rays coming from a camera execute the same shading algorithms. For example, a simple AO renderer always finds the first hit point along the primary ray and, should a geometric surface be found, traces secondary rays to determine the amount of light hitting that particular surface.

However, because different rays may take different paths through shading logic, ISPC's Single Program, Multiple Data (SPMD) programming model is a great way to encode these operations. We will give a brief overview of the programming model and ISPC usage.

```
varying float fcn(varying float v1, varying float v2) {
  if (v1 > 10.f) {
     return v1 + v2;// vector lanes which are > 10.f do this
  } else {
     return v1 * v2;// vector lanes which are < 10.f do this
  }
}
```

FIG. 17.8

Example showing SPMD programming in ISPC. Note that the if-statement represents a vector's worth of comparisons. Each body of the if-statement is executed, where the compiler masks out lanes for which the body does not apply.

SPMD programming is not a new concept but is less commonly used on CPUs as other programming models targeted at vector instructions, likely for historical reasons. The concept of SPMD programming is fairly straightforward—scalar operations are replicated across SIMD vector lanes, where divergent control-flow is handled by vector masks, transparently by the compiler. For example, given an AVX-512 16-wide vector of floats, if the first 10 lanes evaluate "true" in an if-statement, then the remaining 6 lanes are "disabled" by vector masks and the "true" portion of the code is executed. Then, should there be chained "else if" or "else" statements, those code paths are executed with the other lanes which apply to them. Fig. 17.8 shows an example of SPMD control flow code.

This execution model is expressed with two keywords: uniform and varying. Uniform variables represent nonvector values which are the same across all vector lanes, while varying variables represent a vector of values. This allows for programmers to help the compiler by using uniform variables where possible to maximize vector utilization (number of active lanes) by minimizing masking control flow. However, where such divergent code paths can occur, programmers do not need to write code which looks any different, making it easier to write clear and understandable code. Additionally, varying variables are not tied to any particular vector width, making it easy to implement code once and compile it for a number of different vector instruction sets, including AVX-512—making code maintenance much easier.

OSPRay's objects which need to be vectorized by ISPC are implanted as sibling objects to their C++ class. Thus, we use the concept of an ISPC equivalent to create and manage data to and from ISPC. While ISPC and C++ can interchange pointers because they share the same memory address space, we use the aforementioned commit semantics to store object parameters in ISPC data structures for better rendering performance. The named-parameter lookup system works well as an application interface OSPRay objects, but it is far too slow to always look up parameters whenever they are needed. For example, during the commit process, the number of AO rays to trace per-frame is stored in an ISPC AO renderer object for easy access in the ISPC rendering logic.

Thread parallelism

The next level of parallelism we can exploit is *thread parallelism*. For instance, a single-threaded application running on a 72-core Knights Landing processor only uses 1/72nd of the processors compute power, even if the application is completely vectorized.

There are a number of great options available to implement multithreaded applications for Knights Landing which are described in detail in other chapters in Section II, including OpenMP and TBB. Additionally, the new C++11 standard now provides platform agnostic primitives in the standard library to directly access individual threads, making it easier to implement portable tasking systems in C++. Regardless of the threading model, OSPRay maximizes the vector parallelism across the processor cores when rendering frames by scheduling work across the multiple threads available.

A frame is a collection of pixels, each of which can be computed completely independent from each other. This characteristic makes computing the final pixel colors, *embarrassingly parallel*. This gives us a lot of freedom to schedule pixel shading computations across all threads a processor has available. We use the freedom we have in scheduling work for each frame to better map the computation to the way processors like data. OSPRay renders a frame by first dividing up the frame into 2D tiles, which is then further subdivided into individual pixel operations. The higher-level tiles are subdivided as tasks, with each tile spawning smaller subtasks. This multilevel expression of parallelism is known as *nested parallelism* (see also Chapter 8).

Nested parallelism helps a multithreaded tasking system understand what computations are the most similar and thus executes them in parallel. Pixels in a frame which are close together are the most likely pixels to trace rays which encounter the same geometric primitives, that is they are *spatially coherent*. As rays are traced through a scene, rays which are spatially coherent will load the same data from the ray tracing acceleration structure, taking advantage of data locality in caches. Therefore, executing lower-level tasks at the same time maximizes the likelihood that the right data will be in cache for those tasks. As noted in Chapter 7, better data locality improves cache utilization leading to better performance. Fig. 17.9 shows the performance of OSPRay using its vector and thread-level parallelism on both preproduction Knights Landing and Intel Xeon E5 v3 processors. The achieved frame rates provide the level of interactivity needed by scientists to gain insight into complex and time-based datasets.

Node (cluster level) parallelism

The highest level of parallelism we can exploit is *node parallelism*. To continue to scale up to achieve higher performance, multiple networked Knights Landings and/or Intel Xeon Processors execute different portions of the same problem.

HPC, particularly Supercomputing design, is a difficult and diverse field, with many different ways network and compute resources are constructed to work together. OSPRay's method of using multiple nodes of a supercomputer is by distributing tile jobs across nodes to be executed in parallel, collected by a "master" node to

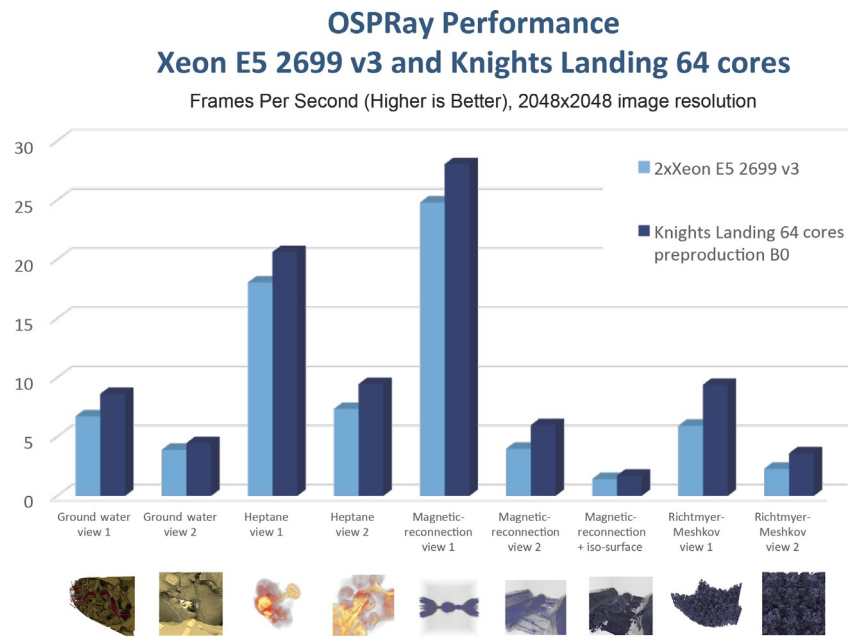


FIG. 17.9

OSPRay rendering performance on preproduction Knights Landing 64 core processor (1.3 GHz, using cache and quadrant modes) and Intel Xeon E5 v3 processor system with 2×18 cores. Each system provides SciVis level interactive frame rates >10 frames per second.

be stitched together (composited), and then returned to the application. In this case, the entire scene is replicated on each compute node, making it easy to assign tile rendering tasks in a round-robin fashion to all compute nodes.

The cost of network communication in these environments determines the peak number of nodes which an application can benefit from. Network capability and per-tile rendering cost contribute to the ability for scalable rendering. For example, a very inexpensive AO rendering of a small dataset would not be enough work to overcome even the fastest supercomputer networks. On the contrary, a photorealistic path tracer with a complex and large scene would give each node enough work to recoup tile communication overhead.

Due to the distributed nature of supercomputing, more aggregate memory can be made available for OSPRay data. In the previous example, a replicated dataset across all compute nodes limits the rendered scene size to the amount of memory found in the smallest compute node. To overcome this limitation, OSPRay can also distribute sub-portions of the scene dataset on to different nodes, but not without cost.

Scaling across memory requires much more communication between compute nodes. For example, rendering a distributed volume with only primary rays requires compute nodes to coordinate which tiles are rendered on which nodes. When volume rendering, in the general case, volumes are sampled at various points along a ray, where a transfer function is applied to get a color and opacity to be blended into the final pixel color. An example method of handling this in a distributed environment is to use tile communication. If a node is rendering a particular tile, it likely does not have all of the data that rays in that tile will encounter. This means the node must send that tile to other nodes for them to render their portion of the tile and send it back to the original node.

Thus the scaling performance varies greatly on many factors—data size, data complexity, rendering complexity, average node compute power, network bandwidth, network latency, and frame size. Given that networks can be the slowest memory I/O mechanism—even on the fastest supercomputers—we expect, and typically see, a trade-off in performance. However, data replicated distributed rendering typically scales well with compute-heavy rendering, while data distributed rendering scales well with scene size. Recent testing with Intel Omni-Path fabric has shown it to provide a strong mix of low latency and high bandwidth supporting both distributed tile (latency, message rate sensitive) and replicated data rendering (higher-aggregate bandwidth).

So, we can see that the right formulation for parallel optimization allows OSPRay to provide strong scaling on modern processors such as Knights Landing and Intel Xeon processors. By maximizing parallelism on OSPRay, as with any parallel code, through leveraging SIMD vector instructions with data locality, multithreading, and multinode work distribution, a well-performing rendering solution can be delivered at any scale.

SUMMARY

SDVis of large data sets is best done on multi- and many-core processors; it is no surprise that Knights Landing does very well for SDVis. Three key open-source libraries that are fundamental for SDVis work (i.e., OpenSWR, Embree, and OSPRay) can be used *now* to benefit from the SDVis capabilities of Knights Landing and multicore processors.

SDVis provides scientists, engineers, and anyone who needs to better understand their data visually to view and interact with the ever-growing "Big Data" processed on HPC systems, including the largest Supercomputers, with improved fidelity and performance. Knights Landing processors with large memory capacity, MCDRAM high-bandwidth memory, high core and thread count, AVX-512 vector processing, plus Intel Omni-Path fabric (Chapter 5) provide a platform that solves many roadblocks to traditional visualization systems like I/O bottlenecks and limited graphics device memory. SDVis is architected to scale to maximize the combination of

interactive frame rates, data detail level, and advanced lighting options that improve the visual cues that drive insight.

The SDVis initiative has three key foundation software rendering libraries: OpenSWR, Embree, and OSPRay. The visualization community is embracing these libraries and has integrated them into prevalent visualization tools like ParaView, VTK, VisIt, VMD, and EnSight, with more expected over time. Because SDVis is a rapidly evolving initiative and as of the publication deadline the tools and libraries have just begun to fully take advantage of Knights Landing's capabilities, we will augment this chapter with example code, updated Knights Landing performance information, unique use case studies, and more at http://www.lotsofcores.com/KNLSDVis. You can also get more general background information on SDVis at the website http://www.sdvis.org. Finally, links to downloadable versions of prominent Visualization tools including ParaView, VTK, VisIt, VMD that include SDVis architecture capabilities are listed next in *For More Information*.

IMAGE ATTRIBUTIONS

- *Conference scene*. Model courtesy of Anat Grynberg and Greg Ward.
- *Crown*. Model by Martin Lubich and provided under the Creative Commons Attribution License http://creativecommons.org.
- *CSAFE Heptane Gas Dataset*. Model courtesy of the Center for the Simulation of Accidental Fires and Explosions (CSAFE) at the Scientific Computing and Imaging Institute (SCI), University of Utah. http://dx.doi.org/10.1103/PhysRevLett.113.155005.
- *Karst Ground Water Simulation Dataset*. Model courtesy Texas Advanced Computing Center (TACC) and Florida International University.
- *Magnetic Reconnection*. Model courtesy Bill Daughton (LANL) and Berk Geveci (KitWare). Also, see Guo et al. 2014, Physical Review Letters, 113, 155005.
- *Mazda, Art Deco, and Villa*. Models copyright © Evermotion used under Evermotion Commercial License agreement.
- *Richtmyer-Meshkov Iso-Surface*. Model courtesy Lawrence-Livermore National Labs (LLNL).
- *XYZ-Dragon*. Model courtesy of Stanford Computer Graphics Laboratory.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this chapter.

- For the latest updates on software-defined visualization efforts and optimization techniques, sample code, case studies and performance on Knights Landing: http://lotsofcores.com/KNLSDVis.

- For background information on the general software-defined visualization initiative, latest news, presentations, component project links, and more: http://www.sdvis.org.
- For detailed information, source code and examples for the Embree Kernel Library see http://embree.github.io.
- For a tutorial presented at Siggraph 2014 on Embree: https://software.intel.com/sites/default/files/managed/1f/f0/siggraph2014_embree_tutorial_012.pdf.
- A detailed description of Embree from Siggraph 2014 Wald et al. paper: http://dl.acm.org/citation.cfm?id=2601199.
- Description of Embree Hair/Fur implementation from HPG 2014 Woop et al. paper: https://embree.github.io/papers/2014-HPG-hair.pdf.
- Description of Embree Catmull-Clark subdivision surface implementation from HPG 2015 Benthin et al paper: https://embree.github.io/papers/2015-HPG-tcache.pdf.
- For detailed information, source code and examples for the OSPRay Ray tracing Infrastructure Library: http://www.ospray.org.
- For OpenSWR source code and information on the OpenGL open-source project Mesa: http://www.mesa3d.org and http://www.openswr.org.
- For Kitware's ParaView binary and source downloads: http://www.paraview.org.
- To learn more about the VisIt visualization application and download source and binary versions including SDVis integration code: https://visit.llnl.gov.
- Get information and download visual molecular dynamics (VMD): http://www.ks.uiuc.edu/Research/vmd/.
- To learn more about VAPOR weather visualization tools: https://www.vapor.ucar.edu.
- To learn more and to license CEI's EnSight CAE visualization tool with OpenSWR integration: http://www.ensight.com.
- Intel SPMD program compiler, an open-source compiler for high-performance SIMD programming on the CPU, https://ispc.github.io/.

CHAPTER

18 Offload to Knights Landing

This chapter covers two topics: the offload programming model and Knights Landing coprocessor specific considerations. They are separate, but related, topics.

What is new with Knights Landing in this chapter?

Using OpenMP target directives, the offload model can be used to offload to a Knights Landing coprocessor over PCIe, or offload to a Knights Landing processor over fabric.

OFFLOAD PROGRAMMING MODEL—USING WITH KNIGHTS LANDING

Programming in an "offload" style simply means writing a program from a perspective of running code on a host by default, and selectively marking code for execution on an offload target. Offload programming also includes a way to describe data that needs to be accessed by the target and control the possible movement of that data to and from the target.

Programs written using an offload programming model, including all programs written for Knights Corner using an offload model, have three options when being recompiled for Knights Landing:

1. *Run natively* on Knights Landing with no offloading. The compiler can be directed to ignore the offload directives and simply run the entire program, unmodified, on the processor natively. This option is available on any processor, not just Knights Landing.
2. *Offload to Knights Landing coprocessors*. In the same fashion as offload worked with a Knights Corner coprocessor, it can work with a Knights Landing coprocessor.
3. *Offload over fabric (OoF) to a Knights Landing processor* (using the same binary as for offload to a coprocessor—no changes!). In clusters containing Knights Landing nodes, a program can be compiled to run on an Intel Xeon processor and offload over the fabric to a Knights Landing processor. This allows any Knights Landing processor to be the target of an offload from another

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00018-1

processor that acts as the host for the program. This is an innovative and new model that continues to offer the offload programming model as a way to accelerate parallel regions. Application code does not require any changes to use OoF versus offload to a coprocessor over PCIe.

PROCESSORS VERSUS COPROCESSOR

Knights Landing is a processor which can also be purchased as a coprocessor. This is distinct from the First Generation Intel Xeon Phi product, Knights Corner, which was available only as a coprocessor. Knights Corner was, in essence, a processor trapped in a coprocessor body. While this provided an excellent first product and even helped power the world's faster computer for years, the limitations of the coprocessor implementation had many people looking forward to an Intel® Xeon Phi™ *processor*. A processor implementation is freed from restrictions of limited memory and PCIe transfers back and forth with a host processor.

The second-generation Intel Xeon Phi product, Knights Landing, is available as either a processor or a coprocessor. Knights Landing can be thought of as a processor that can also be packaged as a coprocessor.

All Intel Xeon Phi products have a many-core design that is fundamentally the same to program. It is a cache-coherent SMP-on-a-chip device whether it is designed into a system as a processor or a coprocessor. The difference is that a coprocessor requires a host processor(s) which feeds it code to execute and data to process. A coprocessor is most often used as an attached device in which we think of our program as running on a host and occasionally targeting (also called offloading) some processing to the coprocessor. Code written in this style can embed the offloaded routine and use OpenMP target directives to achieve the offloading. A coprocessor can also be used natively. When we think of coding using the native method, we write our application to run on a collection of cores which in turn participate in ranks of an MPI application. Programs written in this style do not need to care that some ranks may sit on host processor cores and some on coprocessor cores other than needing an approach to dealing with load balancing to get the best performance.

COPROCESSOR SPECIFIC DIFFERENCES

The Knights Landing on a coprocessor card is actually a Knights Landing processor. It is "magic" on the card itself that allow it to behave as a coprocessor card.

The biggest difference with a Knights Landing coprocessor is all memory is MCDRAM (no DRAM on the card). It logically follows that a Knights Landing coprocessor will always use the memory mode called flat (see Chapter 3).

Otherwise, all of the information in this book applies to Knights Landing coprocessor programming as well as Knights Landing processor programming.

OFFLOAD MODEL CONSIDERATIONS

With the exception of Math Kernel Library (MKL) Automatic Offload support, tasking models need to start on the target of the offload. Recall that offload targets can be a Knights Landing coprocessor over PCIe, or a Knights Landing processor over fabric. Therefore, OpenMP directives, the Fortran DO CONCURRENT, the TBB algorithms, or the thread creates in pthreads, need to actually be run on the target of an offload either natively or inside an offload region of the code. The notable exceptions are calls to MKL, which can automatically create offloads from within the library as previously mentioned.

Like OpenMP, use of TBB as a threading model must exist only in a single memory space. Any TBB use on a given coprocessor will only influence tasks and threading on the same coprocessor, and any use on the host (processor[s] with shared memory between them) will apply only to the host processor(s). An offload directive can invoke TBB, and such actions will impact only where the offload is executed. Since offload regions can potentially be run on the host (if no coprocessor is available), this can be a little confusing, or very convenient when we have the right mindset.

As the TBB flow graph model evolves, this could conceivably change to support graphs that span host and target. This is beyond the scope of our short chapter, and it is worth checking for updates with the open source TBB project. Likewise, hStreams can conceivably support spanning host and target. See *For More Information* at the end of this chapter for more insights.

OFFLOADING VIA MKL

The Intel® MKL has automatic offloading capabilities. These are discussed in Chapter 13 with the rest of the discussion of Intel MKL rather than here.

The MKL automatic offloading capabilities are essentially an offload model built into MKL. A program runs on a host, but calls to MKL may offload to a target device (i.e., a coprocessor over PCIe, or a processor over fabric).

OFFLOADING VIA OPENMP

When Knights Corner was introduced in 2012, OpenMP had not yet standardized a method for supporting targets, although a technical report proposing such support was in draft form. The standardization process had to consider many factors, including supporting a broad range of devices and defining interactions with the rest of OpenMP (parallelism and tasking directives).

Several years before Knights Corner was introduced, Intel defined and supported *Language Extensions for Offload* (LEO) which acted much like extensions to OpenMP. The key LEO pragma was "#pragma offload." Intel has supported LEO targets of Knights Corner and Intel GPUs. At the same time, NVidia introduced OpenACC which supported directives for NVidia GPUs. Neither LEO nor

OpenACC have seen deployment, nor implementations with high performance, for targets other than the original targets envisioned and supported by their creators.

OpenMP 4.0, and updates since then, brought together Intel, NVidia and many other companies and users to define and support a common standard. To support legacy customers, both Intel and NVidia continue to support LEO and OpenACC for their legacy users.

OpenMP 4.x implementations have appeared in multiple compilers, for multiple targets including Intel processors, Intel GPUs, AMD GPUs, and NVidia GPUs. Compiler support exists in Intel compilers, gcc and clang/LLVM. Our list here of targets and compilers is incomplete. In short, OpenMP 4.x offers the standard for offloading that new coding should use.

Therefore, we limit our chapter to briefly discussing OpenMP target. We leave a lot of details on using this standard to the many tutorials and documentation that have appeared in support of the standard.

OPENMP TARGET DIRECTIVES

Target directives come in two primary forms. One addresses moving code to a target with optional data mapping clauses. The other is specific to establish a "data environment" on the target. Effective offload will generally seek to move data to and from a target while the target is busy doing work. In order to support that, the concept of a "data environment" is essential to manage the data movement instead of always tying it to the specification of the computational work to be offloaded.

OFFLOAD: OMP TARGET

Fig. 18.1 shows a simple example which offloads a vector multiply to a target device. The target directive specifies that variables *num*, *a*, and *b* need to be copied to the target, and variable *c* is copied back when the target region completes. Note that the variable "*num*" is not mentioned in any target directive. This is because simple variables can be used within a target directive, and the compiler will automatically

```
void MyVecAdd(int num, float *a, float *b, float *c)
{
  #pragma omp target \
       map(to: a[0:num], b[0:num]) \
       map(from: c[0:num])
  {
    int i;
    #pragma omp parallel for
    for (i = 0; i < num; i++)
      c[i] = a[i] + b[i];
  }
}
```

FIG. 18.1

Simple example to offload a vector multiply.

decide to make them copied between the host and the target device. The target region may be offloaded to a target device, or it may be run on the host. Normally, if a supported target is available at runtime, we would expect the target region will be offloaded. OpenMP specifies an environment variable to control what device is offloaded to if none is specified in the directive (OMP_DEFAULT_DEVICE). There is a corresponding runtime call that can be made omp_set_default_device. That function and related device functions are listed in Fig 18.5. Further information on each function can be found in the OpenMP specification online, see *For More Information* at the end of the chapter.

Each target device has its own threads that are distinct from threads that execute on another device. Threads cannot migrate from one device to another device. The execution model is host-centric such that the host device offloads target regions to target devices; target devices cannot offload to other devices.

When a target directive is encountered, the target region is executed by the implicit device task. The task that encounters the target pragma waits at the end of the region covered by the directive until execution of the region completes. If a target device does not exist, is not supported, or cannot execute the target region, then the target region is executed by the host device. Figures 18.2–18.5 illustrate key aspects of OpenMP target directives.

DATA ENVIRONMENTS: OMP TARGET DATA

Data is "mapped" to the target. Any variable that is mapped must be bitwise copyable (the OpenMP specification defines "mappable type" in detail). That does not have to imply a copy operation if the system supports other methods of sharing data with a target. Hence, the clause and directives are called "map" not "copy." A target data directive creates a data environment. Within that environment, a target update directive causes the specified data to be refreshed. Fig. 18.6 shows an example using these two directives. Together, these help us optimize our data transfers by avoiding frequent transfers and potentially overlap communication with our computations.

```
!$omp target [clause[[,]clause],…] new-line
     structured block
!$omp end target
```

FIG. 18.2

Syntax of a target directive in Fortran.

```
#pragma omp target [clause,[[,]clause],…] new-line
     structured block
```

FIG. 18.3

Syntax of a target directive in C.

```
Directives to execute code on a target device:
omp target [clause[[,]clause],…]
  structured-block
omp declare target [function-definitions-or-
declarations]

Map variables to a target device:
map ([map-type:] list) // used as a clause
  map-type := alloc | tofrom | to | from
omp target data [clause[[,]clause],…]
  structured-block
omp target update [clause,[[,]clause],…]
omp declare target [variable-definitions-or-
declarations]

Workshare for target device
omp teams [clause[[,]clause],…]
  structured-block
omp distribute [clause[[,]clause],…]
  for-loops
```

FIG. 18.4

OpenMP offload directives.

```
void omp_set_default_device(int devicenum )
int omp_get_default_device(void)
int omp_get_num_devices(void);
int omp_get_num_teams(void)
int omp_get_team_num(void);
int omp_is_initial_device(void);
```

FIG. 18.5

Runtime support functions for managing target devices.

Variables can be declared globally to be shared for the entire program. Such variables are always available to be used in target update directives. Fig. 18.7 shows such global definition. Functions and subroutines can be included in the same construct to ensure they are available on the target for calling within code running on target. These declarations ask the compiler to generate code for the host and target for such functions and subroutines.

CONCURRENT HOST AND TARGET EXECUTION

OpenMP tasks can easily be used to specify that work on a host and a target run concurrently, as shown in Fig. 18.8. The same concept can support concurrency among multiple targets. That is especially useful for hosts with multiple coprocessors attached.

```
extern void loadone(float*, float*, int);
extern void loadtwo(float*, float*, int);
extern void spitout(float*, int);
void vec_mult(float *prod,
              float *vec1,
              float *vec2,
              int n) {
  int i;

  loadone(vec1, vec2, n); // may change vec1, vec2

// plan to map/copy prod at the "target data" end
// this way we map prod only once
#pragma omp target data map(from: prod[0:n])
  {
// map/copy vec1, vec2 to target the first time…
#pragma omp target map(to: vec1[:n], vec2[:n])
#pragma omp parallel for
    for (i=0; i<n; i++)
      prod[i] = vec1[i] * vec2[i];

    loadtwo(vec1, vec2); // may change vec1, vec2
// map/copy vec1, vec2 to target again…
#pragma omp target map(to: vec1[:n], vec2[:n])
#pragma omp parallel for
    for (i=0; i<n; i++)
      prod[i] = prod[i] + (vec1[i] * vec2[i]);
  }
  // map/copy prod back here… the "target data" end!
  spitout(prod, n);
}
```

FIG. 18.6

Usage example of target data directive.

```
#define N 1024
#pragma omp declare target
float prod[N], vec1[N], vec2[N];
#pragma omp end declare target

extern void loadone(float*, float*, int);
extern void loadtwo(float*, float*, int);
extern void spitout(float*, int);
void vec_mult() {
  int i;

  loadone(vec1, vec2, N); // may change vec1, vec2

// map/copy vec1, vec2, N to target the first time…
#pragma omp target map(to: vec1[:N], vec2[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      prod[i] = vec1[i] * vec2[i];

    loadtwo(vec1, vec2, N); // may change vec1, vec2
// map/copy vec1, vec2 to target again…
#pragma omp target map(to: vec1[:N], vec2[:N])
#pragma omp parallel for
    for (i=0; i<N; i++)
      prod[i] = prod[i] + (vec1[i] * vec2[i]);

  // we need to explicitly map/copy prod back here
#pragma omp target update from(prod)
  spitout(prod, N);
}
```

FIG. 18.7

Globally declare as target accessible.

```
#pragma omp parallel sections
{
  #pragma omp task
  {
    #pragma omp target…
    // code to offload
  }
  #pragma omp task
  {
    // code to run on host
  }
}
```

FIG. 18.8

Concurrent host and target execution.

OFFLOAD OVER FABRIC

The OpenMP offload model does not specify any restrictions on what a target device is, nor where it is. Knights Landing coprocessors are supported as target devices. But, so are Knights Landing processors!

The ability to offload to a Knights Landing processor from an Intel Xeon processor can be done with no changes to the application at all. The "magic" to do this offload is completely contained within the Intel Coprocessor Offload Infrastructure (COI). COI is open source software started by Intel to originally support Knights Corner coprocessors. It has been extended to support Knights Landing coprocessors over PCIe, as well as to Knights Landing processors over fabric (e.g., the high-speed cluster interconnect).

Today's COI implementation supports OoF by layering on top of the OpenFabrics Interfaces (OFI). The OFI layer is implemented by libfabric, which is part of different OpenFabrics Enterprise Distribution (OFED) distributions, is in many Linux distributions and can be downloaded from GitHub.

Therefore, the programming is entirely the same for OoF to a Knights Landing processor. It is necessary to set up the system to have a "device" ready that OpenMP can target. That "device" would be setup using COI. To configure the application topology, we can use OFFLOAD_NODES and OFFLOAD_DEVICES environmental variables on compiler/MKL level or COI_OFFLOAD_NODES and COI_OFFLOAD_DEVICES on COI level. The discovery of the available nodes is left to the resource managers in cluster environments (and require support in them).

More information on how to setup and use COI for OOF is covered online (see *For More Information* at the end of this chapter).

SUMMARY

We covered two concepts in this chapter:

1. offload programming model, using OpenMP target directives and
2. the Knights Landing coprocessor.

As with Knights Corner, we can choose to use a Knights Landing coprocessor natively as we would a processor, or program it using an offload model (program is presumed to run on host except when code is specifically marked for offload).

The offload model, as implemented by the OpenMP standard, is the subject of the majority of this chapter. Offloading is supported to coprocessors over PCIe, or to Knights Landing processors via OoF.

FOR MORE INFORMATION

Some additional reading worth considering includes:

- OpenMP specifications (for target directive information): http://openmp.org.
- OoF information, http://lotsofcores.com/OOF.
- OpenFabrics Interfaces (OFI), http://ofiwg.github.io/libfabric.
- Supported Environment Variables in the Intel Compilers documentation, search "Intel Supported Environment Variables" for the latest online documentation.
- James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

CHAPTER

Power analysis

19

In this chapter, we explore the fundamentals of power and performance analysis on Knights Landing using both open-source and Intel tools. Because Knights Landing is compatible with other Intel Xeon processors, the power measurement techniques we cover are also applicable to server systems based on other Intel processors.

What is new with Knights Landing in this chapter?

Knights Landing power thermal, and performance analysis tools are now standard; these methods are the same as those found on other modern Intel Xeon processors.

Today, power is the critical consideration for new large-scale system procurement and operation costs. Power analysis can allow us to be aware of how much power our workloads consume. Because of the scale factor associated with large cluster installations, even a modest saving in a single node power can yield amazing savings in total system power. For example, if we can save even 20 W of total power on a single compute node with software tuning then we can save 200,000 of Watts of power on a 10,000 node cluster which may be worth $210,000/year, or more, in power costs. Power consumption is a continuous operating expense for these large-scale clusters, so it is understandable why power consumption and the efficiency of HPC parallel workloads are getting more attention recently. In this chapter, we show an easy and common method to find the power consumption of our workloads while tuning a typical parallel program on a single node or workstation, using a software-based Knights Landing platform power analyzer.

POWER DEMAND GATES EXASCALE

Power is the main limiting factor today for exascale systems. Some cluster installations today are consuming upwards of 20,000,000 W (20 MW) of power to run ground-breaking scientific workloads and simulations at a very large-scale. The expense for simply powering such a system today is often estimated at $1 million/MW/year (U.S. dollars), a figure that varies around the world. Deployment of such large-scale clusters are typically limited in scaling by the power capacity available to a data center as well as the cooling infrastructure constraints. As the HPC industry marches towards exascale-class systems, there is more and

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00019-3

more of a focus on the peak power consumption of these large-scale systems. The U.S. Department of Energy has set targets for the maximum power consumption of these large HPC systems today at 20 MW. System designers must give consideration for the available power/cooling and how to measure and enforce these power policies at the data center to achieve exascale performance.

To help with this power-limited challenge, Intel has several levels of power management ranging from CPU clock gating and power-capping to managing the power budget at the data center rack-level. In high-performance computing today, we use a metric called performance per watt (PPW) or FLOP/s per watt as a measure of the energy efficiency of a particular computer system. FLOP/s per watt measures the rate of computation (typically floating-point operations per second or FLOP/s) that can be delivered by a computer system for every watt of power consumed. As a software developer, we would use FLOP/s as a measure of the computational performance of our parallel workload.

PPW is how much energy or power we need to spend in order to reach a computation throughput. In other words, PPW is how much power or watts do we have to dissipate to order to get some number of FLOP/s out of the system. There is a very simple formula to compute the PPW of a given system:

$$\text{Peformance Per Watt (PPW)} = \frac{\text{Performance}}{\text{Power}}$$

For example, if we measure our HPC parallel workload (DGEMM) performance in GFLOP/s (double precision) as 3000 and the average total system power consumption during the parallel compute phase as 400 W, then we can resolve what our FLOP/s/W or PPW as:

$$\text{PPW} = \frac{3000\ \text{GFLOPS}}{400\ \text{W}}$$

Using the formula above, we can see that our *pre-production platform* had a PPW of 7.5 GFLOP/s per watt. This means that we will consume 1 W of system power to produce 7.5 GFLOP/s of computation. This is about 2× the compute efficiency of Knights Corner. When measuring PPW, we should only utilize power samples taken within the parallel compute phase. We should not include power measurement data from the initialization and cleanup phase of the application because that could bring the total average total power reading down and this is not a fair comparison of parallel compute efficiency. An HPC system will typically experience peak power consumption during the parallel computation phase of an application, so we should only include these power readings within that window of time in our energy efficiency metrics.

The best known industry benchmark for ranking large-scale system compute power efficiency is called the Green 500 List which is published biannually at www.green500.org. LINPACK is the HPC parallel workload that this forum has standardized on at the time of writing, and GFLOP/s per watt is the measurement of total system power efficiency used.

This chapter describes the methods we can utilize to instrument a single workstation or even a large HPC cluster to understand how much power the application consumes and hence the power efficiency of the parallel workload. For this chapter, we will use a standard benchmark called DGEMM as our sample parallel workload to perform power measurement and FLOP/s/W or PPW indicators.

POWER 101

Power is defined as the amount of energy consumed per unit time, where the unit of power is joule per second (j/s) or more commonly, Watt. We will use watt as our standard unit of measurement for power going forward. The formula for instantaneous wattage power is based on Ohm's Law:

$$\text{Power} = \text{Current} \times \text{Voltage}$$

Current is the flow of electrons through a conductor and amperes (amps) is its standard unit of measurement. Voltage is equal to the work done per unit charge against a static electric field, and voltage is its standard unit of measurement.

We can compute power (watts) simply by multiplying the amperage by the voltage. There are various ways we can measure and report power. For example, we can measure the instantaneous raw power, average power, peak power, minimum power, root mean square (RMS) power, or a moving average applied to the raw power data. For this chapter, we will use a standard called the "1-second moving average." We chose this form of power analysis because it is thermally relevant. This averaging provides correlation to real world measureable thermal events. For example, if we only measure instantaneous power we may see spikes in power levels for very short durations that will not have measureable impacts to the silicon temperature on the heat sink or other thermal solution. In fact, the Knights Landing platform uses the 1 s moving average when reporting the total platform AC power being consumed in watts.

Another common power term is thermal design power (TDP). This is the device or product power rating which is typically specified in watts. The TDP rating refers to the maximum amount of heat generated by the device for which the designed cooling system is required to dissipate while running commercially useful software. This does not mean that power is strictly limited to the TDP rating. The TDP rating is not the same as "peak possible power" but more like a power rating when running with typical parallel applications. A carefully crafted program (e.g., power virus) could possibly exceed the TDP spec, but this would not be a typical workload used in HPC computing. For example, the TDP rating is 215 W for some Knights Landing processor SKUs. The TDP specification is also a good baseline number of the wattage power budget needed to run the Knights Landing to full performance. It is possible for the Knights Landing processor to consume more than the TDP power specification for a short period of time without it being thermally significant because heat will take some time to propagate, so a short spike in power consumption

typically will not violate TDP. To ensure that the Knights Landing processor stays within the thermal specification under such "power-virus" type conditions, the CPU has built-in power management hardware which reduces CPU power by reducing the CPU voltage and/or modulates the CPU clock frequency until the thermal violation is corrected.

HARDWARE-BASED POWER ANALYSIS TECHNIQUES

There are both hardware and software solutions available today to measure the power consumption of our system while running parallel applications on Knights Landing. One of the easiest and most common hardware solutions for measuring total system power is the use of an AC power meter. This is a device that passes power to our server under load and measures the power consumption in real time. Most of these AC power meters have a GPIB or USB interface that can be used for power data logging or to flag the user if we exceed some hard power limit in the system. Fig. 19.1 shows the Yokogawa WT210 digital power meter that we used to confirm the power measurement accuracy of our software-based Knights Landing power analyzer which we build later in the chapter.

Fig. 19.2 shows how to use a hardware-based AC power analyzer to measure a single node or workstation. The advantage of this setup is ease of use and also being nonintrusive to the HPC system under load. The Linux OS and software stack is completely unaware that the HPC system is being monitored and analyzed for power consumption. The remote power data logging system is typically a single desktop or

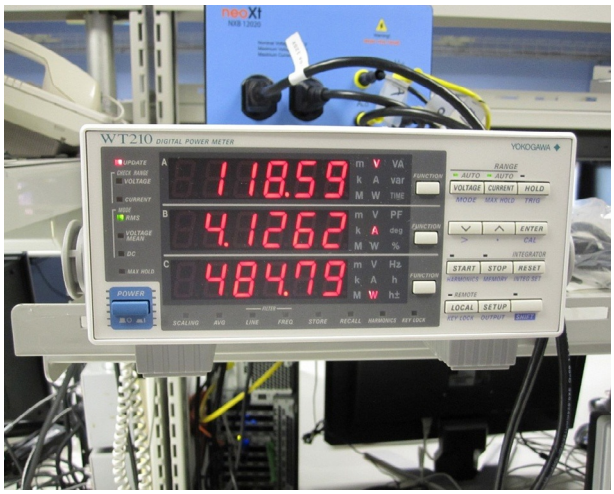


FIG. 19.1

Hardware-based power analyzer used for system power measurement.

Claude's Bench Photo.

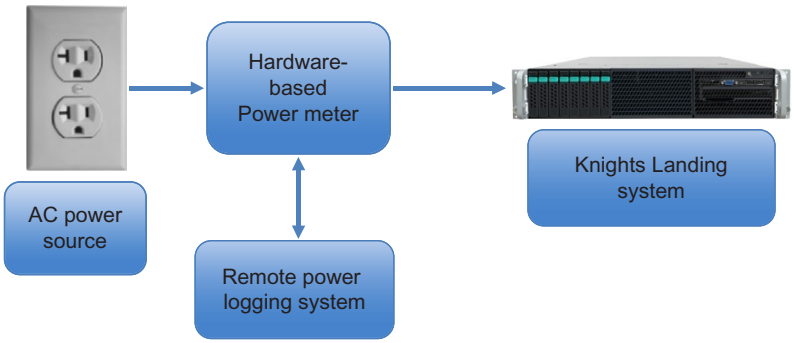


FIG. 19.2

Topology of a typical hardware-based power measurement system.

workstation with a dedicated connection (GPIP or USB) to the AC power meter. Software running on the remote power logging system will sample the AC power reading often enough to provide reasonable power data fidelity, in this case about 100 ms or 10 times per second. By using time stamps on both the power log files and the HPC benchmark runtime output, we can perform temporal alignment to find the compute phase of the parallel program. The Green 500 benchmark will accept the average power of LINPACK across a cluster, but we need to make sure we measure the compute phase and not include the program initialization and cleanup phases, for example.

Another method used to measure system power is the use of power distribution units (PDUs). These are typically used in the data center racks that house HPC systems and clusters, for example. We can think of a PDU as a smart AC power strip that has an embedded controller that is continually monitoring the total power consumption or load of all the devices it supplies with power. This is really useful for measuring power at the larger system scale or cluster level, since every compute node, parallel file system, and network/fabric switch are also drawing power from the same PDUs. We can connect to these smart PDUs using a simple web interface or simple network management protocol (SNMP) to gather and log the total system power. In the case of a large cluster installation, we would simply gather power data from all the PDUs in the cluster and aggregate this power data together to produce a total system power measurement indicator. There are several commercial applications today that can read a collection of PDUs and produce rich indicators showing cluster-level power analysis. Rack-level power measurements can also be collected from the respective PDUs to understand shifts in power consumption due to the range of HPC jobs submitted by the users to the job scheduler, for example. Fig. 19.3 shows a typical PDU used in the data center.

The PDUs used in the data center typically have an embedded microcontroller with an Ethernet port that we can use to remotely access the rack-level power consumption data. Fig. 19.4 shows the useful total system power consumption

FIG. 19.3

A typical smart power distribution unit (PDU).

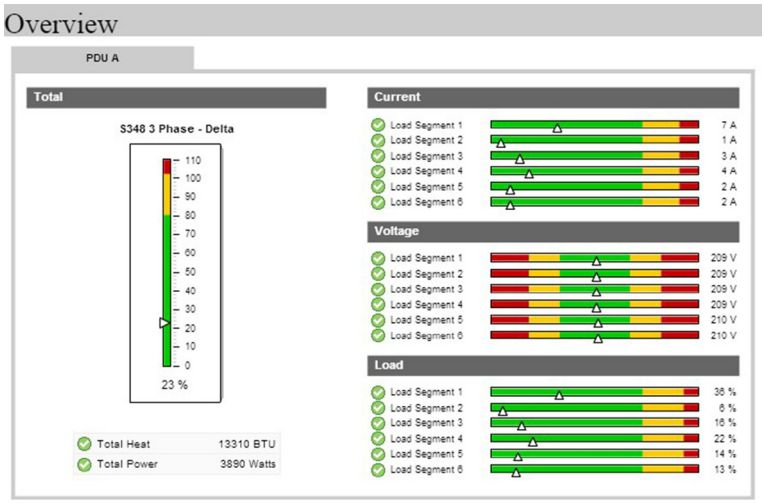


FIG. 19.4

PDU web interface with system power consumption.

information that a user or data center manager can use to monitor and track power usage over time. Notice that we can get the system thermal or heat footprint also. This information can be useful to manage the data center cooling facilities.

By using SNMP, we can sample these PDUs also at a reasonable sample rate and log the total system power into a text file for later plotting and analysis with Excel. Fig. 19.5 shows an example of how we can measure the total power consumption of a cluster of using SNMP to query a group of PDUs to produce a large system total power measurement while running LINPACK over many compute nodes or cabinets. In this example, we wrote some python code to simply query all the PDUs from a text

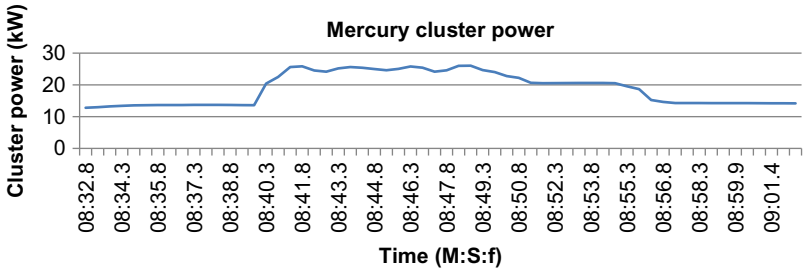


FIG. 19.5

Cluster-level power measurement made with PDUs.

file that described the rack location and PDU IP address, and then we simply aggregated these all together to produce a single cluster-level total power consumption indictor in an Excel spreadsheet.

These hardware-based power measurement devices like the AC power meter and the PDU are a relatively simple method to understand the power consumption of a HPC system under load. By using the built-in interfaces like USB/GPIB/SNMP, we can query these devices for basic power telemetry of our Knights Landing platform while running HPC parallel workloads and log this data to a remote system for later postprocessing and indicators. The main reason we want to use a remote system for the power data collection is to keep the Knights Landing system under test free from the burden of communication and logging while running our parallel workloads. A key concern leading to our wanting to offload power measurement work to a remote system is to be sure it is nonintrusive to the systems we want to measure. However, we may also need to understand where inside the system that our power is being spent. The AC power meter and PDU are a relatively blunt instrument for power analysis, and hence a lot of the detailed information about what component in the system (e.g., CPU, memory, or fabric) is the power hog is abstracted behind a system-level power measurement. In the next sections, we show how to implement an accurate and reliable software-based power measurement solution using the Intel tools and open-source software which has been shown to be accurate and informative without the need to use external power measurement hardware.

SOFTWARE-BASED KNIGHTS LANDING POWER ANALYZER

In this section, we will present a software-based method to measure Knights Landing platform power consumption using Intel tools and standard open-source software. Some key requirements for our Knights Landing power analyzer might be:

- Power monitoring should be nonintrusive to the HPC application and not consume Knights Landing cores/threads for power analysis.

- Support for a decent power sensor sampling rate that will yield good power data fidelity in our log results.
- Network Time Protocol (NTP) used to sync timestamps between our Knights Landing HPC workload under test and the management server logging the power data.
- Generates power data that could be imported into common visualization tools like Excel or Matlab.

Our goal for our software-based power analyzer is to gather the basic power efficiency of the HPC workload running on the Knights Landing platform to produce a baseline of our FLOP/s-per-watt power efficiency. In its simplest form, we would expect basic power consumption measurements as shown in Fig. 19.6.

The Knights Landing platform has the same built-in power and thermal sensors that we find in the systems featuring Intel Xeon processors, and we can query these sensors to gather some useful data for our software-based power analyzer. Some of these sensors are the total system AC power, Knights Landing processor power and silicon temperature, memory power and thermals, and platform intake and exhaust temperatures. We can access these useful power sensors through a standard software interface called the "Intelligent Platform Management Interface" or "IPMI." This is a low-level interface specification that has been adopted by most server hardware vendors, and it allows a system administrator or user to remotely manage data center servers at the hardware level without dependencies on the operating system. IPMI is an open standard that communicates with the server baseboard management controller (BMC) and provides access to the platform hardware and sensors. One of the key features of the BMC is that it is independent of the platform processor and is a very reliable agent in the system for management and gathering system health, including power consumption data.

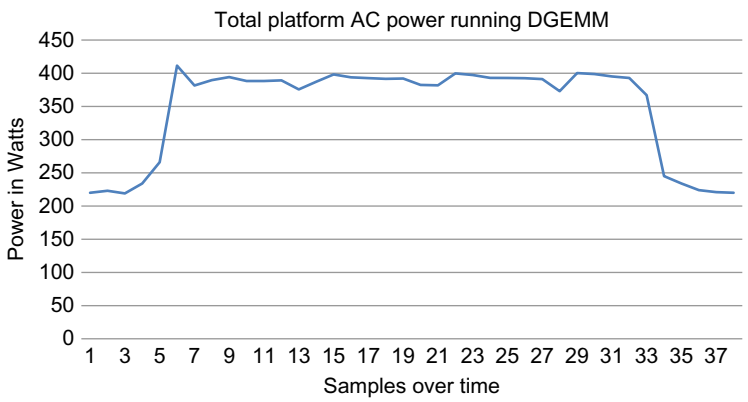


FIG. 19.6

DGEMM workload total AC power profile on Knights Landing.

We start our software-based power analyzer project by ensuring the open-source IPMI driver and ipmitool application are installed on our Knights Landing platform:

$ rpm –qa | grep IPMI

If this package query did not return an IPMI package, then we can install it now and enable it with the following:

Note

Installing the IPMI driver and tool requires elevated user privileges or root access, so we precede commands with the sudo command when running as a user. Seek help from a system administrator if you do not have sudo access.

```
$ sudo yum install OpenIPMI OpenIPMI-tools
$ sudo systemctl enable ipmi
$ sudo systemctl start ipmi
```

We can confirm that the IPMI driver has started service OK:

```
$ sudo ipmitool sdr
```

We should see a system message indicating that the IPMI service is loaded and active. Now we can do a basic test to ensure that the IPMI driver and ipmitool are all ready for us to use for our software-based Knights Landing power analyzer:

```
$ sudo ipmitool sdr
```

This command will display many platform sensor records (SDR) for us including system health, fan speeds, thermals, motherboard voltages, and our main sensor of interest for this chapter: power. Take a look at some of the Knights Landing sensors we can read with this simple IPMI command in Fig. 19.7.

We can see from the listing in Fig. 19.7 that there are many useful power and thermal sensors we can query from the Knights Landing platform using IPMI. Some of these sensors include the system cooling air flow volume rate (CFM), platform inlet and exhaust temperatures, voltage regulator temperatures, cooling fan speed in revolutions per minute (RPM), memory DIMM thermal margin, CPU thermal margin, and AC power supply thermals. The thermal margins displayed in Fig. 19.7 are showing us how close we are to the thermal limits of those system components. In the case of the CPU (P1), this is how far we are (in degrees Celsius) from throttling the CPU clock speed down due to overheating. The status fields in this listing give us a PASS/FAIL indicator also for that system component of interest. However, this list has more sensors than we need for our basic software-based power analyzer, so we want to narrow down the query to just the main AC total power consumption and system thermal-related items in the list. Here is how we can get only the total platform AC power, for example, using the ipmitool application:

```
$ sudo ipmitool sensor get 'PS1 Input Power' 'PS2 Input Power'
```

After we execute this command, we should see something resembling Fig. 19.8.

```
System Airflow    | 34 CFM          | ok
BB P1 VR Temp     | 29 degrees C    | ok
Front Panel Temp  | 22 degrees C    | ok
SSB Temp          | 46 degrees C    | ok
BB P2 VR Temp     | 28 degrees C    | ok
BB Vtt 2 Temp     | 35 degrees C    | ok
BB Vtt 1 Temp     | 27 degrees C    | ok
HSBP 1 Temp       | 26 degrees C    | ok
Exit Air Temp     | 26 degrees C    | ok
LAN NIC Temp      | 50 degrees C    | ok
System Fan 1      | 3822 RPM        | ok
System Fan 2      | 3871 RPM        | ok
System Fan 3      | 3822 RPM        | ok
System Fan 4      | 3871 RPM        | ok
System Fan 5      | 3822 RPM        | ok
PS1 Status        | 0x00            | ok
PS2 Status        | 0x00            | ok
PS1 Input Power   | 4 Watts         | ok
PS2 Input Power   | 88 Watts        | ok
PS1 Curr Out %    | 0 percent       | ok
PS2 Curr Out %    | 9 percent       | ok
PS1 Temperature   | 26 degrees C    | ok
PS2 Temperature   | 28 degrees C    | ok
P1 Status         | 0x00            | ok
P2 Status         | 0x00            | ok
P1 Therm Margin   | -55 degrees C   | ok
P2 Therm Margin   | -57 degrees C   | ok
P1 Therm Ctrl %   | 0 percent       | ok
P2 Therm Ctrl %   | 0 percent       | ok
P1 DTS Therm Mgn  | -55 degrees C   | ok
P2 DTS Therm Mgn  | -57 degrees C   | ok
DIMM Thrm Mrgn 1  | -59 degrees C   | ok
DIMM Thrm Mrgn 2  | -61 degrees C   | ok
DIMM Thrm Mrgn 3  | -63 degrees C   | ok
DIMM Thrm Mrgn 4  | -61 degrees C   | ok
```

FIG. 19.7

A select listing of the Knights Landing platform sensors we will discuss (there are many more sensors available).

Note that there are two AC power supply units (PSU) in the Knights Landing platform used in this example; this is done primarily for redundancy of the power delivery system for the server in case of a PSU failure in the field. We want to read both power supplies for our software-based power analyzer because under heavier power load these two PSUs might share the total system AC power load. The sensor reading label is the main line we want to grab and store in our power logger. A specific Knights Landing platform may only have a single AC power supply, so in this case we could remove the need to query PS2. There are also labels for other key power consumption factors like the average AC power over time and the upper AC power consumption limits of the Knights Landing PSU.

We should also pay attention to the Knights Landing platform internal temperatures while we are running our HPC workloads. Higher processor thermals will cause an increase in the Knights Landing processor silicon leakage current, which

```
Locating sensor record...
 Sensor ID              : PS1 Input Power (0x54)
 Entity ID              : 10.1 (Power Supply)
 Sensor Type (Threshold)  : Other (0x0b)
 Sensor Reading         : 248 (+/- 0) Watts
 Status                 : ok
 Nominal Reading        : 208.000
 Normal Minimum         : 0.000
 Normal Maximum         : 816.000
 Upper critical         : 920.000
 Upper non-critical     : 868.000

 Sensor ID              : PS2 Input Power (0x55)
 Entity ID              : 10.2 (Power Supply)
 Sensor Type (Threshold)  : Other (0x0b)
 Sensor Reading         : 0 (+/- 0) Watts
 Status                 : ok
 Nominal Reading        : 208.000
 Normal Minimum         : 0.000
 Normal Maximum         : 816.000
 Upper critical         : 920.000
 Upper non-critical     : 868.000
```

FIG. 19.8

Knights Landing platform power consumption sensors.

will in fact increase the total system power consumption of the workload even more. There is a handy IPMI command we can use to query the system for all kinds of useful temperature and thermal data including:

- Knights Landing processor
- Memory (DIMM)
- AC power supply
- Voltage regulator
- LAN (Ethernet NIC)
- Inlet and exhaust temperatures

To read these system thermal sensors described above, we simply run this command in our terminal:

```
$sudo ipmitool sdr type temperature
```

This will display a list of thermal sensors we should pay attention to as we are running our Knights Landing platform under heavy compute load. Fig. 19.9 shows us some of these thermal details.

We now look at a simple python script that will sample the total Knights Landing platform AC power and display it on the console. Simply enter the code in Fig. 19.10 into a text file and save as power.py or download this code sample from lotsofcores.com.

We can execute this script using the python interpreter that is built into most Linux OS distributions these days. Enter this command to kick off our most basic use of the software-based Knights Landing power analyzer:

```
$ python power.py
```

```
SSB Therm Trip          | 0Dh | ok  |  7.1 |
Front Panel Temp        | 21h | ok  | 12.1 | 26 degrees C
Inlet Air Temp          | 26h | ok  |  7.1 | 28 degrees C
VRD Temperature         | 2Dh | ok  |  7.1 | 38 degrees C
Exit Air Temp           | 2Eh | ok  |  7.1 | 33 degrees C
PS1 Temperature         | 5Ch | ok  | 10.1 | 33 degrees C
PS2 Temperature         | 5Dh | ok  | 10.2 | 36 degrees C
P1 Therm Margin         | 74h | ok  |  3.1 | -49 degrees C
P1 Therm Ctrl %         | 78h | ok  |  3.1 | 0 percent
P1 DTS Therm Mgn        | 83h | ok  |  3.7 | -49 degrees C
P1 VRD Hot              | 90h | ok  |  3.1 |
P1 Mem01 VRD Hot        | 94h | ok  |  3.1 |
P1 Mem23 VRD Hot        | 95h | ok  |  3.1 |
Mem P1 Thrm Trip        | C0h | ok  | 32.1 |
```

FIG. 19.9

Knights Landing platform thermal sensors.

```
import os
import sys
import time
import subprocess
from datetime import datetime

print ('Press CTRL-C to stop total AC power measurements...')
# Log file header or labels
print('Timestamp                Total System AC Power')

# Run forever till user is done with power measurements with CTRL-C
while True:
    # Get the current timestamp with millisecond accuracy
    timestamp = (
         datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S.%f')[:-3])
    # fetch the power data record via IMPI and stuff into an object
    # Read AC power consumption from power supply unit #1 via IPMI
    power_1 = subprocess.check_output(
                  "sudo ipmitool sensor get 'PS1 Input Power'",
                  shell=True)
    # Read AC power consumption from power supply unit #2 via IPMI
    power_2 = subprocess.check_output(
                  "sudo ipmitool sensor get 'PS2 Input Power'",
                  shell=True)
    # Strip out the unneed text from the power measurement records
    for line in power_1.splitlines():
        if "Sensor Reading" in line:
            power_1 = line.split()[3]
    for line in power_2.splitlines():
        if "Sensor Reading" in line:
            power_2 = line.split()[3]

    # Aggegate the two power supply measurements
    # into a single total power indicators
    total_power = int(power_1) + int(power_2)
    # show the power measurement timestamp and total power data
    print "%s\t\t%s" % (timestamp,  total_power)

    # sample rate time control
    time.sleep(1)
```

FIG. 19.10

Simple Python-based Knights Landing power analyzer code.

When this script is running, it displays a simple control panel showing the total AC power consumption for our Knights Landing platform in a free-running fashion. We have a timestamp with millisecond accuracy and total server power in watts, and we are sampling the total system AC power every second in this example. The timestamp data is useful for us later when doing temporal alignment of the parallel compute phase, so we need to ensure that NTP is installed and running. NTP communicates with an external server to simply keep the time and date accurate for us between all systems. We can now simply pipe the output of this python script to a text file and import that data into Excel to generate some nice power plots and waveforms. Fig. 19.11 shows the text output and a sample power graph from our software-based AC power measurement script.

```
Press CTRL-C to stop total AC power measurements...
Timestamp                    Total System AC Power
2016-02-12 13:21:35.236         348
2016-02-12 13:21:36.585         348
2016-02-12 13:21:37.994         360
2016-02-12 13:21:39.344         348
2016-02-12 13:21:40.675         344
2016-02-12 13:21:42.140         360
2016-02-12 13:21:43.467         356
2016-02-12 13:21:44.794         356
```

Total AC power consumption measurements made with our script will include all the power consumed in the Knights Landing server, such as CPU, memory, fans, drives, fabric, and all electrical devices within the system. This type of measurement is very applicable to Green 500 power measurements where we are looking at the entire compute node energy efficiency. Our script samples the power data at 1 s intervals, but the total AC power is being monitored at a much faster sample rate (typically 100 ms) internally by the BMC power monitoring circuit. This means we do not need to sample the power sensor at a very fast sample rate, since it is being

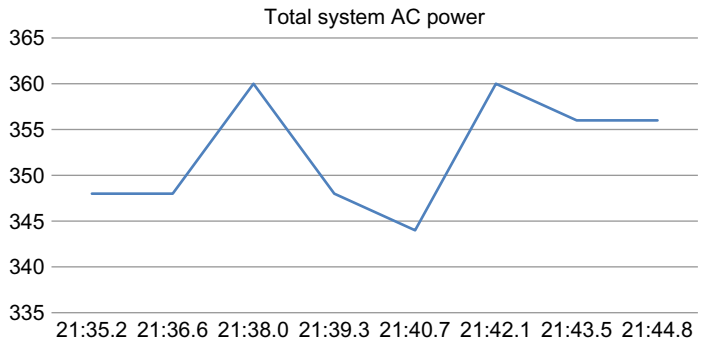


FIG. 19.11

Total AC power measurement script output.

monitored and averaged by internal hardware agents at a good sample rate already. If our code were to query the total power consumption sensors too often, we might impact the HPC workload we are trying to measure. In other words, we might steal more CPU thread time from the HPC workload we are trying to measure than we have to. Every second, our script will sample the latest average power consumption data that is available from the BMC power monitoring circuit.

Our simple power measurement script in Fig. 19.10 works well as a baseline reading of where our parallel application is relative to compute efficiency. However, there is one major issue we should address now with our method. We are doing all the power sensor readings and logging in-band, which means we are intrusive to the host software environment. Recall, we mentioned earlier that one requirement for our software-based power analyzer was to be nonintrusive to the HPC system that is running the HPC workloads. We do not want our IPMI commands and power logging to steal threads away from the Knights Landing processor; we want all available cores and threads free for the parallel workload. Therefore, we are going to make a change to our software-based power measurement design. See Fig. 19.12 for an improved and nonintrusive method we could employ to stay out of the way of the HPC workloads running on the system.

Fig. 19.12 shows that we want to now use an external system or computer to read the power consumption data and perform all logging functions out-of-band (OOB). This will allow the free cores and threads on Knights Landing to work only on the HPC application verses also doing its own power analysis. We will exploit the same IPMI interface we have been using all along, except this time we will perform the measurements OOB or on a separate network port. We will use the Intel Remote Management Module (RMM) interface as our dedicated OOB Ethernet interface to now get access to the platform power and thermal sensors. See the Intel RMM setup guide for instructions on how to setup the Knights Landing RMM interface for external remote access.

The RMM port is a hardware device in the Knights Landing platform with a dedicated Ethernet port that connects directly to the motherboard BMC and will allow us to get power consumption data without bothering the OS running on Knights Landing. In fact, the OS kernel and user will not even know we are connected and polling power data while the HPC parallel application is running. This is what we need to be

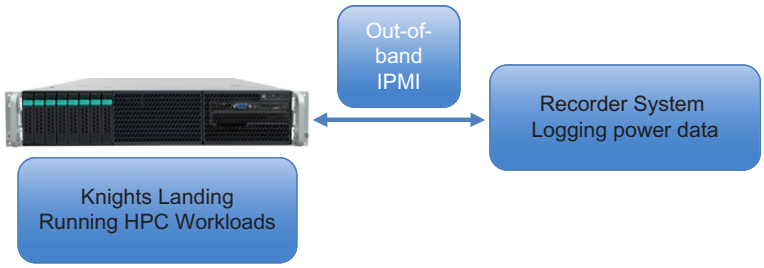


FIG. 19.12

Improved nonintrusive software-based power analyzer.

nonintrusive and stay out of the way of the HPC system while it is doing great scientific computations.

The first thing we need to do is get a remote recorder system setup to act as the power collection and logging agent. We do not need a very powerful system for this also even a low-end desktop system will work since its job will be only to query the remote Knights Landing system for power telemetry and save the data to a file. We will use the standard Ethernet interface on both the power log recorder system and the Knights Landing RMM port.

On the Knights Landing system, there is a dedicated RMM management port on the rear of the server. We connect this management port to the Ethernet switch that is also shared with the remote logging system as this will allow the remote recorder system to see the Knights Landing RMM port and IP address. Next, we need to get an IP address setup for the RMM so that our remote logging system can find it over the network. The platform BIOS setup screen under "server management" will have the IP address of the RMM module we can use. Notice that the RMM IP address is different than the Knights Landing main Ethernet port IP address that the user would use. We now have an OOB network port setup to query power and thermal sensors via IPMI. We can test the connectivity between the remote logging system and the Knights Landing RMM with a simple ping command. From our remote logging system, open a command line prompt in a terminal console and type the command "ping" followed by the IP address we got from the RMM setup from the BIOS. Note that the placeholder IP address used (10.11.12.14) would be changed to your actual IP address.

```
$ ping -c 1 10.11.12.14

PING 10.11.12.14 (10.11.12.14) 56(84) bytes of data.
64 bytes from 10.11.12.14: icmp_seq=1 ttl=64 time=0.669 ms
--- 10.11.12.14 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.669/0.669/0.669/0.000 ms
```

We can see that the remote system and the Knights Landing RMM have established network connectivity. This is our first and basic requirement for our nonintrusive power measurement script. We will now proceed to invoke the ipmitool on the remote logging system to make sure we have the basic IPMI stuff setup.

```
$ ipmitool -I lanplus -H 10.11.12.14 -U <login> -P <password> sdr
```

We insert the BMC login and password that we set in the RMM setup screen, and we should see a listing of all the same BMC sensors that we got earlier on in this section. We now have complete remote IPMI control of our Knights Landing system and can now proceed to monitor the platform power in a nonintrusive manner. Again, the idea here is that we do not want to consume any cores, threads, memory, etc., on Knights Landing that is busy doing real HPC workloads.

We will make a small change also to our power measurement python script, and now we will run this script from the remote logging system as well. Our new script is shown in Fig. 19.13.

```
import os
import sys
import time
import subprocess
from datetime import datetime

# Define the remote system IPMI settings
ipmi_address = <your IPMI address>
ipmi_user = <your IPMI user name>
ipmi_pwd = <your IPMI password>

print ('Press CTRL-C to stop total AC power measurements...')
# Log file header or labels
print('Timestamp                Total System AC Power')

# Run forever till user is done with power measurements with CTRL-C
while True:
    # Get the current time-stamp with millisecond accuracy
    timestamp = ( datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S.%f')[:-3])
    # fetch the power data record via IMPI and stuff into an object
    # Read AC power consumption from power supply unit #1 via IPMI
    power_1 = subprocess.check_output("ipmitool -I lanplus -H" +
                  ipmi_address + " -U " + ipmi_user + " -P " +
                  ipmi_pwd + " sensor get 'PS1 Input Power'",
                  shell=True)
    # Read AC power consumption from power supply unit #2 via IPMI
    power_2 = subprocess.check_output("ipmitool -I lanplus -H" +
                  ipmi_address + " -U " + ipmi_user + " -P " +
                  ipmi_pwd + " sensor get 'PS2 Input Power'",
                  shell=True)
    # Strip out the unneeded text from the power measurement records
    for line in power_1.splitlines():
        if "Sensor Reading" in line:
            power_1 = line.split()[3]
    for line in power_2.splitlines():
        if "Sensor Reading" in line:
            power_2 = line.split()[3]

    # Aggregate the two power supply measurements
    # into a single total power indicators
    total_power = int(power_1) + int(power_2)
    # show the power measurement time-stamp and total power data
    print "%s\t\t%s" % (timestamp,  total_power)

    # sample rate time control
    time.sleep(1)
```

FIG. 19.13

Improved out-of-band power measurement script.

Enter the code in Fig. 19.13 in a text editor and save as power_logger.py or download this code from lotsofcores.com. Substitute the IPMI IP address and user login info from a Knights Landing RMM setup at the top of the program and execute this script using the built-in python interpreter:

```
$ python power_logger.py
```

This code produces the same output and power consumption measurements as our first code example in Figs. 19.10 and 19.11 except now we are making our power measurements OOB on a remote logging system. This script could also be expanded to query more than one Knights Landing platform and hence could be improved to measure the total power consumption of a small cluster.

ManyCore PLATFORM SOFTWARE PACKAGE POWER TOOLS

This section will cover some useful power and performance analysis tools that come bundled with the ManyCore Platform Software Package (MPSP) software releases from Intel. The Linux Kernel delivered along with MPSP is based on your specific OS distribution kernel. MPSP additions are Knights Landing specific patches, which enable different core functionalities of the Processor like CPU power and performance indicators. Another useful package that comes with MPSP is micperf, which includes come common HPC workloads that we can use to test our software-based power analyzer we developed in the previous section. Some of these HPC parallel workloads bundled with micperf include HPLinpack, SGEMM, DGEMM, SHOC, and the memory bandwidth test STEAM. Some of the specific power and performance monitoring tools include cpupower and turbostat. The MPSP user guide gives step-by-step instructions on how to install MPSP and use the software power tools we will use in this section.

We start with how to measure the Knight's Landing processor and DDR DIMM memory power to get a deeper analysis of the system power consumption while running our HPC workloads. The MPSP tool that we will use for this job is called turbostat, and this comes bundled with MPSP. Fig. 19.14 shows the many useful measurements we can leverage to get a better understanding of where power is being spent in the system. Some of these indicators include CPU utilization, clock rate, CPU residency time in C1/C6/PC3 idle states, CPU silicon core and package temperatures, and CPU/memory power consumption. The turbostat utility included in MPSP gets this CPU and Memory power wattage information by reading the running average power limit (RAPL) model specific registers (MSRs).

With MPSP installed on our Knights Landing system, we run the turbostat power/thermal/performance profiling tool and see how we can display the key indicators in a free-running control panel.

```
$ sudo turbostat --interval 1 --Summary
```

Note

Because the turbostat application needs to access some model specific registers (MSR), we must execute this command with elevated privileges. Seek help from a system administrator if you do not have sudo access.

This command line argument will kick off turbostat as a free-running monitor and display some very key power and performance measurements in the background

FIG. 19.14

CPU power, thermal, and performance statistics from turbostat.

while we run some basic HPC workloads. The -interval switch above sets the sample rate of the sensors and performance counters to once per second, as opposed to the default sample rate of every 5 s with the current release. The -Summary switch tells turbostat to only display the basic power and performance statistics for us. We can get more usage info for turbostat by invoking the manpage with "man turbostat". In Fig. 19.14, the first row of statistics is a summary for the entire Knights Landing system. For the CPU residency time % columns, the summary is a weighted average, and for the Temperature columns, the summary is the column maximum. For the CPU and memory power consumption columns, the summary is the total measured in watts.

To produce some useful power and thermal measurement data, we can simply pipe the output from turbostat to a log file and import that data into Excel for some plots and graphs giving a clear understanding of where we are consuming power and CPU thread time, etc. Type the following command in our console to start the power and performance measurements and pipe that measured data to a log file called power.log:

```
sudo turbostat --interval 1 --Summary > power.log
```

This command will immediately start measuring power and performance data and log it to a file called power.log until we press ctrl-c to stop logging. This file is delimited by spaces, so it is easy to parse out with the Excel built-in graphing and plotting tool as shown in Fig. 19.15.

We now have a handy method to measure the Knights Landing processor power, thermals, and utilization with a single command line argument. The power consumption of the on-die MCDRAM is also included in the power measurement, so we have good visibility into the main power consumers inside the Knights Landing CPU package. As we make optimizations to our parallel source code or compiler, we can use this handy turbostat tool to see the impact to the Knights Landing package power consumption and thermal conditions.

RUNNING AVERAGE POWER LIMIT

The TDP specification represents the maximum amount of power the cooling system in a system is required to dissipate. For a Knights Landing with a TDP rating of 215 W, this means that if the system cooling solution is capable of dissipating that much heat, the CPU will operate and perform as specified. The TDP specification is therefore the power budget under which the system needs to operate for normal HPC workloads.

RAPL can help us keep our system within TDP specification by monitoring power and thermal conditions and by taking action to limit power consumption. RAPL also exposes a set of internal CPU counters providing energy and power consumption information by using PAPI (Performance API). RAPL is not the same as a dedicated hardware power measurement device, but rather it uses a combination of

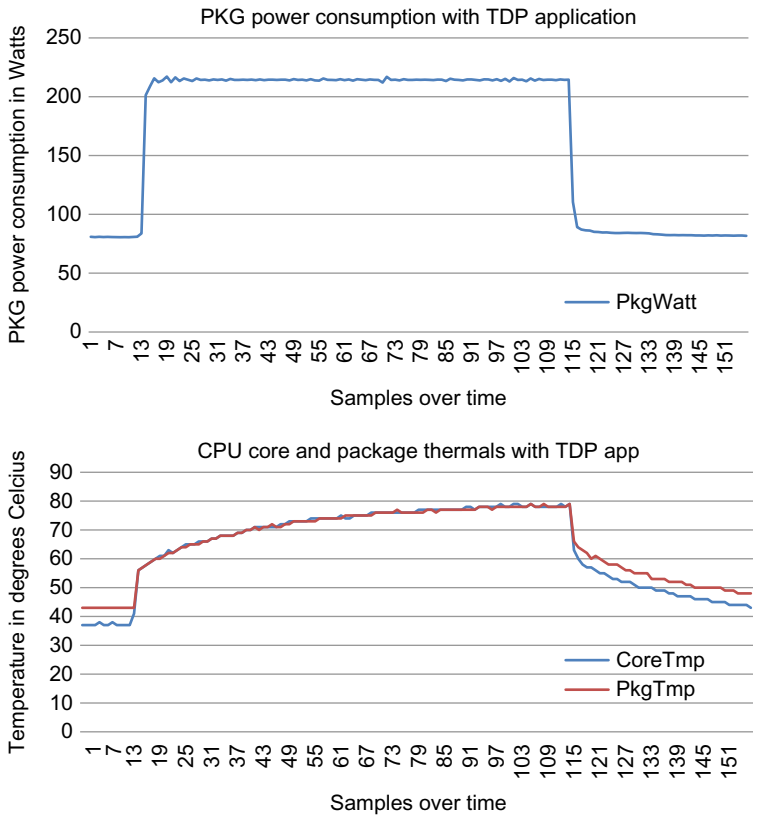


FIG. 19.15

CPU package power and thermal measurements with TDP application.

current sensor data from platform voltage regulators and a software-based power analysis model that estimates power consumption by leveraging some CPU hardware performance counters and memory models.

Some of the key features of RAPL include:

- ENERGY_STATUS for power consumption or analysis
- POWER_LIMIT and TIME_WINDOW for controlling power consumption over time
- PERF_STATUS for monitoring the performance impact of the power limit
- RAPL_INFO contains information on measurement units

RAPL provides approximately 1 ms resolution measurements, and the sensor data is read via MSRs and made available by the Linux kernel via the device file system (devfs), which makes these measurements readily available to profiling tools like Intel VTune Amplifier XE.

RAPL gives us a method to limit or cap the CPU and memory power consumption and may allow a power monitoring program to dynamically limit the power consumption to match the system power and cooling budget. RAPL supports both short- and long-term power limit windows which can be adjusted dynamically by the user or power monitoring application.

Power capping is implemented in the Linux Power Capping Class driver on the Knights Landing platform. The power capping framework provides a consistent interface between the Linux Kernel (Ring 0) and the user space (Ring 3) that allows the power capping driver to expose the settings to user space in a familiar way. This is an interface to set the limits on maximum power consumption and exports a wealth of power data through sysfs thus allowing a user or a process to access (read) RAPL power data without elevated privileges. This feature is widely used today for cluster monitoring applications, such as ORCM, Nagios, Ganglia, etc. The objects at the root level of the Powercap tree represent "control types," which correspond to the different methods of power limiting in the system. For example, the intel-rapl control type represents the Intel RAPL power-limiting methods. Power zones represent the various parts of the system (i.e., which CPU socket), which can be controlled and monitored using the power capping method determined by the control type a power zone belongs to. These zones each contain statistics for monitoring power consumption (energy_uj), as well as controls represented in the form of power constraints (i.e., power limits). If the various power zones in the system are hierarchical, then those power zones may also be organized in a hierarchy with one parent power zone containing multiple subzones to show the power control topology of the system.

Each of the power zones and subzones contains power monitoring attributes (energy consumption in Joules) and constraint attributes (power limits) allowing power controls to be applied. Each power zone in the tree also contains a name attribute that we can query:

```
$ cat /sys/class/powercap/intel-rapl/intel-rapl\:0/name
```

When we execute the command above in the terminal, we will see the power zone name as "package-0" or our Knights Landing primary CPU package. We can also query Powercap for the power limits currently set on our system. If we look inside of the sysfs file system, we can see the power capping attributes:

- constraint_0_name
- constraint_0_power_limit_uw
- constraint_0_time_window_us
- constraint_1_name
- constraint_1_power_limit_uw
- constraint_1_time_window_us

The Intel RAPL feature allows two kinds of power limiting constraints, short- and long-term, with two different time windows to be applied to each power zone. Another way to look at this is we have both a small and larger window of time that

we will allow power consumption to exceed these defined power limits. Short-term power capping is typically used to protect against power and current limitations on the system power supply, while long-term power capping is typically used to enforce the system thermal constraints or TDP. We can use this sysfs interface to set limits on how much power consumption the Knights Landing processor can consume over a given time window. We take a look at the two power capping constraints set on our Knights Landing platform:

```
$ cat /sys/class/powercap/intel-rapl/intel-rapl\:0/constraint_0_ power_limit_uw
215000000
$ cat /sys/class/powercap/intel-rapl/intel-rapl\:0/constraint_1_ power_limit_uw
258000000
```

We now see the two power capping values, or power constraints for long- and short-term durations. The Knights Landing package power limit values are shown in microwatts (μw), so this comes out to 215 and 258 W, respectively. These are the absolute power consumption values that will cause the power capping feature to limit power consumption via simple CPU throttling down of the core clock and voltage. With a Knights Landing processor TDP rating of 215 W, we can exceed the TDP rating for a short period of time that would not be thermally relevant or cause our CPU package to heat up very much. Now there is also a time value that goes with power limiting, where we specify how long the power violation must persist before we take action and throttle. To see these time constraint values (in microseconds), we simply need to look at the time windows for time constraint 0 and time constraint 1:

```
$ cat /sys/class/powercap/intel-rapl/intel-rapl\:0/constraint_0_time_window_us
999424
$ cat /sys/class/powercap/intel-rapl/intel-rapl\:0/constraint_1_time_window_us
9760
```

We can see that the time window for power constraint 0 is 1 s and is about 10 ms (rounded) for power constraint 1. In the power attributes above, "constraint_0_power_limit_uw" is the power limit in microwatts, which should be applicable for the time window specified by "constraint_0_time_window_us" in microseconds. Both of these have power capping read/write file attributes and can be set by the user. The two power capping attributes above can be setup for long- (Constraint 0) and short-term (Constraint 1) power violations. If the power exceeds "constraint_0_power_limit_uw" for more than "constraint_0_time_window_us," then the CPU will throttle the clock and reduce CPU core voltage until the power limit is met.

Power capping with RAPL will force the Knights Landing processor to stay within some defined band of power consumption and may be very useful in the case where the system may be power limited. We use a higher power cap value for constraint 1 because the defined time window is for a much shorter duration of time than constraint 0. We will allow a very short-term burst of higher power consumption because the bulk capacitance in the system power supply can help deliver extra

power for a short duration of time without causing a significant thermal condition in the system. Likewise, for long-term power capping we will allow a longer duration of excess power consumption but at a much lower wattage than short-term power capping to ensure the excess power consumption is not thermally significant. This RAPL power limiting feature can also be used to target a set of compute nodes to stay within some power budget regardless of the high degree of code vectorization which typically can increase overall CPU power consumption in a system. For more information on how to use RAPL to limit power consumption in your system, see *For More Information* at the end of this chapter.

PERFORMANCE PROFILING ON KNIGHTS LANDING

There is a useful performance profiling tool included with Linux called perf_event or "perf," and we can call upon this program to profile our Knights Landing system while we run our HPC parallel workloads. RAPL energy consumption readings are now available via the perf_event interface as of Linux kernel version 3.14 and higher. This application can profile several key performance events that a software developer might tune for both power and performance. Fig. 19.16 shows some of the key performance and power predefined events we can query using the perf_event application.

As we can see in Fig. 19.16, we have some very useful performance counters we can leverage with the perf_event tool. The perf application has many profiling options like logging perf data to a file, comparing the perf results from various parallel workload runs, and memory access profiling. See the man page for the perf application for more info and usage models. We can use these predefined event strings in Fig. 19.16 to quickly measure power or performance in an area of interest. Since this chapter is all about power, we start with the energy consumption of the Knights Landing processor package power. Enter the following command in our console:

```
perf stat -a -e " power/energy-pkg/"
```

This command will invoke profiling of the CPU package power consumption and will continue to collect energy usage data until we stop profiling by pressing ctrl-c on the console. The perf_event tool reports energy consumption in Joules over time instead of simply watts like we saw with other programs. There are two key measurements, reported by perf_event for energy or power consumption, Joules and Time:

```
1446.45 Joules power/energy-pkg/
16.865334510 seconds time elapsed
```

We can convert Joules and Time back into watts with this simple equation:

$$\text{Power in Watts} = \text{Energy in Joules} / \text{Time in Seconds}$$

```
  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]

  alignment-faults                                   [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]

  L1-dcache-load-misses                              [Hardware cache event]
  L1-icache-load-misses                              [Hardware cache event]
  L1-icache-loads                                    [Hardware cache event]
  LLC-load-misses                                    [Hardware cache event]
  LLC-loads                                          [Hardware cache event]
  LLC-prefetch-misses                                [Hardware cache event]
  LLC-prefetches                                     [Hardware cache event]
  LLC-store-misses                                   [Hardware cache event]
  LLC-stores                                         [Hardware cache event]
  branch-load-misses                                 [Hardware cache event]
  branch-loads                                       [Hardware cache event]
  dTLB-load-misses                                   [Hardware cache event]
  iTLB-load-misses                                   [Hardware cache event]
  iTLB-loads                                         [Hardware cache event]

branch-instructions OR cpu/branch-instructions/      [Kernel PMU event]
branch-misses OR cpu/branch-misses/                  [Kernel PMU event]
bus-cycles OR cpu/bus-cycles/                        [Kernel PMU event]
cache-misses OR cpu/cache-misses/                    [Kernel PMU event]
cache-references OR cpu/cache-references/            [Kernel PMU event]
cpu-cycles OR cpu/cpu-cycles/                        [Kernel PMU event]
instructions OR cpu/instructions/                    [Kernel PMU event]
power/energy-pkg/                                    [Kernel PMU event]
power/energy-ram/                                    [Kernel PMU event]
```

FIG. 19.16

Power and performance event types we can monitor with perf_event.

```
1446.45 Joules / 16.865334510 seconds = 85.7 Watts
```

We can also use this tool to measure the DDR DIMM memory power consumption by passing the "power/energy-ram/" event type to perf_event:

```
$ perf stat -a -e "power/energy-ram/"
54.93 Joules power/energy-ram/
17.470058155 seconds time elapsed
```

Using the equation above, we can calculate the Knights Landing MCDRAM power consumption by converting Joules back to watts again:

```
54.93 Joules / 17.470058155 seconds = 3.14 Watts
```

We can also use the perf_event tool to profile our parallel application for some key event types like the number of instructions retired or CPU cache misses. We can compare the number of CPU instructions retired for both idle condition and running a TDP application like DGEMM. While the Knights Landing system is idle, we measure the rate of retired CPU instructions:

```
$ perf stat -a -e "instructions"
Performance counter stats for 'system wide':
16,169,895 instructions
4.153776531 seconds time elapsed
```

Next, we run a good parallel program like DGEMM and measure the number of retired instructions over roughly the same period of time:

```
683,773,070,273 instructions
4.146878932 seconds time elapsed
```

Fig. 19.16 contains many more useful performance event types a SW developer can call upon to profile a HPC parallel application. Take a look at the man page for perf_event with the "man perf" command at the console or google some more examples of how to use this performance profiling tool for your own application needs.

INTEL REMOTE MANAGEMENT MODULE

In this section, we will cover a nifty feature of the Knights Landing called the Remote Management Module or RMM. This feature allows an administrator or user to remotely monitor the health of the Knights Landing platform using a nice web interface or even low-level IPMI commands over the network. The underlying mechanism of the RMM is the platform BMC and the IPMI interaction that we have been using for the software-based power analyzer in the previous section. Another key feature of the RMM is that it is implemented using an OOB networking method. This means that the RMM has an Ethernet network port that is not exposed to the Linux OS and hence is a very reliable agent to manage Knights Landing platform remotely. Consider the simple case of an in-band management interface where the user could disable or crash the Linux kernel or network services thereby bringing down the main Ethernet connection. By exploiting an OOB method, we have some protection from rogue user code or deliberate in-band network hacking. The RMM is also powered from the platform standby power, which is great in case a user accidentally shuts the system down remotely. The administrator can still power up the platform again with a simple IPMI command or visit to the RMM webpage. In fact, the Knights Landing RMM IP address we used in the previous section can be entered into a web browser to get access to a super useful admin webpage. We enter the RMM IP address into a web browser web address field, and we are be greeted by a login screen for the RMM module. After you login with the user name and

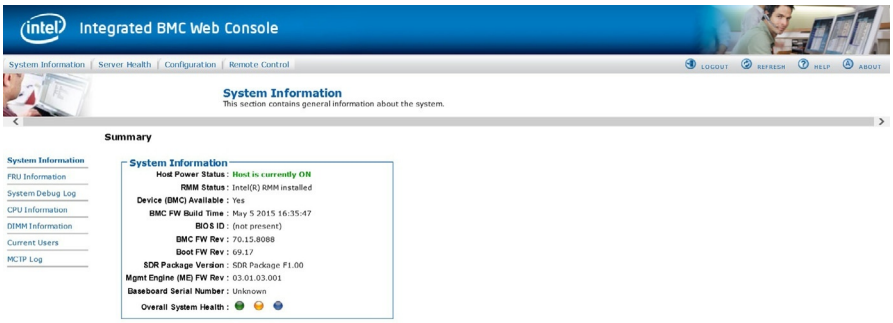


FIG. 19.17

Knights Landing Remote Management Module webpage.

password that was setup in the BIOS, we should see something like Fig. 19.17 showing us the RMM webpage for basic system information.

Fig. 19.17 shows the RMM BMC web console, which is a great interface to remotely manage a Knights Landing platform. The top of the RMM webpage has four main tabs: System information, Server Health, RMM configuration, and lastly remote control. Some of these remote activities might be to monitor the platform BMC sensors like we did in the previous section. Another common use case for the RMM is to remotely reboot the Knights Landing system if it were hung by some buggy user code. We can even change the BIOS settings for the Knights Landing platform remotely using this handy Intel tool. To get a quick look at the Knights Landing platform health, we look at the server health tab as shown in Fig. 19.18.

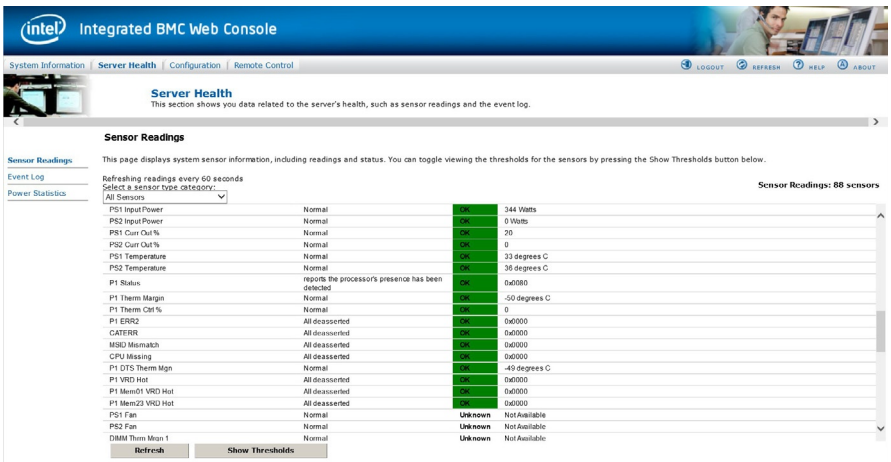


FIG. 19.18

Knights Landing health check from RMM webpage.

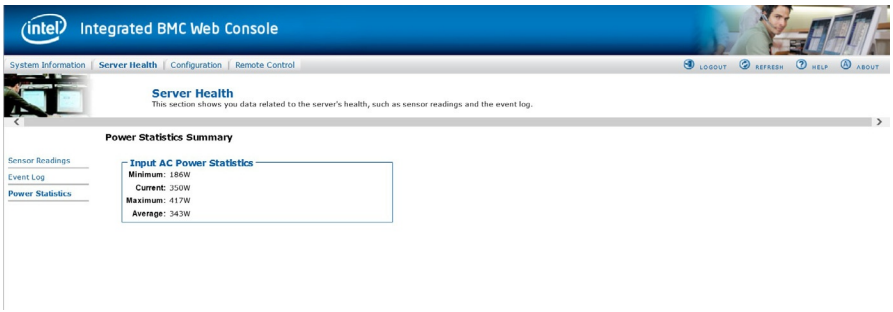


FIG. 19.19

Knights Landing power consumption from the RMM webpage.

We can see much of the same sensors that we can query directly with the ipmitool like we did in the previous section, except here we have a nicer GUI to work with plus color codes health indicators. Notice also our familiar power consumption sensors at the top of the sensor reading dump in Fig. 19.18. There is also a power statistics link on the left side of the server health screen as well. Click on the Power Statistics link now.

As shown in Fig. 19.19, we can see the total Knights Landing platform power consumption reported as the most recent power sample, minimum, maximum, and average power consumption over time. The power statistics shown here are very useful to see what the power consumption range is for a given workload on a Knights Landing platform or compute node.

The Intel RMM module and embedded web interface are another very useful tool for administrators and software developers to use to remotely manage a Knights Landing platform or compute node OOB and nonintrusive to the software environment running on the node. We can use this interface to gather useful health indicators for our platform, get access to diagnostic information, measure power consumption and thermals, change the BIOS settings, or simply reboot a hung node. See the Intel RMM User Guide for more information about initial setup and even more useful options.

SUMMARY

In this chapter, we discussed some common methods to expose the power consumption of a Knights Landing platform using both hardware and software methods.

We produced a software-based Knights Landing power analyzer using the open-source ipmitool and simple python scripts that would sample the total AC power consumption while running key HPC workloads. By using NTP to time-sync our power logging system to the Knights Landing system under load, we are able to perform

temporal alignment of the parallel compute phases of the HPC workload and hence compute an effective FLOP/s-per-watt measurement for compute efficiency. The output of our software-based power measurement script could easily be imported into Excel or Matlab for even more analysis on the power profiles created.

We covered how to measure the Knights Landing processor and DDR DIMM memory power consumption and thermals by using some of the useful software tools included in the MPSP releases including Turbostat. Some key performance data is also provided with the turbostat tool like Knights Landing CPU utilization, time spent in idle states, and actual CPU clock speed at various parts of our parallel program.

We showed how to utilize RAPL to keep the system power within some defined infrastructure power and cooling budget. By using the Powercap driver exposed in standard sysfs, we can limit the amount of power consumption our CPU can consume under any parallel workload to stay within a system or cluster power budget.

We showed how to use the built-in Remote Management Module or RMM to also gather power consumption indicators for a Knights Landing platform. The RMM uses a nice webpage and GUI to help quickly see the health of the Knights Landing system, perform basic maintenance like BIOS updates, reboot a hung server, or simply to monitor the minimum and maximum power consumption of our platform remotely.

Going forward, the entire HPC system ecosystem will have to address all the various hardware and software areas of power and thermal management in order to reach Exascale compute capabilities. As software developers, we will also need to be aware of the compute power efficiency of the parallel workloads before deploying to large clusters, as it is easier to tune the HPC workload before it is deployed at a much larger scale.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this chapter.

- Intel® Xeon Phi™ processor website: http://www.intel.com/XeonPhi.
- Intel Remote Management Module, http://lotsofcores.com/Intel-RMM.
- For more information on how to use RAPL to limit power consumption in your system, visit https://01.org/rapl-power-meter.
- Intel ManyCore Platform Software Package (MPSP), http://lotsofcores.com/MPSP.
- For additional power measuring techniques, visualization ideas—Taylor Kidd, Rob Farber, Belinda Liviero, and Evan Felix, Chapter 25—Power Analysis for Applications and Data Centers, in High-Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, edited by James Reinders and Jim Jeffers, 2015, ISBN 978-0-12-803819-2.

- For Knights Corner specific power analysis—Claude J. Wright, Chapter 14—Power Analysis on the Intel Xeon Phi Coprocessor, in High-Performance Parallelism Pearls (Volume One): Multicore and Many-core Programming Approaches, edited by James Reinders and Jim Jeffers, 2015, ISBN 978-0-12-802118-7.
- OpenIPMI, http://openipmi.sourceforge.net/.
- Green 500 List, www.green500.org.
- Download the code from this, and other chapters, http://lotsofcores.com.

SECTION

Pearls III

This section focuses on parallel programming in full applications with examples with notes on Knights Landing specific results and optimizations. Chapters 20–26 build upon High-Performance Parallelism Pearls*, volumes one and two by being among the first published examples using the high-bandwidth memory (MCDRAM) and the clustering modes in the Intel Xeon Phi processors. We believe that we all learn a great deal when we see the work of other experts. We highly recommend benefiting from the excellent work shared in this section and both of the Pearls books. The techniques found in these many examples really bring home how to harness the power of multicore processors, including Knights Landing.*

This book has three sections: (I) Knights Landing, (II) Parallel Programming, and (III) Pearls. The book also has an extensive Glossary and Index to facilitate jumping around the book.

CHAPTER

20 Optimizing classical molecular dynamics in LAMMPS

What is new with Knights Landing in this chapter?
AVX-512, AVX-512CD, AVX-512ER, MCDRAM

LAMMPS is a software package that performs classical molecular dynamics simulations. It was first developed at Sandia National Laboratories in the mid-1990s for large-scale parallel computation and has grown to be very popular due to its versatility and support for a wide range of problems including materials science and molecular biophysics. In this chapter, we describe optimizations to improve LAMMPS performance that are applicable to both many-core Intel® Xeon Phi™ processors (codenamed Knights Landing) and recent generation multicore Intel® Xeon® processors. Models for simulation of soft matter, biomolecules, 3-body potentials, and aspherical particles will be discussed. We will show benefits of optimization using standard benchmark datasets and production simulations currently being used to model solar cells and hydrocarbon lubricants.

MOLECULAR DYNAMICS

Molecular dynamics (MD) simulation is a widely used approach to study the time evolution of a system of "particles," typically atoms or molecules with defined properties. Example applications include simulation of how a therapeutic drug will interact with molecules in the body, simulation of the microscopic mechanism of water droplet freezing, calculation of thermodynamic and rheological properties of different hydrocarbon mixtures, and simulation to determine how a specific blend of organic molecules will influence properties that determine solar cell efficiency.

The input for MD includes (1) initial particle positions/velocities and other model-specific parameters (particle charge, type, rotation, bond topology, etc.); (2) the equation for the energy of the system; (3) the type of boundary conditions (periodic, fixed, shrink-wrapped, reflecting, etc.); (4) the statistical ensemble from which the simulation is sampled; and (5) the options for logging particle movement and any in-situ statistics and visualization options (in-situ calculations and rendering are both performed during the simulation as opposed to simulation followed by postprocessing). The equation for the energy of the system is typically referred to as the force-field or the potential energy model (often referred to as a potential).

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00020-X

The statistical ensemble describes the possible states for the system of particles that will depend on whether it is desirable to perform the simulation with constant pressure and temperature (NPT ensemble), constant volume and temperature (NVT or canonical ensemble), or constant volume and energy (NVE or microcanonical ensemble).

The fundamental steps for each iteration of a MD simulation include (1) calculation of the force on each particle as the gradient of the energy with respect to position/rotation; (2) time integration to calculate the new particle positions/velocities with respect to the force; (3) thermostat/barostat calculations for NPT/NVT simulations; (4) calculation of relevant statistics; and (5) output of data and restart files. In contrast to quantum MD, classical MD uses less computationally costly empirical potentials to determine the system energy allowing for better time complexities, more efficient parallel decompositions, and ultimately a capability to simulate much larger systems for longer time scales. As a consequence, many different potential energy models (force-fields) can be used. For biomolecules, a typical force field is fit to terms for van der Waals energies (U_{VDW}), electrostatic energies (U_{C}), and energies due to covalent bonds between atoms in the system (U_{S}, U_{A}, U_{D}),

$$U_{\mathrm{Bio}} = \sum_i \sum_{j>i} U_{\mathrm{VDW}}(r_{ij}) + \sum_i \sum_{j>i} U_{\mathrm{C}}(r_{ij}) + \sum_{\mathrm{bonds}} U_{\mathrm{S}}(r) + \sum_{\mathrm{angles}} U_{\mathrm{A}}(\theta) + \sum_{\mathrm{dihedrals}} U_{\mathrm{D}}(\varnothing)$$

where r represents a distance between pairs of atoms, θ is the angle formed by two bonds shared by an atom, and ϕ is the dihedral angle for a bond involving four covalently bonded atoms. Because the number of covalent bonds for an atom is limited to a small number, the time complexity for the bonded interactions is O(N) for N atoms. However, the nonbonded terms in the equation are calculated over all pairs of atoms, and therefore the force-field is described as a *2-body potential*. Without approximation, this would result in an O(N^2) time complexity.

In order to improve performance for larger systems, it is typical to exploit the rapid decay in the van der Waals energy with distance (typically $1/r^6$) by using a *cutoff* distance beyond which the energy is 0. This is achieved by storing a neighbor list for each atom. Rather than looping over all atoms in the system, the inner loop is evaluated for all *neighbors* of each atom. The neighbor list is constructed by first binning atoms into cells and then for each atom, looping over all atoms in neighboring cells to create a list of atoms within the sphere determined by the cutoff. The resulting time complexity for U_{VDW} is O(N) because the cell size is typically much smaller than the simulation box size. In order to avoid rebuilding the neighbor list every timestep, a *skin* distance is typically added to the cutoff such that no atom should move farther than the skin distance within the neighbor-list build interval (Fig. 20.1).

Although the van der Waals energy can be evaluated as a short-range problem in classical MD, the decay in the coulombic term is reduced (typically $1/r$), and therefore, long-range interactions are unavoidable for many simulations in molecular biophysics. For periodic boundary conditions, the problem becomes worse because the energies between atoms in multiple periodic cells are nonnegligible. In order to make

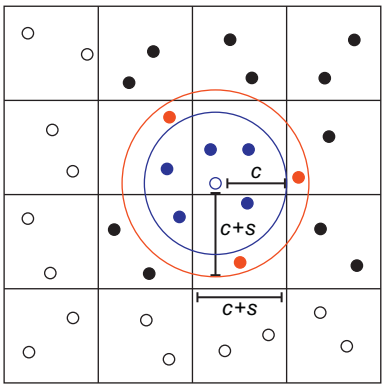


FIG. 20.1

2D illustration of the division of the simulation box into cells for neighbor-list builds. In this case, the cell dimensions are equal to the cutoff (c) plus the skin distance (s) so that no more than nine cells need to be searched for neighbors (for 2-body potentials even fewer cells are searched due to symmetry from Newton's third law). On the initial time step, only atoms in blue will contribute to the energy. The skin distance should be parameterized such that no neighbor outside of the cutoff and skin distance can move within the cutoff before the next neighbor-list build.

evaluation of U_C tractable, Ewald summation is typically used to split the electrostatic energy into a summation over rapidly varying short-range interactions in real space (U_R) and a Poisson summation allowing the long-range interactions to be computed in reciprocal space (U_K). U_R and U_K are chosen such that U_R is negligible beyond some cutoff distance, U_K is a slowly varying function for all distances, and $U_C = U_R + U_K$. This allows for standard short-range cutoff evaluation for the real space calculation (together with the van der Waals) and also for the Fourier transform to be represented with few k vectors. Although the Ewald approach facilitates an accurate method for calculation of electrostatic interaction energies in a periodic box, the best implementations have a time complexity of $O(N^{3/2})$. For this reason, several methods have been proposed for representing the charge density for reciprocal space calculations on a mesh. This allows for a discretized solution to Poisson's equation using FFTs and an overall time complexity of $O(M\log M)$, where M is the number of k-space mesh points. For typical accuracies of interest in MD calculations, a mesh spacing that yields $M \approx N$ is typical. The most popular variants of these particle mesh methods include particle mesh Ewald, smooth particle mesh Ewald, and particle-particle particle-mesh (P^3M); see Deserno et al. in *For More Information*, at the end of this chapter, for a comparison of methods.

For materials applications, the potential energy models can be more diverse. Electronic screening can limit the distance for charged interactions and allow for simulation without long-range coulombic terms; however, the equations describing the short-range interactions can be very complex. In some cases, 3-body or

many-body potentials are used. For 3-body potentials, all triplets of atoms are considered in addition to all pairs,

$$U_\tau = \sum_i \sum_{j>i} U_2(r_{ij}) + \sum_i \sum_{j\neq i} \sum_{k>j} U_3(r_{ij}, r_{ik}, \theta_{ijk})$$

Potential energy models have also been developed for coarse-grain simulation of aspherical particles such as the Gay-Berne potential often used to model liquid crystal mesogens (individual liquid crystal molecules represented as a coarse-grain particle).

LAMMPS

LAMMPS is a software package that performs classical MD simulation. It is popular due to its versatility and support for a wide range of potential energy models, long-range solvers, and simulation options. It is open-source, licensed under the GNU General Public License, with a large user base and active mailing list.

LAMMPS now exceeds 500 K lines of source code. To keep build times manageable, encourage modularity, and decrease binary sizes, LAMMPS is organized into *packages* and has only limited functionality in the core code base. Installing a package, for example, the KSPACE package for Ewald and P^3M calculations, is as simple as typing "`make yes-kspace`" on the command-line to copy the source files from the package subdirectory to the build directory.

LAMMPS supports a wide range of potential energy models, constraints, time integration options, and statistics computations in a modular manner with a variety of different *style* choices that are specified by the user in an input script. At a high level, each style is implemented in the code as a C++ virtual base class with an appropriate interface to the rest of the code. For example, the choice of potential (e.g., lj/cut for Lennard-Jones with a cutoff) selects a pairwise interaction model that is used for force, energy, and virial (terms used to determine pressure) calculations. Individual pair styles are child classes that inherit the base class interface. Thus, adding a new pair style to the code (e.g., lj/cut/dipole/cut) is as conceptually simple as writing a new class with the appropriate handful of required methods or functions, some of which may be inherited from a related pair style (e.g., lj/cut).

At the core, LAMMPS is parallelized using a spatial decomposition with MPI. Therefore, each MPI task stores data only for the *local* atoms within the spatial subdomain and for *ghost* atoms that are in a neighboring subdomain but within the cutoff of the local atoms. At each timestep, each MPI task synchronizes ghost atom data using point-to-point communication with six neighbor MPI tasks. When thermodynamic output is required and for simulations sampled from NPT or NVT ensembles, collective communications are used for calculation of global statistics. For simulations using P^3M calculations, the 3D-FFTs involved require all-to-all communications between tasks in 2D slabs of the simulation box.

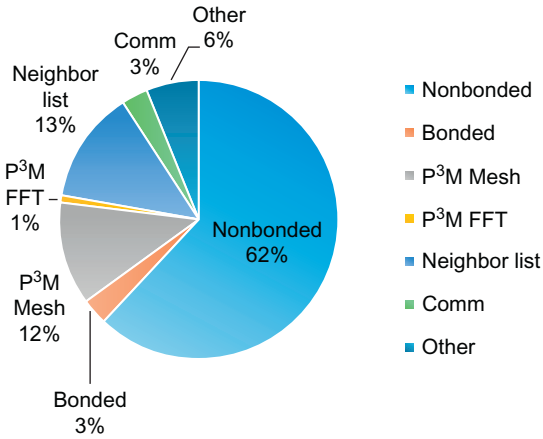


FIG. 20.2

Simulation profile in LAMMPS for the rhodopsin benchmark scaled to 256 K atoms on an Intel Xeon E5-2697v2 processor-based dual-socket node with 48 MPI tasks.

Additional *hybrid* parallelization options are available in packages that combine shared-memory parallelization with MPI using OpenMP or CUDA/OpenCL. Although most of the computations in LAMMPS support hybrid parallelism with OpenMP, flat MPI typically performs best for a variety of reasons that will not be enumerated here. The profile for simulation of a protein benchmark in the NPT ensemble using the CHARMM force-field with long-range electrostatics is shown in Fig. 20.2.

KNIGHTS LANDING PROCESSORS

Knights Landing processors have significant architectural improvements that are relevant to MD. Knights Landing processors are the first to include support for the AVX-512 instruction set. AVX-512 instructions allow mask registers (eight available on Knights Landing) to be used for conditional execution, and the instructions support fault-suppression for memory operations that are masked. In general, this makes it much easier for the compiler to generate efficient vector code for loops with branches (for example, the cutoff check in MD). Additionally, gather and scatter operations are supported for vectorizing loops with random memory access that is common in MD.

Knights Landing also supports the AVX-512ER extensions for fast exponential and reciprocal approximations with single or double precision operands. The AVX-512ER instructions can only dispatch on one of the vector ports. This can limit the instructions issued per cycle compared to most floating-point math that can dispatch on both vector ports. However, the instructions can provide significant improvements for calculations of the terms in some MD force fields as described below.

Other important additions in AVX-512 relevant for random memory access are the instructions supported by the AVX-512CD extension. AVX-512CD is a set of instructions that support conflict-safe random-access accumulation in memory. For example, to calculate the energy due to bonds, a loop over each pair of atoms involved in every bond is required. When vectorized, this could result in simultaneous updates to the force for the same atom because one atom can be bonded to multiple other atoms. AVX-512CD supports detection of identical elements in different data lanes of a vector register. For Knights Landing, this detection requires only 2-cycles and can be used to detect if different data lanes have the same index for an atom. In this case, the values in the lanes corresponding to the same atom index can be accumulated before the memory update in order to produce correct results. As opposed to using atomic operations to prevent memory conflicts, this is advantageous in that it allows for vectorized memory updates that are deterministic (for deterministic operations, the order of operations is identical from run to run allowing for reproducible results and potentially reduced effort for debugging and regression testing).

The Knights Landing architecture includes on-package MCDRAM memory that can achieve much higher bandwidth than traditional DDR memory with slightly higher idle memory access latency. Almost 5x read + write memory bandwidth performance can be achieved for data stored in MCDRAM when compared to DDR4 memory. Although MD software typically has low mean bandwidth requirements, bursts in memory bandwidth needs for MPI communications and routines that loop over contiguous data can result in decreased performance if many cores access data from DDR4. With a typical 16GB capacity, the MCDRAM can be configured as a high-bandwidth cache for DDR memory, or as addressable memory exposed to the operating system as a separate NUMA (nonuniform memory access) node, or a combination of both. Because increasing the size of MD simulations is often coupled to an exponential increase in the number of samples required for meaningful results, time-to-solution is very important and therefore it will most often be desirable to scale simulation runs to a small number of atoms per node. In these cases, MCDRAM can be used for all data without any code modification by using *numactl* or other operating system commands to run a simulation entirely within the MCDRAM NUMA nodes.

The mesh interconnect between cores, Quadrant support, out-of-order buffers, and improved scheduling for hyper-threads can improve MD performance. These improvements benefit MD codes through reduced or amortized memory latencies that otherwise would affect performance from random access of atom data.

One of the most important improvements in the Knights Landing architecture for scalable HPC workloads is the potential for greatly increased electrical power efficiency. Electrical power has become a significant bottleneck as supercomputers approach exascale capabilities. With a single precision peak FLOP rate that is over 3x Intel Xeon E5-2697v3 (Haswell) processor-based nodes but only about 70% of the typical max power, Knights Landing processors offer the potential not only for increased per-node performance, but also for scaling to more nodes within the same power envelope enabling dramatic performance improvements for the same power cost.

We now describe optimizations to LAMMPS to improve performance on modern Intel architectures. In general, the optimizations are not specific to a single processor architecture and should improve performance in general. Some optimizations involving conflict detection are only applicable going forward, for Knights Landing and future Intel architectures supporting AVX-512CD.

LAMMPS OPTIMIZATIONS

We evaluated the code for optimization opportunities for recent Intel Xeon processors and Knights Landing processors. The most important optimizations are summarized in the following subsections. Our early optimizations have focused on the neighbor-list build and nonbonded calculations as these typically dominate the simulation profile and are used by almost all simulations performed in LAMMPS. The optimizations described here are available in LAMMPS as an optional package. This approach can give scientists access to improved performance now, while still allowing the developers to experiment with code modernization strategies to improve future performance and converge on models and algorithms that will perform best and can eventually be adopted as the default in LAMMPS.

DATA ALIGNMENT

Aligning data allocations to 64B boundaries can be important for several reasons. First, because the cache-line sizes on Intel Xeon processors and Knights Landing are also 64B, this can help to prevent false sharing for per-thread allocations. False sharing occurs when threads with affinity for different local caches modify different variables that are stored on the same cache line. Although the variables are not actually being shared by the threads (true sharing), modification of the cache line can require memory updates to maintain cache coherency resulting in a performance decrease. Secondly, alignment to the cache line size can also improve SIMD performance for vectorized loops since aligned accesses provide the fastest path through the memory hierarchy to the registers and core vector processing units (VPUs). In some cases, the compiler will generate separate code paths to handle the first and last iterations of a vectorized loop where not all data on a cache line is used for the calculation. For example, a loop to sum two contiguous arrays in memory requires loading the two source cache lines into registers, adding the results in the registers, and storing the register containing the result to memory. For most iterations of the vectorized loop, all data on the cache lines can be used without masking or permutation. However, for the first and last iteration, additional instructions can be required to prevent memory operations for data in the cache lines that is not relevant to the loop calculation. These additional code paths are called the *peel* (for special handling of data at the beginning of the loop) and *remainder* (for handling data at the end of the loop) code. Aligning arrays to the cache-line size can prevent

```
// Compile-time error if alignment not expected
#if (LAMMPS_MEMALIGN != 64)
#error Please set -DLAMMPS_MEMALIGN=64 in CCFLAGS
#endif

// LAMMPS routine for malloc
void lmp_malloc(size_t nbytes, const char *name)
{
  if (nbytes == 0) return NULL;

#if defined(LAMMPS_MEMALIGN)
  void *ptr;
  int retval = posix_memalign(&ptr, LAMMPS_MEMALIGN,
nbytes);
  if (retval) ptr = NULL;
#else
  void *ptr = malloc(nbytes);
#endif
  if (ptr == NULL) {
    // Exit with error message based inc. name
  }
  return ptr;
}

// Align array to 64 bytes
double myarray[1024]
__attribute__((aligned(LAMMPS_MEMALIGN)));
```

FIG. 20.3

Code listing—Examples for aligning memory allocations.

execution of peel code resulting in a measureable increase in performance. Although gather and scatter operations are not necessarily sensitive to alignment when using the gather/scatter instructions, the compiler may choose to use alternate sequences when gathering multiple elements that are adjacent in memory (e.g., the x, y, and z coordinates for an atom position). These alternate sequences can be faster for aligned data.

In C++, dynamic allocations on the heap can be aligned by using `_mm_malloc()` and `_mm_free()` functions or the `posix_memalign()` function. For allocations on the stack, alignment is achieved using the qualifier `__attribute__((aligned(n)))`. Because LAMMPS includes the capability to align data with `posix_memalign()`, enforcing alignment in our optimization package was achieved simply by defining a preprocessor symbol and adding a compile time check in the package to ensure correct alignment (see Fig. 20.3). Also, alignment qualifiers were added to a few important arrays allocated on the stack.

DATA TYPES AND LAYOUT

For particle simulation, there is an option for storing three-dimensional positions, forces, torques, etc., as an array of structures (AoS) or as a structure of arrays (SoA). In the former, the x, y, and z coordinates for a particle position are contiguous in memory such as x_1,y_1,z_1,z_2,y_2,z_2,x_3,y_3,z_3, etc. In the latter, the x positions for all

particles are contiguous in memory such as x_1,x_2,x_3,x_4, etc. Similarly, the y positions and the z positions are contiguous in memory. The optimal layout will depend on the data access patterns and how the vector calculations are performed. As discussed in earlier chapters on vectorization, SIMD performance will be best when the elements for each data lane are contiguous in memory with the first element aligned to the vector width. Therefore, SoA will perform best for a unit-stride loop over particles where the x positions for every particle are needed for a vector calculation. For the neighbor algorithms presented here, however, particles are accessed indirectly within the loop. This results in random access of particle positions, and therefore, SoA will not provide a benefit for loading a vector register in most cases. Since the x, y, and z coordinates are all needed but randomly accessed, SoA can perform significantly worse; the coordinate data will be in three separate array lists and therefore on three cache lines instead of one.

For LAMMPS, we continue to store data in AoS for cache-efficient random access but modify the structure to be {x, y, z, *type*} rather than {x, y, z}. As discussed below for the code listing in Fig. 20.5, this is because the atom type is also necessary for the nonbonded loops. Because a cache line is 64B, the latter format can result in the position data for a particle to be split across two cache lines and therefore result in reading up to three cache lines for a single particle. For the former, the maximum is 1 cache line (Fig. 20.4). Force and torque structures are also padded to four vectors with the fourth element used to store per-atom energy and virial terms when required for the simulation (not shown in the code listings). Advanced approaches can use low-level AVX-512 vector intrinsics and possibly different data layouts to improve the performance for loading vector registers further. These approaches are not discussed here, but references are given in *For More Information* at the end of this chapter.

The core LAMMPS routines use double precision (64b per data item) for storing and operating on floating-point numbers. Use of single precision (32b per data item) can improve performance significantly, since (1) data for more atoms fit in cache and on a page and therefore average random-access latencies are reduced, and (2) twice

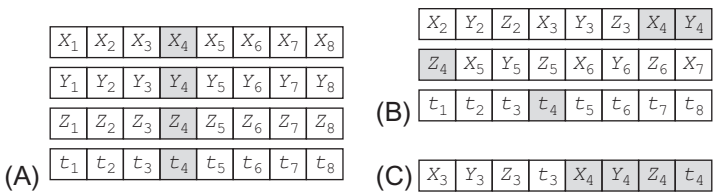


FIG. 20.4

Storage of atom data (x, y, z position and type t) on cache lines for (A) SoA, (B) AoS with three elements, (C) AoS with four elements. Because data access is random for many routines, SoA does not improve performance since gather operations are still required and the data for atom 4 is on four cache lines. For aligned AoS with four elements, at most one cache line is required. This data layout also allows the compiler more flexibility in generating potentially faster sequences for random access of adjacent elements that do not use gather instructions.

as many calculations can be performed concurrently on the vector units. However, using single precision can produce less accurate results when compared to double precision for the same parameterization of a simulation. Therefore, in addition to supporting single and double precision in the Intel package for LAMMPS, mixed precision is also supported. Single precision is generally accurate enough for individual inner loop calculations but when accumulating results of many calculations, double precision is important. So, for mixed precision, particle positions, quaternions, charges, etc., are stored in single precision, and most neighbor list and nonbonded calculations are performed in single precision. All accumulation is performed in double precision, and all variables storing accumulated values (such as forces, torques, energies, and virials) are double precision striking a good balance between accuracy and performance.

All three modes {single, mixed, double} are supported in a single code path using C++ templates and the selection can be made at runtime. In order to allow for incremental optimization to LAMMPS while maintaining full compatibly with all of its options and features, the data is repacked (and optionally recast) every timestep. This adds some overhead, but typically only a small percentage of the total simulation time.

VECTORIZATION

The vector widths on Knight Landing are 512 bits (AVX-512) allowing for up to 16 (512/32) simultaneous single precision floating-point operations per vector operation and up to 8 (512/64) double precision operations. Therefore, using scalar (single operation), execution can leave a lot of performance on the table. Clear evidence of the performance benefit for MD is shown in the Gromacs software that was developed with careful attention to vector performance. Although there are significant improvements in autovectorization (see Chapter 9 on vectorization) for each version of Intel compiler products (discussed later in this chapter), code complexity may require the programmer to use compiler directives to improve performance or to instruct the compiler that it is safe to vectorize a loop. In the case of the MD short-range loop, for example, the compiler does not have the domain-specific information to determine that there are in fact no data dependencies because a particle does not appear in the neighbor list twice. Additionally, it is possible to inform the compiler that arrays in the vector loop are aligned to the vector width, so it will generate optimal code for aligned memory accesses. There are multiple approaches for the programmer to inform the compiler including `#pragma simd` and directives added to the OpenMP 4.0 standard. In Fig. 20.5, a simplified example to calculate the Lennard-Jones potential is shown with pragmas to inform the compiler that (1) the data is aligned, (2) the loop is safe to vectorize, and (3) the variables storing the force, energy, and virial terms should be summed into a single variable at the end of the vector loop.

For simulations that require small cutoffs (small loop trip counts based on the number of neighbors) and also for potential energy models that are very complex

```
#pragma vector aligned
#pragma simd reduction(+:fxtmp, fytmp, fztmp)
for (int jj = 0; jj < jnum; jj++) {
  flt_t forcelj, evdwl;
  forcelj = evdwl = (flt_t)0.0;

  const int j = jlist[jj];
  const flt_t delx = xtmp - x[j].x;
  const flt_t dely = ytmp - x[j].y;
  const flt_t delz = ztmp - x[j].z;
  const int jtype = x[j].w;
  const flt_t rsq = delx*delx + dely*dely + delz*delz;

  if (rsq < cutoff_squared(itype,jtype)) {
    flt_t r2inv = 1.0 / rsq;
    flt_t r6inv = r2inv * r2inv * r2inv;
    forcelj = r6inv *
         (ljc12oi[jtype].lj1*r6inv - ljc12oi[jtype].lj2);
    flt_t fpair = factor_lj * forcelj * r2inv;

    fxtmp += delx * fpair;
    fytmp += dely * fpair;
    fztmp += delz * fpair;
    f[j].x -= delx * fpair;
    f[j].y -= dely * fpair;
    f[j].z -= delz * fpair;

    if (EVFLAG) {
      if (EFLAG) {
        evdwl = r6inv *
            (lj34i[jtype].lj3 * r6inv-lj34i[jtype].lj4) -
            ljc12oi[jtype].offset;
        sevdwl += evdwl;
      }
      IP_PRE_ev_tally_nbor(vflag, ev_pre, fpair,
                           delx, dely, delz);
    }
  } // if rsq
} // for jj
```

FIG. 20.5

Code listing—Simplified loop over neighbors for short-range calculations with the Lennard-Jones potential. flt_t is a template floating-point type that depends on the precision mode. For mixed precision, blue code is single precision and red is double precision. EVFLAG and EFLAG are template arguments to eliminate branches on timesteps when energy and pressure are not required.

with long vector regions, the execution of *loop remainder* code can be inefficient on some processors. Since including a remainder breaks the optimized flow of continuous loop execution and memory accesses, it can limit performance. For example, if there are 36 neighbors using single precision with 16 data lanes for the vector unit, a vectorized loop can use all 16 data lines twice with only 4 data lanes required for the last iteration. In order to prevent execution of remainder code in LAMMPS, a “dummy” atom is created that can never be within the cutoff of any atom in the simulation box. The neighbor lists are then padded so that the scalar trip count is always a multiple of the vector width.

As can be seen in the listing in Fig. 20.5, some potential energy equations in MD do not require transcendentals or long-latency instructions such as square root. This is not the case for simulations that use Ewald methods for long-range electrostatics, however. For the standard Ewald sum, the real-space calculation for the energy computed over neighbors (often referred to as *direct summation*) is given by,

$$U_{\mathrm{R}} = \sum_i \sum_{j>i} \frac{C_1 q_i q_j}{r_{ij}} \left(\mathrm{erfc}\left(C_2 r_{ij}\right) - s_{\mathrm{b}}\right),$$

where C is a constant, q is the partial charge of the atom, r is the distance between the atoms, and s_{b} is a scaling factor applied when atoms i and j are bonded or separated by a small number of bonds. This results in a force term given by,

$$U'_{\mathrm{R}} = \sum_i \sum_{j>i} \frac{C_1 q_i q_j}{r^2} \left(\mathrm{erfc}(C_2 r) + C_3 C_2 r \exp\left(-C_2^2 r^2\right) - s_{\mathrm{b}}\right).$$

This calculation can be expensive to compute as it involves division, square root, exponential, and the complementary error function for every pair. For IEEE arithmetic, the latencies for divide and square root are 38 cycles on Knights Landing; for exp() and erfc(), calls to a math library are required.

For classical MD, however, full precision is not always necessary due to limitations in numerical accuracy that result from finite time-steps, discretization of the Ewald summation, use of cutoffs, etc. Therefore, most legacy MD applications switched to using linear table interpolation to calculate the Ewald term. This provides significant speedups for scalar calculations but can be difficult to vectorize efficiently. If table interpolation is used with the squared distance as the index, LAMMPS requires four tables (due to a distance lookup and the need to scale bonded interactions) to compute the force and each table requires 4096 elements to achieve acceptable accuracy. If energy calculation is needed on a given timestep, an additional two tables are needed. Although bitwise operations are used for very efficient indexing into the tables and the tables can be combined into AoS format to improve cache efficiency, in double precision, this requires 128–192 KB to store the tables (remember that the L1 cache on Knights Landing is 32 KB). With random access for atom data and gather operations required to do table interpolations for atom pairs with different distances, it can be difficult to get good performance with this approach.

A better option that we have used in LAMMPS is to vectorize the explicit math for the Ewald terms. For the complementary error function, an approximation developed by Abramowitz and Stegun is used where,

$$\mathrm{erf}(w) \approx 1 - \left(a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5\right) \exp\left(-w^2\right),$$

a represents a constant and $t = 1/(1+pw)$ with $p = 0.3275911$. This approximation is convenient because the exponential term is already required to calculate the force. The approximation is bounded by a maximum error of 1.5×10^{-7}. Fig. 20.6 shows the calculation of the Ewald term ($U'_{\mathrm{R}} \cdot r$) in LAMMPS. When compiled with the "`-fp-model fast=2`" compiler option, the Ewald term is calculated with a sequence

```
const float A1 =  0.254829592;
const float A2 = -0.284496736;
const float A3 =  1.421413741;
const float A4 = -1.453152027;
const float A5 =  1.061405429;
const float C_3 = 1.12837917;
const float INV_EWALD_P = 1.0 / 0.3275911;

const float r = sqrt(rsq);
const float grij = C_2 * r;
const float expm2 = exp(-grij * grij);
const float t = INV_EWALD_P / (INV_EWALD_P + grij);
const float erfc = t * (A1+t*(A2+t*(A3+t*(A4+t*A5)))) *
                   expm2;
const float prefactor = C_1 * q_1 * q_2 / r;
forcecoul = prefactor * (erfc + C_3 * C_2 * expm2);
if (EFLAG) ecoul = prefactor * erfc;
if (sbindex) {
    const flt_t adjust = ((float)1.0 - s_b)*prefactor;
    forcecoul -= adjust;
    if (EFLAG) ecoul -= adjust;
}
```

FIG. 20.6

Code listing—Calculation of Ewald terms for energy (ecoul = U_R) and force (forcecoul = $U'_R \cdot r$) using an elemental approximation to the complementary error function. *q_1* and *q_2* are the charges on the two atoms in the *i*, *j* pair and *rsq* is the squared distance between the atoms.

that requires only four 8-cycle AVX-512ER instructions (1 reciprocal, 1 reciprocal square root, and 2 base two exponents). When compared to a SoA vectorized version of the linear interpolation on Knights Landing, this results in a 28% reduction in the time required to calculate the *entire* short-range force loop including van der Waals interactions.

Three-body potentials can be more difficult to vectorize efficiently due to multiple loops for two neighbors in each triplet used for the energy calculation. There are surprisingly many ways to decompose this problem, but a loop structure similar to the baseline code in LAMMPS for a 3-body potential is shown in Fig. 20.7.

Vectorization of the inner-most *kk* loop in Fig. 20.7 is the simplest option and can be done safely because neighbors *j* and *k* will never be identical for a triplet. Unfortunately, the scalar trip count can be very low for some simulation models and will decrease to 1 for the last *j* neighbor evaluated. As written, it is not safe to vectorize the *jj* loop due to the potential for memory conflicts where *k* is the same for different *j* particles. Additionally, there would be vector inefficiency because only half of the *j* neighbors are involved in computation for the 2-body terms in the potential. Vectorization aside, there is also an optimization opportunity resulting from some redundant computation in the 3-body loop such as the call to `cutoff_squared()` for *j* and *k* indices that are repeated multiple times for the same neighbor.

These issues are addressed in the listing in Fig. 20.8. Each triplet is evaluated twice to eliminate the update to the force indexed by *k* and allow for safe vectorization of the *jj* loop. The logic for load balancing (to only evaluate half of the neighbors for the 2-body term) is moved to the neighbor-list build that is modified to use a dual-

```
for (i = 0; i < num_local_atoms; ++i) {
  …
  // two-body interactions, skip half of them
  for (jj = 0; jj < num_nbors; ++jj) {
    j = neighbor(i,jj);
    jtag = tag[j];

    if (rsquared(i,j) > cutoff_squared(i,j)) continue;
    if (load_balance(i,j,itag,jtag)) {
      twobody(i,j,fpair);
      fi += fpair;
      f[j] -= fpair;
    }

    // three-body interactions
    for (kk = jj+1; kk < num_nbors; ++kk) {
      k = neighbor(i,kk);

      if (rsquared(i,k) > cutoff_squared(i,k)) continue;
      threebody(i,j,k,fj,fk);
      fi -= fj + fk;
      f[j] += fj;
      f[k] += fk;
    }
  }
  f[i] += fi;
}
```

FIG. 20.7

Code listing—Loop structure for a 3-body potential in the baseline LAMMPS code. For brevity, separate updates to *x*, *y*, and *z* dimensions are not shown. Energy and virial terms are also removed.

offset build that orders particles that should be evaluated for the 2-body term first (with count *jnumhalf*). Although there is still a branch, the ordering asserts that at most one iteration of the vector loop will have divergence across the data lanes. Some redundant computation is eliminated by adding an initial loop over neighbors for computing distance and any other redundant terms specific to the potential energy model that are used in the 3-body loop. This also serves to eliminate divergence within the vector loop by packing only neighbors within the cutoff into a temporary array. When vectorized, branch divergence will decrease vector efficiency because some lanes will be masked resulting in wasted computations.

The preprocessed terms needed by the 3-body loop are stored in temporary arrays in SoA format for efficient loading to vector registers. This optimization is significant as it reduces the number of gather operations from $O(K^2)$ to $O(K)$ where K is the number of neighbors. When compared to vector loads that require a minimum of 5 cycles with up to 2 instructions per cycle, gather operations require at least 15 cycles with no more than 0.2 instructions per cycle. Likewise, the *kk* loop modification to evaluate each triplet twice reduces the number of scatter operations from $O(K^2)$ to $O(K)$.

For brevity, we do not show atom type parameters in the code listing. These constants require additional gather operations within the force loop. We store the atom

```
for (i = 0; i < num_local_atoms; ++i) {
  …
  for (jj = 0; jj < num_nbors; ++jj) {
    j = neighbor(i,jj);
    if (rsquared(i,j) > cutoff_squared(i,j)) continue;
    ccache_soa(i,j);
    ejnum++;
    if (jj < jnumhalf) ejnumhalf++;
  }
  ccache_pad();

  #pragma vector aligned
  #pragma simd reduction(+:fi)
  for (jj = 0; jj < ejnum_pad; ++jj) {
    j = cneighbor(jj);
    if (jj >= ejnumhalf) {
      twobody(i,j,jj,fpair);
      fi += fpair;
      fjtmp -= fpair;
    }

    // three-body interactions
    for (kk = 0; kk < ejnum; ++kk) {
      k = cneighbor(kk);
      threebody(i,j,k,jj,kk,fj,fk);
      fi -= fj;
      fjtmp += fj;
    }
    f[j]+=fjtmp;
  }
  f[i] += fi;
}
```

FIG. 20.8

Code listing—Optimized loop structure from Fig. 20.7 to improve vector performance. *ccache_soa*() represents calculations to pack temporary arrays to eliminate redundant computation and a temporary neighbor list including only those particles within the cutoff (*cneighbor*). *ccache_pad*() pads the temporary arrays to prevent remainder code execution with *ejnum_pad* forced to be a multiple of the vector width.

types in temporary arrays to prevent redundant gathers, but it does not make sense to do this for constants that depend on pairs or triplets of atoms as the gathers are not redundant. For the Stillinger-Weber 3-body potential energy model, it is common to perform simulations where only a single atom type is used (at least for nonbonded 3-body interactions). Therefore, we have added an additional optimization that uses an integer template parameter to generate code without type-specific gathers for cases where only a single atom type is used in Stillinger-Weber. This can provide a small but significant (5–10%) improvement in performance.

These modifications can improve simulation performance significantly for 3-body potentials. However, vectorization efficiency for some simulations can still be limited by low trip counts for the *jj* and *kk* loops. For Knights Landing and future processors with AVX-512CD, an alternative approach is to vectorize the outer loop and use conflict detection sequences to safely update the forces. This approach is shown in the code listing in Fig. 20.9.

```
// Vectorization of outer loop over i
for (i = 0; i < num_local_atoms; ++i) {
  …
  for (jj = 0; jj < num_nbors; ++jj) {
    j = neighbor(i,jj);
    if (rsquared(i,j) > cutoff_squared(i,j)) continue;
    ccache_soa(i,j);
    ejnum++;
    if (jj < jnumhalf) ejnumhalf++;
  }

  zero_cforce();
  for (jj = 0; jj < ejnum; ++jj) {
    j = cneighbor(jj);
    if (jj < ejnumhalf) {
      twobody(i,j,jj,fpair);
      fi += fpair;
      fjtmp -= fpair;
    }

    // three-body interactions
    for (kk = jj+1; kk < ejnum; ++kk) {
      k = cneighbor(kk);
      threebody(i,j,k,jj,kk,fj,fk);
      fi -= fj + fk;
      fjtmp += fj;
      cforce[kk] += fk;
    }
    cforce[jj] += fjtmp;
  }

  for (jj = 0; jj < ejnum; ++jj) {
    j = cneighbor(jj);
    conflict_safe_accumulate(j,cforce[jj]);
  }

  f[i] += fi;
}
```

FIG. 20.9

Code listing—Optimized loop structure from Fig. 20.8 to exploit fast conflict detection with AVX-512CD. Redundant computation is eliminated and an additional loop over neighbors is added to prevent redundant noncontiguous memory updates.

In this case, vector inefficiency due to small trip counts is eliminated and the `ccache_pad()` code can be removed. Rather, vector inefficiency will occur due to variance in the number of neighbors (trip count for *jj* and *kk* loops). However, the redundant computation in the loops can be eliminated almost entirely as it is no longer necessary to loop over each triplet twice to avoid memory conflicts. Although this would increase the number of gather/scatter operations for force updates from O(K) to O(K^2) for a naïve approach, we can take advantage of the fact that when $jj = kk$, the index for the neighbor is the same. Therefore, analogous to the precomputations to store distances, etc., in the `ccache_soa()` code, we can add temporary arrays to store the forces for the neighbors of the atoms that are processed simultaneously. Although the `ccache_soa()` requires scatter to store precomputed values when the outer loop is vectorized, keeping the force update gather/scatter at O(K) is important

due to the additional overhead of using AVX-512CD sequences to handle any potential memory conflicts. With this approach, there is no requirement for gather/scatter during the nested force computation loop, and memory conflicts are not a concern. An additional loop over the neighbors is added to perform the conflict-safe force accumulation with gather/scatter operations from the values stored in the small temporary arrays.

As Knights Landing is the first architecture to be released with support for AVX-512CD, compiler vectorization with conflict is currently limited to autovectorization. This is due to the fact that current syntax for explicit vectorization using directives also instructs the compiler that the loops do not have data dependencies preventing vectorization. We hope to see compiler improvements and additions to the OpenMP standard to address this; however, currently, the routines with conflict-safe accumulation are implemented with AVX-512 vector intrinsics (introduced in Chapter 12) with conflict detection in the function `conflict_safe_accumulate`. This also allows for optimizations for adjacent gather/scatter when updating three force elements based on a single atom index. A sample vector intrinsics sequence for the force update is shown in Fig. 20.10 for single precision.

Optimizations for the loops calculating forces due to bonds (U_S, U_A, U_D) are similar, but typically account for a much smaller fraction of the total simulation time. For forces due to bond vibration, a loop over a list of all pairs of atoms is used to update the force terms on the atoms. For angle-bending, there is a loop over triplets of atoms, and for dihedral/improper, the loop is over groups of four atoms. Efficient vectorization of these loops requires use of conflict detection sequences or alternatively, list reordering to guarantee that no two atoms will be accessed in multiple lanes during vector calculations. Initially in LAMMPS, we plan to use conflict detection with vectorization; however, this is not implemented yet as we work to improve compiler capabilities. For the measurements presented here, the optimizations for bonded forces are limited to using the optimized data structures for atom data without vectorization.

Vectorization of the time integration routines is trivial; however, there is one optimization for storage of particle masses that can improve performance. Because the time integration involves division of the three components of the force by the mass, storing the reciprocal of the mass with duplicates for each component of the force can improve performance by eliminating division/reciprocal operations and operations for packing the registers for SIMD arithmetic.

NEIGHBOR LIST

The latency for accessing data from memory is very high. When instructions cannot execute due to dependencies on data that is not yet available from memory, cycles are wasted. Ideally, this latency is not exposed; data is loaded into registers from the data cache unit and prefetching pulls data that will be needed for future instructions into the cache before it is needed. For contiguous memory access and other simple access patterns, this often works quite well and the hardware can prefetch data without software optimization. For random and complex noncontiguous access, it can be

```
// __mmask16 rmask      - mask for data lanes with atoms inside cutoff
// __m512 amx, amy, amz - forces(x,y,z) to be added to force array at j
// force_t *f           - AoS for atom forces (f[j].x, f[j].y, f[j].z)
// __m512i   joffset    - indices for neighbors multiplied by 16 (for
//                        byte offset into f for atom force data)

// Set the index for neighbors outside the cutoff to -1 to avoid
// erroneous conflicts with neighbors inside the cutoff
__m512i jcutoff = _mm512_mask_mov_epi32(_mm512_set1_epi32(-1),rmask,j);

// Find any data lanes with the same value for j that would result
// in memory conflicts.
__m512i cd = _mm512_maskz_conflict_epi32(rmask,jcutoff);

// Mask for data lanes that require reduction of forces
__mmask16 todo_mask = _mm512_test_epi32_mask(cd,_mm512_set1_epi32(-1));

if (todo_mask) {
  // Get the indices to permute forces for reduction of lanes indexing
  // the same atom into a single data lane
  __mm512i lz  = _mm512_lzcnt_epi32(cd);
  __mm512i lid = _mm512_sub_epi32(_mm512_set1_epi32(31),
                                  _mm512_lzcnt_epi32(cd));
  // If more than 2 data lanes have the same neighbor index, require
  // multiple iterations of reduction loop
  while(todo_mask) {
    // Handle multiple conflicts simultaneously, but only 2 with the
    // same j index at a time.
    __mm512i todo_bcast = _mm512_broadcastmw_epi32(todo_mask);
    __mmask16 now_mask = _mm512_mask_testn_epi32_mask(todo_mask, cd,
                                                      todo_bcast);
    // Add the forces from multiple lanes with the same index
    __m512 am_perm;
    am_perm = _mm512_mask_permutevar_epi32(_mm512_undefined_epi32(),
                        now_mask, lid, _mm512_castps_si512(amx));
    amx = _mm512_mask_add_ps(amx, now_mask, amx, am_perm);
    am_perm = _mm512_mask_permutevar_epi32(_mm512_undefined_epi32(),
                        now_mask, lid, _mm512_castps_si512(amy));
    amy = _mm512_mask_add_ps(amy, now_mask, amy, am_perm);
    am_perm = _mm512_mask_permutevar_epi32(_mm512_undefined_epi32(),
                        now_mask, lid, _mm512_castps_si512(amz));
    amz = _mm512_mask_add_ps(amz, now_mask, amz, am_perm);
    todo_mask = _mm512_kxor(todo_mask, now_mask);
  }
}

// Gather old forces, add terms from amx, amy, amz and update in memory
__m512 jfrc;
jfrc = _mm512_mask_i32gather_ps(_mm512_undefined_ps(), rmask, joffset,
                                &(f[0].x), _MM_SCALE_1);
jfrc -= amx;
_mm512_mask_i32scatter_ps(&(f[0].x),rmask, joffset, jfrc, _MM_SCALE_1);
jfrc = _mm512_mask_i32gather_ps(_mm512_undefined_ps(), rmask, joffset,
                                &(f[0].y), _MM_SCALE_1);
jfrc -= amy;
_mm512_mask_i32scatter_ps(&(f[0].y),rmask, joffset, jfrc, _MM_SCALE_1);
jfrc = _mm512_mask_i32gather_ps(_mm512_undefined_ps(), rmask, joffset,
                                &(f[0].z), _MM_SCALE_1);
jfrc -= amz;
_mm512_mask_i32scatter_ps(&(f[0].z), rmask, joffset, jfrc,
                          _MM_SCALE_1);
```

FIG. 20.10

Code listing—Vector intrinsics sequence for conflict-safe gather-modify-scatter force update with AVX-512CD. When using gather/add/scatter for vectorized accumulation of forces, unique indices must be used for the gather/scatter; otherwise only the last value will be written to memory. This sequence accumulates forces from multiple lanes with the same neighbor index into a single data lane so that the last value written to memory will contain the correct result.

important for software developers to make modifications to improve data locality. For many-core architectures without a large last-level cache, data locality optimizations can be even more important. In some cases, software prefetching can be used to reduce effective data access latencies. There are limits to the number of outstanding loads from memory, however, and unless the arithmetic intensity is very high, it is not feasible to prefetch data for 16 data lanes per vector operation if 16 different cache lines might be needed.

The loop over neighbors in the force calculation requires fetch and store of data for atoms that are close in the 3D simulation box. Therefore, data locality can be improved by sorting data in memory such that atom data that is nearby as measured by the physical memory address is also nearby in the 3D simulation box. In this manner, *spatial* locality can be improved because it is more likely in the *jj* loops that access to neighbor data on a cache line will be followed by access to another neighbor's data on the same or a nearby cache line. Likewise, *temporal* locality can be improved because it is more likely that data on the cache lines will be reused in the next iteration of the *i* loop; atom $i+1$ is likely to share many neighbors with atom *i*. Sorting of atom data is performed by binning atoms into 3D cells and then repacking the atom data in memory by walking through the bins, optionally in order determined by a space-filling curve. Therefore, it makes sense to do this along with neighbor-list builds. The capability to periodically sort atom data in memory has been available in LAMMPS for some time; however, on Intel Xeon Phi many-core architectures, we increase the frequency of the sort to be performed every neighbor-list build. Data locality can be more important for performance on these architectures, in part due to the fact that there is not a large L3 cache. Although the MCDRAM on Knights Landing can be configured as a memory-side cache, the latencies for accessing data from MCDRAM cache are not reduced compared to DDR4 or addressable MCDRAM, and therefore the benefit from this configuration is primarily for routines requiring high memory bandwidth.

The first step in the neighbor-list build involves binning the atoms into 3D cells (Fig. 20.1). This is typically a very small fraction of the simulation time. Following binning, for each atom, the neighbor list is built by looping over other atoms in the same 3D cell and surrounding cells within the cutoff and skin distance. For 2-body interactions, only half of the surrounding cells and atoms in the same cell are used due to symmetry from Newton's third law. For 3-body interactions, all surrounding atoms must be considered. For each surrounding atom, the squared distance is compared with the cutoff and skin distance to see if it should be added to the neighbor list. Special checks for exclusion lists (atoms that should not be neighbors of other atoms), neighbors that are bonded, and special handling when cutoffs are greater than half of the simulation box size can be required.

Despite this complexity, neighbor-list builds can vectorize efficiently aside from two complications. The first is that the number of atoms in each cell is not necessarily large compared to the vector width, and in LAMMPS, this number can vary dramatically depending on the type of simulation being performed. Therefore, it is desirable to vectorize over all atoms in surrounding cells and not just a

single cell. Modification to the loop structure can be necessary so that loads for the surrounding atom indices can be loaded into each data lane without concern for cell boundaries. In LAMMPS this is performed by using a temporary array to store atom indices that is filled prior to the vectorized neighbor-list build loop. The second complication is that stores for neighbor atom indices depend on the cutoff check and will not be stored for all lanes. AVX-512 supports compress instructions to handle this efficiently, but it can be difficult for the compiler to use these instructions efficiently in large complex loops. Currently in LAMMPS, we do not compress neighbors that are within the cutoff inside of the explicitly vectorized neighbor list, but rather change the index of atoms outside the cutoff to be greater than the number of atoms processed by the MPI task. A final loop to perform the compression is then added that is much simpler and can potentially be autovectorized by the compiler.

In order to improve data locality and performance, we use a dual offset neighbor-list build for each atom so that two lists are built that are later coalesced into a single list. For 2-body potentials, the second list is used to store ghost atoms. This is performed to improve data locality because ghost atoms are always at the end of the atom data arrays, regardless of their location within the simulation box. As discussed earlier, for 3-body potentials, the second list is used to store neighbors that are required for the 3-body interactions, but not the 2-body interactions (remember that the 3-body potential energy model also includes a 2-body term).

We note that there are approaches that trade vector inefficiency for improved data locality and efficient register loading in MD. These approaches handle atom neighbors differently; one example is the work by Páll et al. referenced in *For More Information* at the end of this chapter.

LONG-RANGE ELECTROSTATICS

Optimizations to enable vectorization and improve parallel efficiency for MD can be very important for performance. These will be provided in future publications. For the performance measurements herein, the standard particle mesh routines in LAMMPS for reciprocal space calculations are used.

MPI AND OpenMP PARALLELIZATION

LAMMPS development began in the mid-1990s, and the software has been developed primarily for MPI parallelization. As the number of cores sharing the memory subsystem has increased, more developers have put effort into exploiting hybrid parallelism with the so-called MPI+X programming model (where X represents a shared-memory parallelization using OpenMP, POSIX threads, etc.). Although LAMMPS supports hybrid parallelism with OpenMP for most important routines and there is active development to improve shared-memory performance, one

MPI task per core typically performs best at the time of writing. Using one MPI task per tile (two cores on Knights Landing) with OpenMP parallelization currently decreases performance by 10–15% on Knights Landing.

Using high MPI task counts is not necessarily detrimental to performance on many-core architectures, however; there are inherent advantages to using higher task counts that can require some effort to realize with shared-memory parallelization. First, the per-core memory bandwidth is limited and an increase in the number of MPI tasks sending smaller messages can decrease the overall message passing time. Second, when scaling to large node counts on supercomputers, there is inevitable variance in calculation times due to small imbalances in the amount of work, O/S noise, etc. High MPI task counts naturally avoid an issue with bulk synchronization on the node such that some MPI tasks can begin work as soon as messages are available. Most importantly, however, there is usually an inherent improvement in data locality with increasing MPI tasks for scalable HPC codes, and this is particularly beneficial with most MD codes. When using shared-memory parallelization, it should not be assumed that all cores accessing a larger shared working set will be faster; it can be the case that this approach actually increases memory access times resulting in an overall decrease in performance. Therefore, it can be the case that the best shared-memory implementations use a parallel decomposition that is very similar to the MPI parallelization. This can be difficult to implement in an incremental manner, however, and to achieve good scalability almost every routine must be parallelized efficiently—including data packing and movement for internode communications. Because most MPI implementations use shared-memory optimizations for intranode communications, the performance benefits of MPI+X hybrid parallelization might not be as dramatic as one might expect assuming that (1) the shared-memory algorithm is the same, (2) the processor supports the memory bandwidth needed for intranode communications, (3) memory usage scales well with MPI tasks for the software, and (4) the internode interconnect is balanced with the core count, supporting higher message rates and communication buffer management for higher task counts.

There are several optimizations that can help performance when using high MPI task counts on each node. For spatial decompositions, it is a best practice to factor the simulation box into subdomains in a manner that minimizes the surface-to-volume ratio of the subdomain that is owned by each MPI task. This decreases the amount of ghost data that is shared between different MPI tasks. Likewise, the MPI tasks on each *node* should be mapped to the simulation box in a manner that minimizes the surface-to-volume ratio of the subdomain that is owned by all MPI tasks on the *node*. If this is done correctly, the internode communications volume will not necessarily change dramatically with increasing MPI task counts because the surface area of the borders between different nodes does not change significantly (the communications with neighboring subdomains are ordered so that communication is only required with six neighbor MPI tasks and never with neighbor MPI tasks at the corners). This is illustrated in Fig. 20.11. The MPI standard includes routines for task mapping for Cartesian topologies that can be optimized for specific processor architectures and fabrics; however, these optimizations are almost never

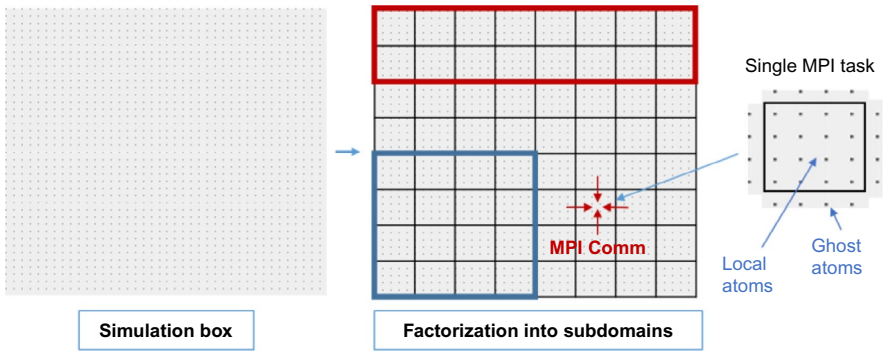


FIG. 20.11

2D illustration of a spatial decomposition. The simulation box is factored into 64 subdomains (*small boxes*) for 64 MPI tasks. The 8×8 factorization shown minimizes the ratio of ghost atoms to local atoms (imagine a 1×64 factorization). When mapping 16 MPI tasks on a node to the simulation box, it is preferable to map them to subdomains in the region bound by the square (4×4) box rather than the region bound by the *rectangular (8×2) box*. In the former, most of the MPI communications at borders of the subdomains occurs within a node using shared-memory optimizations.

implemented. Therefore, the developer might have to make modifications to adjust the task mapping in order to minimize internode communications. This does not necessarily require significant code modification; in LAMMPS this requires modification of a single subroutine. The description of the approach used is given in *Procedia Computer Science* as referenced in *For More Information* at the end of this chapter.

Another important consideration for legacy MPI software is hyper-threading. Hyper-threading allows the core to rapidly switch to another process or thread when it is stalled, for example due to a dependence on data that is not yet available from memory. Therefore, most MD codes see a performance benefit from using hyper-threading. Oversubscribing cores with MPI tasks for hyper-threading can improve performance for single node runs; however, we do not recommend this for scaling on clusters. Knights Landing supports up to four hyper-threads per core, but it is important to note that the hyper-threads share resources that are partitioned among the awake threads. Therefore, latency sensitive routines might see improved performance with hyper-threading, while other routines might decrease in performance. For the LAMMPS runs presented here, we address this by using OpenMP only for latency sensitive routines on hyper-threads. We set the time before sleep for OpenMP threads to zero in this approach for optimal use of shared resources. This can be done by setting the `KMP_BLOCKTIME` environment variable or by using the `kmp_set_blocktime()` function.

Although we can achieve acceptable performance using an MPI task for each core on Knights Landing, the approach is not ideal for two reasons. In a spatial decomposition, the space is divided such that each MPI task has a unique set of local atoms. However, the position, type, charge, and other data for ghost atoms is a duplicate,

and therefore shared-memory parallelization has the potential to improve cache efficiency. This does not necessarily extend to ghost data for forces, per-atom energies and virials, torques, mesh points for long-range, etc.; for the general case, better performance is obtained with duplication from data privatization in order to avoid memory conflicts (although there are exceptions). The second issue is that the 3D-FFT required for the Poisson solve does not scale as well as the other routines in MD; the degree of concurrency that can be efficiently exploited is much lower for this routine when compared to neighbor-list builds and short-range force calculation. Additionally, all-to-all communication is required. With a MPI+X implementation, it is usually trivial to decrease the number of threads (possibly to one) for a routine where parallel overheads dominate the calculation. This is also possible in the MPI-3 standard (MPI+MPI for shared-memory parallelization), but it is more involved and is complicated by the lack of specification of a memory model in the MPI-3 standard. For the 3D-FFT in LAMMPS, there are several approaches to improve performance for scaling on clusters and supercomputers; these will be described in a future publication.

PERFORMANCE RESULTS

The optimizations described here are available as part of the Intel package that is included with LAMMPS. The link to the latest distribution of LAMMPS is available in *For More Information* at the end of this chapter. We have evaluated several workloads for performance including protein, water, and liquid crystal benchmarks. Additionally, we present performance results from production simulations used to study molecular alignment in organic solar cells and the thermodynamic and transport properties of complex hydrocarbon mixtures. We provide the best performance that can be obtained in LAMMPS without the Intel package (referred to as the baseline results) where only double precision is supported. For the Intel package of LAMMPS, with the optimizations described here, mixed precision is used.

SYSTEM, BUILD, AND RUN CONFIGURATIONS

Performance numbers were measured on Intel Xeon E5-2697v3 processors (codenamed Haswell) and on a pre-production Knights Landing processor. We compare dual socket E5-2697v3 to single socket Knights Landing. The E5-2697v3 processors have 28 cores (56 hardware threads) running at 2.6 GHz and a 290 W TDP. The Knights Landing used contains 68 cores (272 hardware threads) running at 1.4 GHz with a 215 W TDP and 16 GB MCDRAM. The Intel Xeon processor-based system had 64 GB (8x8) 2133 MHz DDR4 memory and runs Red Hat Enterprise Linux Server release 6.6. The Knights Landing system runs SUSE 3.12.28 patched with AVX-512 support. All Knights Landing runs are performed entirely in MCDRAM configured in quadrant/flat mode.

The publically available "15 Feb 2016" version of LAMMPS contains all optimizations presented here. It was built with the C++ compiler, MPI library (see

Chapter 15), and Intel Math Kernel Library (MKL, see Chapter 13) included in the 2016 Intel Parallel Studio version 1.056. The compiler flags used are "`-g -O2 -fno-alias -ansi-alias -restrict -DLAMMPS_MEMALIGN=64 -override-limits -fp-model fast=2 -no-prec-div -openmp -no-offload -static-intel -DLAMMPS_GZIP -DLAMMPS_JPEG -DFFT_MKL -DFFT_SINGLE`."

WORKLOADS

The simulation rate and impact from performance improvements can vary drastically in LAMMPS. Arithmetic intensity of the potential energy model, average neighbor-list size, and even the simulation ensemble influence performance. Here, we present early results for several simulations run in LAMMPS. We note that the simulations for organic photovoltaics, hydrocarbon mixtures, and the rhodopsin protein all use P^3M for long-range electrostatics. These routines have not yet been modified for efficient vectorization or to use the mixed precision data structures presented here. We expect significant improvements from these optimizations and modest improvements from vectorization of bonded terms with conflict detection. Nonetheless, the initial performance numbers are very encouraging. The results for all workloads are summarized in Fig. 20.12.

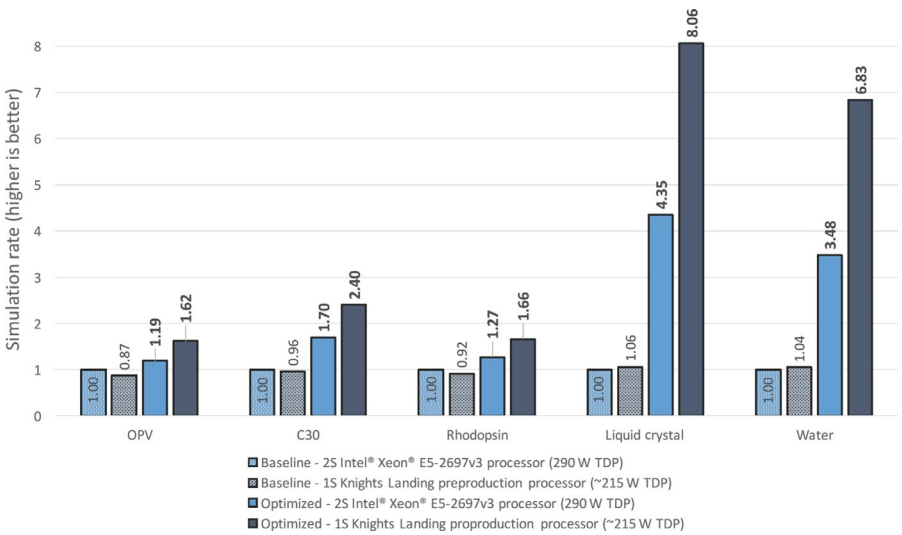


FIG. 20.12

Summary of speedups for the LAMMPS workloads relative to the Intel Xeon processor-based system running the baseline LAMMPS code.

ORGANIC PHOTOVOLTAIC MOLECULES

The most common devices for converting sunlight into energy use inorganic molecules. Recently, research into alternative solar cell designs using organic molecules (organic photovoltaics or OPV) has become popular due to their cheap cost and desirable mechanical properties. One popular design for organic solar cells uses a combination of donor molecules (that absorb the sunlight) and acceptor molecules (that facilitate the conversion of the absorbed energy into electricity). The morphology and molecular alignment that characterize these blends is known to have a significant impact on the efficiency of the device. Therefore, researchers at Oak Ridge National Laboratory have been performing simulations to predict how changes to the molecular blend might improve the efficiency of organic devices. The simulation used here is a shortened run from production simulations used to study the sharpness of donor/acceptor interfaces with detail at the atom level.

In these simulations, the Generalized Amber Force-Field (GAFF) is used to simulate blends with 1,777,520 atoms. For the benchmark runs, the simulation was run for 1.5 picoseconds (ps) in an isothermal-isobaric (NPT) ensemble utilizing a Nose-Hoover thermostat and barostat (423 K, 1 atm) with an integration timestep of 1 femtoseconds (fs) where the coordinates of the atoms and image snapshots are written to output files every 0.5 ps. A 10-timestep warmup run was performed before timing for performance. The temperature damping parameter for the Nose-Hoover thermostat was set to 100 fs and the pressure damping parameter for the barostat was set to 1000 fs. For the OPV simulation, the optimizations presented gave a 19% improvement in the simulation rate when compared to the baseline code on the Intel Xeon processor-based system. For Knights Landing, the performance improvement was more significant at 1.85×. This simulation rate was 1.36× the result obtained on the Intel Xeon processor-based system. The benchmark runs included all of the statistics and I/O used in production simulations.

HYDROCARBON MIXTURES

For the second case, we chose a workload being used to study thermodynamic, transport, and rheological properties of hydrocarbon mixtures (typically 30-carbon atoms long) used as lubricants (C30 workload). This simulation uses 165,600 atoms modeled using the CHARMM force field in a periodic box with an inner cutoff of 11 Å, an outer cutoff of 13 Å, and a neighbor-list skin distance of 2 Å. Long range electrostatics were calculated using particle-particle particle-mesh with an accuracy of 1×10^{-4}. The simulation is sampled from the isothermal-isobaric (NPT) ensemble at 313 K in a triclinic box that is deformed throughout the simulation. The performance measurements were for 400 timesteps with I/O including a single configuration dump. Again, a 10-step warm-up run was used before measurements. The optimizations presented here resulted in a speedup of 1.7× on the Intel Xeon processor-based system and 2.5× on the Knights Landing system. The Knights Landing was 1.42× faster for this simulation.

RHODOPSIN PROTEIN IN SOLVATED LIPID BILAYER

The rhodopsin protein benchmark is a scalable benchmark provided with LAMMPS. This is an all-atom benchmark simulating the rhodopsin protein in a solvated lipid bilayer using the CHARMM force field, particle-particle particle-mesh long-range Coulombics, and SHAKE constraints. The benchmark is based on simulations by Crozier et al (see *For More Information* at the end of this chapter). The model contains counter-ions and a reduced amount of water to make a 32,000 atom system. The benchmark was scaled to 512 K atoms for the performance measurements here. An inner cutoff of 8 Å and an outer cutoff of 10 Å are used for short-range force calculations. The simulations are performed using the isothermal-isobaric (NPT) ensemble with a timestep of 2.0 fs. The benchmark was run for 100 timesteps following a 10 timestep warm-up run. The speedups from our optimizations were 1.27 × on the Intel Xeon processor-based system and 1.82 × on the Knights Landing system. Compared to the Intel Xeon processor-based system, the Knights Landing performance was 31% faster.

COARSE GRAIN LIQUID CRYSTAL SIMULATION

We have also evaluated a liquid crystal benchmark based on previous materials simulations using the Gay-Berne potential. Each liquid-crystal molecule is treated as a single ellipsoidal particle. The computational cost per atom (or particle in the coarse-grained case) of many potentials developed for materials science has grown significantly over time, and these models have perhaps the greatest potential improvements from vectorization. In this benchmark, the simulation is comprised by 524 K biaxial ellipsoidal liquid crystal mesogens with an aspect ratio of 2:1.5:1 and a mass of 1.5 (reduced units). The simulation is sampled from the microcanonical ensemble for 50 timesteps, following a 10 timestep warm-up. The short-range cutoff is 4.0 with a neighbor-list skin of 0.8. This benchmark simulation is also available in the LAMMPS distribution. The optimizations presented here resulted in a 4.35 × speedup on Intel Xeon processor-based system and 7.62 × on Knights Landing. The Knights Landing performance is impressive for this simulation, running 1.85 × faster than the Intel Xeon processor-based system.

COARSE-GRAIN WATER SIMULATION

There is great interest in understanding the microscopic mechanism of droplets freezing on surfaces, and yet probing this behavior experimentally is extremely challenging. Molecular dynamics is a tool that allows such molecular-level interactions to be probed, but the requirement for large system sizes and in particular very long simulation times renders modeling cost-prohibitive in most simulations of rare, activated processes. By accelerating the simulation time, the barrier to adoption is greatly

reduced, opening up the possibility of using molecular dynamics as a valuable tool that complements experimental findings.

Recently, there have been advances in achieving significantly longer time-scales for water simulations with use of a coarse-grain model based on the 3-body Stillinger-Weber potential. We refer to this model as the milliwatt (mW) model. It is comprised of a single effective particle that preferentially forms four tetrahedral bonds. The model has no explicit charges, and hence no hydrogen-bonds or long-range electrostatic terms, but it reproduces the quantitative behavior of water as well as or better than conventional 3, 4, or 5 point charge models. The mW model has been shown to facilitate the observation of spontaneous freezing in water with much fewer timesteps relative to well-known traditional point-charge potentials while still reproducing many quantitative water properties of interest.

Here, we report performance improvements using the mW potential to simulate water in a cubic box with periodic boundary conditions. The simulation is performed for 256 K water molecules (that corresponds to 256 K particles for the single-site model). The simulations were sampled from the canonical ensemble where the relaxation time of the Nose-Hoover thermostat was 1 ps. The timestep was 10 fs and a neighbor-list skin of 1 Å was used. On the Intel Xeon processor-based system, we observed a speedup of 3.48× from the optimizations presented here. With the Knights Landing system supporting AVX-512CD, the speedup was 6.54×, performing 1.96× faster than the Intel Xeon processor-based system.

SUMMARY

With several straight-forward optimizations to LAMMPS routines, we were able to obtain significant performance improvements on Intel Xeon processors and Knights Landing many-core processors. Although the unoptimized code also performs better on Knights Landing when electrical power is considered, optimizations to support vectorization and improve data layout resulted in much faster simulation rates on Knights Landing ranging from 1.82× to 7.62× of the unoptimized code. These optimizations also improved performance on Intel Xeon processors, but to a lesser extent (1.19×–4.35×). The simulations on a single-socket Knights Landing processor were up to 1.96× faster than dual socket Intel Xeon processors in configurations with a significantly lower processor TDP. Compared to the best that could be performed a year ago on Haswell processors, simulations can now be performed over 8× faster with optimized LAMMPS code running on Knights Landing. As we continue to optimize long-range electrostatics, we hope to improve performance for some of the workloads even further.

Because many-core processors achieve electrical power efficiency with a larger number of cores, synchronization occurs over a larger number of processes and threads. However, the general optimization guidelines remain the same for codes that have been running on large-scale clusters and supercomputers for some time. Developers should maximize overlap of internode communications with computation and

avoid collective synchronization where possible. These optimizations do not necessarily require changing the programming model/libraries/directives used for parallelization, but more importantly, careful attention to synchronization and data sharing/communication between the software processes and threads.

ACKNOWLEDGMENT

Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Co., for the US Department of Energy under Contract No. DE-AC04-94AL85000.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this chapter.

- All of the code optimizations presented here are available in the open-source package LAMMPS, http://lammps.sandia.gov/.
- Plimpton, Steve. "Fast parallel algorithms for short-range molecular dynamics." Journal of computational physics 117.1 (1995): 1–19.
- Deserno, Markus, and Christian Holm. "How to mesh up Ewald sums. I. A theoretical and numerical comparison of various particle mesh routines." *The Journal of chemical physics* 109.18 (1998): 7678–7693.
- Brown, W. Michael, et al. "Optimizing legacy molecular dynamics software with directive-based offload." Computer Physics Communications (2015).
- Crozier, Paul S., et al. "Molecular dynamics simulation of dark-adapted rhodopsin in an explicit membrane bilayer: coupling between local retinal and larger scale conformational change." *Journal of molecular biology* 333.3 (2003): 493–514.
- Brown, W. Michael, et al. "An evaluation of molecular dynamics performance on the hybrid Cray XK6 supercomputer." *Procedia Computer Science* 9 (2012): 186–195.
- Pennycook, Simon J., et al. "Exploring SIMD for Molecular Dynamics, Using Intel® Xeon® Processors and Intel® Xeon Phi Coprocessors." Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE, 2013.
- Páll, Szilárd, and Berk Hess. "A flexible algorithm for calculating pair interactions on SIMD architectures." Computer Physics Communications184.12 (2013): 2641–2650.
- Molinero, Valeria, and Emily B. Moore. "Water Modeled As an Intermediate Element between Carbon and Silicon†." The Journal of Physical Chemistry B113.13 (2008): 4008–4016.
- Moore, Emily B., and Valeria Molinero. "Structural transformation in supercooled water controls the crystallization rate of ice." Nature 479.7374 (2011): 506–508.

CHAPTER

21 High performance seismic simulations

What's new with Knights Landing in this chapter

Example of application tuning for Knights Landing, including flat and cache memory modes, AVX-512, use of the memkind library, instruction pipeline considerations, and data structure refactoring.

In this chapter, we review our individual optimization steps to accelerate the high-order finite element software SeisSol on the latest Intel architectures including Knights Landing. SeisSol is capable of simulating the dynamic rupture process and seismic wave propagation at petascale performance in production runs. One of the most important ingredients to achieve this best-in-class performance is high single-node performance. This is only possible when leveraging all available hardware features, such as different memory subsystems, cache hierarchy, shared memory cores, and the vector and threading capabilities inside each core.

The goal of this chapter is to serve as an example on how the end-to-end optimization of an entire simulation code can be performed. We point out concepts which can be reused in other applications. It is important to note that our optimization is not limited to "just" tuning a given application. Instead, our optimizations transformed the original implementation into a version that is well aligned with today's and tomorrow's hardware properties.

The chapter is structured as follows: in the next section we describe SeisSol's numerical background and application characteristics and why the chosen spatiotemporal discretization is a good match for modern many-core architectures, that is, Intel® Xeon Phi™ (codenamed Knights Landing) product family. We also derive the macroscopic finite element kernels and their implementation. This is followed by deep-diving into our implementation of the performance-critical matrix-matrix-multiplication kernels and their tuning. We close this chapter by evaluating performance of SeisSol in two seismic setups and formulate conclusions. The version of SeisSol which was used to produce the results of this chapter can be found on github: https://github.com/SeisSol/SeisSol/releases/tag/201511.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00021-1

HIGH-ORDER SEISMIC SIMULATIONS

The simulation of seismic waves requires a vast amount of computational resources. This demand is driven by the need for accurate discretization of geometric features and highly resolved frequencies of the seismic waves. Examples for geometric features include topography, sedimentary basins, or fault systems as an important internal boundary in the earthquake generation process.

The finite element method used in this chapter provides the flexibility to tackle the posed challenges. In detail, we use a high-order Discontinuous Galerkin (DG) method, combined with tetrahedral meshes allowing for accurate discretization of complex geometrical features. In combination with the high-order ADER discretization in time, we reach the same order in space and time, allowing to reduce computational costs due to high-order convergence. Further, the used explicit ADER scheme allows for local time stepping (LTS), advancing elements with different time steps. This feature is important to circumvent severe time step restrictions on all elements, which might be posed by stability requirements of a few elements. Since our clustered LTS scheme is capable of advancing elements with small time steps separately, the impact of these elements on the overall application performance is minimized.

In terms of modern hardware, the used ADER-DG method is highly local and free of a global system matrix. Only information of face-neighboring elements contributes to an element's update. Additionally, the computational load per element is high for high-order configurations. This not only reduces communication in distributed memory settings but also allows for efficient usage of cache levels, high bandwidth memory, and prefetching techniques.

NUMERICAL BACKGROUND

SeisSol solves the three-dimensional elastic wave equations, a system of hyperbolic partial differential equations:

$$q_t + \sum_{c=1}^{3} q_{x_c} A^{x_c} = 0. \tag{21.1}$$

The space-time dependent solution $q\left(\vec{x}, t\right) = (\sigma_{11}, \sigma_{22}, \sigma_{33}, \sigma_{12}, \sigma_{13}, \sigma_{23}, u, v, w)$ contains the three normal stresses σ_{11}, σ_{22}, and σ_{33}; the three shear stress components σ_{12}, σ_{13}, and σ_{23}; and the three particle velocities u, v, and w in x_1-, x_2-, and x_3-direction. The three space-dependent Jacobians $A^{x_c}\left(\vec{x}\right)$ describe the propagation of the quantities and depend on the material parameters. The homogeneous elastic formulation (21.1) can be extended with different features, such as viscoelastic attenuation, source terms, or internal dynamic rupture boundaries.

SeisSol uses the DG method for spatial discretization and subdivides the three-dimensional computational domain via an unstructured tetrahedral mesh. In each

tetrahedron k, we store the corresponding part Q_k of the overall, numerical solution as modal coefficients of a polynomial basis. In comparison to a Continuous Galerkin formulation, continuity of the numerical solution at the elements' boundaries is not imposed. We call the modal coefficients Q_k for all our nine elastic quantities degrees of freedom (DOFs). The quality of the approximation depends on the degree N of the per-element polynomial approximation. A given degree N results in an order $\mathcal{O} = N + 1$ discretization at the cost of an increased number $B_\mathcal{O}$ of per-quantity, modal coefficients. Thus, the total number of DOFs per element is given by $B_\mathcal{O} \times 9$. Typical runs of SeisSol use fifth or sixth order of convergence, leading to $B_5 = 35$ or $B_6 = 56$ basis functions. A desired order of convergence $\mathcal{O}$ of the overall scheme is achieved by using the same order $\mathcal{O}$ for the spatial and temporal discretization.

The computational core of SeisSol iterates over each tetrahedron k twice to update it to the next (possibly local) time step. The first iteration performs all element-local operations, which require only the DOFs Q_k of the tetrahedron k to proceed. In contrast, the second iteration accounts for the contribution of the face-neighboring elements to an element's time step. Here, each tetrahedron k completes the update of its DOFs Q_k by using data of its four face neighbors k_i, $i \in 1 \ldots 4$.

In the following paragraphs, we formulate the fully discrete form of computational core of SeisSol. This formulation solely uses matrix-operators to describe the time, volume, and surface integration. Our matrix-operators are only applied locally to data of an element or face-neighboring element. Thus, the scheme is "matrix-free," meaning that no system matrices covering all elements are assembled. Our "global" matrices (e.g., stiffness matrices) are derived using symbolic integration with respect to a reference element. These matrices are small in size ($B_\mathcal{O} \times B_\mathcal{O}$) and shared among all elements. Here, we use, instead of a multiindex for the principal functions, a one-dimensional index for the basis functions in the generalized tensor product expansion.

Element-local: The first step of the element-local iteration recursively computes the time derivatives $\partial^d / \partial t^d Q_k^{n_k}$ of a tetrahedron k from the DOFs at time step n_k, $\partial^0 / \partial t^0 Q_k^{n_k} = Q_k^{n_k}$:

$$\frac{\partial^{d+1}}{\partial t^{d+1}} Q_k^{n_k} = -\sum_{c=1}^{3} \hat{K}^{\xi_c} \left(\frac{\partial^d}{\partial t^d} Q_k^{n_k} \right) A_k^{\xi_c}, \quad d \in 1, \ldots, \mathcal{O} - 1. \tag{21.2}$$

The three Jacobians $A_k^{\xi_c}$ (size 9×9) are unique per tetrahedron, while the transposed stiffness matrices $\hat{K}^{\xi_c}$ (size $B_\mathcal{O} \times B_\mathcal{O}$) are identical for all tetrahedrons. Note that the effect of element-dependent scalars, appearing in the ADER-DG discretization, is included in the Jacobians. Similar, the inverse, diagonal mass matrix is premultiplied to the transposed stiffness matrices. In the following ADER-DG kernels, we follow the same approach. Fig. 21.1 illustrates the involved matrix-matrix multiplications in the first two steps of this recursive scheme for a fifth-order method.

Simultaneously to the time derivatives we derive element-local time integrated DOFs $\mathcal{T}_k(t^{n_k}, t^{n_k}, \Delta t_k)$ (size $B_\mathcal{O} \times 9$) by directly multiplying the derivatives with time step dependent scalars:

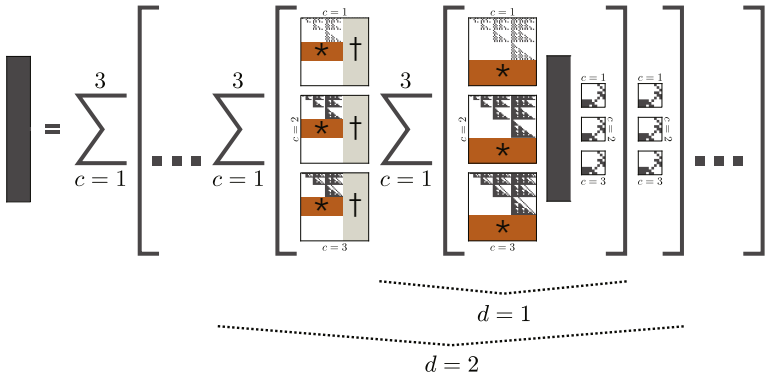


FIG. 21.1

Recursive computation of the time derivatives in the ADER scheme (see Eq. (21.2)). With the DOFs as input, we obtain higher derivatives recursively by multiplying each summand with a transposed stiffness matrix from the left and a Jacobian from the right. Our scheme exploits zero-blocks, which are generated for higher derivatives. Blocks marked $\star$ generate, and blocks marked † hit zero blocks, in the derivatives. We obtain the final time integrated DOFs from the derivatives via Eq. (21.3), which sums up scaled derivatives.

$$\mathcal{T}_k(t_0,\hat{t},\Delta t)=\sum_{d=0}^{\mathcal{O}-1}\frac{(\hat{t}+\Delta t-t_0)^{d+1}-(\hat{t}-t_0)^{d+1}}{(d+1)!}\frac{\partial^d}{\partial t^d}Q_k(t_0). \tag{21.3}$$

t_0 is the simulation time at which we computed the time derivatives, $\hat{t}$ is the start of the time integration interval $[\hat{t},\hat{t}+\Delta t]$, and Δt its length. For later use by k's face-neighbors in their update with the neighboring elements' contribution, we add, depending on the LTS configuration of k, the time integrated DOFs $\mathcal{T}_k$ to a time integration buffer $\mathcal{B}_k$ and/or store the derivatives in $\mathcal{D}_k$.

Last, the element-local step updates the DOFs $Q_k^{n_k}$ with the contribution of the volume integration and element-local contribution to the surface integration:

$$Q_k^{*,n_k+1}=Q_k^{n_k}-\sum_{c=1}^{3}K^{\xi_c}(\mathcal{T}_k)A_k^{\xi_c}-\sum_{i=1}^{4}F^{-,i}(\mathcal{T}_k)A_{k,i}^{+}. \tag{21.4}$$

Q_k^{*,n_k+1} is the intermediate solution, which still requires the neighboring elements' contribution for the final state of tetrahedron k at the next time step n_k+1. Again, the Jacobians $A_k^{\xi_c}$ (size 9×9) are fixed per tetrahedron. Also the four flux solvers $A_{k,i}^{+}$ (size 9×9) are fixed per tetrahedron, while the three stiffness matrices K^{ξ_c} (size $B_{\mathcal{O}}\times B_{\mathcal{O}}$) are identical for all elements. Fig. 21.2 shows an illustration of all involved matrix operations for a fifth-order scheme.

Neighbor contribution: The neighboring elements' contribution either uses the time derivatives $\mathcal{D}_{k_i}$ and first assembles time integrated DOFs $\mathcal{T}_{k_i}$ via Eq. (21.3) or directly operates on the buffers of the neighbors: $\mathcal{T}_{k_i}:=\mathcal{B}_{k_i}$. In this step, information flows from an element k's face-neighbors k_i to k. Starting from the intermediate DOFs Q_k^{*,n_k+1}, we obtain the final DOFs of tetrahedron k at time t^{n_k+1} via:

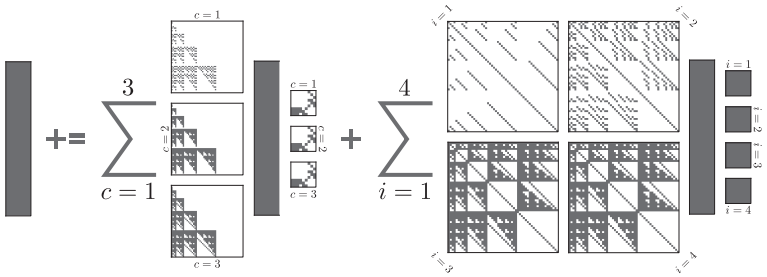


FIG. 21.2

Element-local update-operations of the ADER-DG scheme (see Eq. (21.4)). The first sum illustrates the volume integration. Here, in every summand, we multiply the DOFs with a stiffness matrix from the left and a Jacobian from the right. The second sum illustrates the element-local contribution of the surface integration. Here, in every summand, we multiply the DOFs with a flux matrix from the left and a flux solver from the right.

$$Q_k^{n_k+1} = Q_k^{*,n_k+1} - \sum_{i=1}^{4} F^{+,i,j_k(i),h_k(i)}(\mathcal{T}_{k_i}) A_{k,i}^{-}. \tag{21.5}$$

$A_{k,i}^{-}$ (size 9×9) are four per-element flux solvers. $F^{+,i,j_k(i),h_k(i)}$ (size $B_{\mathcal{O}} \times B_{\mathcal{O}}$) are 48, element-independent, neighboring flux matrices of which we only use four in an element update. The choice depends on the mesh structure. In Fig. 21.3, we illustrate the structure of all involved matrix operations occurring in the neighbor contribution for a fifth-order scheme.

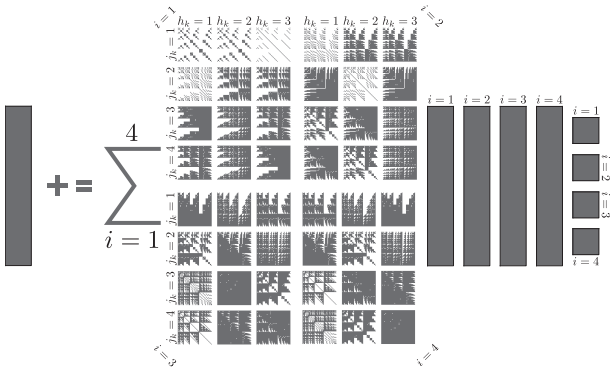


FIG. 21.3

Contribution of the four face-neighboring tetrahedrons in the ADER-DG scheme (see Eq. (21.5)). For every summand, we multiply the time integrated DOFs of the face-neighbor with 1 out of 12 flux matrices from the left and with a flux solver from the right.

APPLICATION CHARACTERISTICS

Note

This chapter uses C++ code-snippets to discuss the implementation of our ADER-DG kernels. All of the following code-snippets follow a set of coding conventions. For example, the code shown in Fig. 21.4 utilizes many of the most important conventions.

Function parameters are prepended by i_ if the parameter is accessed read-only by the function, prepended by o_ if the parameter is accessed write-only, and by io_ if the function reads and writes the parameter. For example, the call-by-value parameter i_timeStep in Line 1 of the code in Fig. 21.4 is accessed in a read-only way. Inside function bodies we prepend 1_ for local variables and m_ for member variables of the surrounding C++ class.

We use the typedef-name real as alias for floating-point values in our computational scheme. This allows us to switch between the native C++ types `float` and `double` at compile time. `float` translates to single-precision (32 bits per value), while `double` is double-precision (64 bits per value). All variables related to our time-management, such as `i_timeStep` in Line 1 of Fig. 21.4, are not performance relevant. Therefore we fix these to double-precision for improved accuracy.

Preprocessor variables are named in uppercase. Examples shown in Fig. 21.4 are `CONVERGENCE_ORDER` or `NUMBER_OF_ALIGNED_DOFS`. Before invoking the compiler, we use the preprocessor to replace all occurrences of these variables with values matching our requirements.

Time Kernel: The combined time scheme of (21.2) and (21.3) is our first ADER-DG kernel. Lines 1–6 in Fig. 21.4 contain the function parameters of our kernel. `i_timeStep` is the time step Δt used to compute the time integrated DOFs via (21.3). `computeAder` in Fig. 21.4 computes time integrated DOFs of the tetrahedron's complete time step in any case; here $t_0 = \hat{t} = t^{n_k}$ holds and the respective scalars cancel out.

`i_stiffnessMatrices` are the three locations in memory, which contain the values of our matrices $\hat{K}^{\xi_1}$, $\hat{K}^{\xi_2}$, and $\hat{K}^{\xi_3}$. The DOFs $Q_k(t^{n_k})$ of the tetrahedron are passed via `i_dofs` and the matrices $A_k^{\xi_1}$, $A_k^{\xi_2}$, and $A_k^{\xi_3}$ via `i_starMatrices`. `o_timeIntegrated` is the location to which our kernel writes the time integrated DOFs. Similarly, `o_timeDerivatives` might point to the location where we store the derivatives. We only have to store derivatives permanently if a tetrahedron k is required to provide time derivatives $\mathcal{D}_k$ for its face-neighbors in LTS settings. Our `computeAder` function in Fig. 21.4 assumes that we pass the null-pointer constant `NULL` if no derivatives have to be stored.

Considering the data-heavy structures in the arguments to the kernel, the global matrices $\hat{K}^{\xi_c}$ most likely are in cache, since the time kernel is called over and over for a large number of elements. The DOFs $Q_k^{n_k}$ in `i_dofs` and matrices $A_k^{\xi_c}$ in `i_starMatrices` are usually touched for the first time. However, our storage of element-local data ensures linear access in the local kernels enabling hardware prefetching benefits. The computational load in the local ADER-DG kernels is high for high-order simulations and, in this case, the memory-accesses are most likely shadowed behind computations. The pointer `o_timeIntegrated` points either to a temporary data

```
1  void computeAder(double i_timeStep,
2                   real** i_stiffnessMatrices,
3                   real*  i_dofs,
4                   real   i_starMatrices[3][STAR_NNZ],
5                   real*  o_timeIntegrated,
6                   real*  o_timeDerivatives ) {
7  // scalars in the taylor-series expansion
8  real l_scalar = i_timeStep;
9
10 // temporary result
11 real l_temporaryResult[NUMBER_OF_ALIGNED_DOFS]
12   __attribute__((aligned(PAGESIZE_STACK)));
13 real l_derivativesBuffer[NUMBER_OF_ALIGNED_DERS]
14   __attribute__((aligned(PAGESIZE_STACK)));
15
16 // initialize time integrated DOFs and derivatives
17 initialize( l_scalar, i_dofs,
18             o_timeIntegrated, l_derivativesBuffer );
19
20 // stream out first derivative (d=0)
21 if ( o_timeDerivatives != NULL ) {
22   streamstoreFirstDerivative( i_dofs,
23                               o_timeDerivatives );
24 }
25
26 // compute derivatives and contributions to time integrated DOFs
27 for( unsigned int l_derivative = 1;
28      l_derivative < CONVERGENCE_ORDER; l_derivative++ ) {
29   // iterate over dimensions
30   for( unsigned int l_c = 0; l_c < 3; l_c++ ) {
31     // compute $K_{\xi_c}.Q_k$ and $(K_{\xi_c}.Q_k).A*$
32     m_matrixKernels[ (l_derivative-1)*4 + l_c ] (
33       i_stiffnessMatrices[l_c],
34       l_derivativesBuffer+m_derivativesOffsets[l_derivative-1],
35       l_temporaryResult,
36       NULL, NULL, NULL );
37
38     m_matrixKernels[ (l_derivative-1)*4 + 3   ] (
39       l_temporaryResult,
40       i_starMatrices[l_c],
41       l_derivativesBuffer+m_derivativesOffsets[l_derivative],
42       NULL, NULL, NULL );
43   }
44
45   // update scalar for this derivative
46   l_scalar *= i_timeStep / real(l_derivative+1);
47
48   // update time integrated DOFs
49   integrateInTime( l_derivativesBuffer,
50                    l_scalar,
51                    l_derivative,
52                    o_timeIntegrated,
53                    o_timeDerivatives );
54 }}
```

FIG. 21.4

Time kernel computing time integrated DOFs and time derivatives (if requested).

structure, overwritten in every call of the time kernel, or to memory for global time stepping relations. The derivatives `o_timeDerivatives` are accessed write-only.

In Line 8 shown in Fig. 21.4, we initialize the scalar $\left((\hat{t}+\Delta t-t_0)^{d+1}-(\hat{t}-t_0)^{d+1}\right)/(d+1)!$ of the zeroth derivative ($d=0$) summand in (21.3). Lines 11–14 define two arrays, which we use to store intermediate results of our time kernel. Both arrays are aligned to page size boundaries. The call `initialize` in lines

17–18 shown in Fig. 21.4 initializes the zeroth derivative with the DOFs Q_k and the intermediate time integrated DOFs with $\Delta t \cdot Q_k$. Also, in the function `initialize` all higher derivatives are reset to zero. We are not using the commonly known `memset` routine in this case as it might be implemented by the compiler using nontemporal stores. `initialize` uses intrinsics (`_mm(256,512)_setzero_pd` and `_mm(256,512)_store_pd`) to achieve an in-cache erase with the lowest overhead.

The function call `streamstoreFirstDerivative` in lines 22–23 wraps a set of low-level intrinsic functions performing nontemporal stores (via the `_mm(256,512)_stream_pd` intrinsic) for the zeroth derivatives if requested (`o_timeDerivatives != NULL`). Since our time kernel permanently stores the derivatives in a write-only fashion, nontemporal stores bypass the cache and directly write to memory. This avoids read for ownership, where the entire cache-line is read from memory before we would overwrite it completely.

The loop in lines 27–54 shown in Fig. 21.4 performs, per iteration, a single step of the recursive derivative computation (21.2). Additionally, following (21.3), we add the effect of the current iteration's derivatives directly to the time integrated DOFs `o_timeIntegrated`. The inner-loop in lines 30–43 iterates over dimensions ξ_1, ξ_2, and ξ_3. In each inner iteration, we perform two matrix-matrix multiplications by calling function-pointers of `m_matrixKernels`. The first call multiplies the previous derivative with the matrices $\hat{K}^{\xi_c}$ from the left and stores the intermediate results to `l_temporaryResult`. The jump in memory for the derivatives on the stack via `m_derivativesOffsets` is equivalent to our vector-aligned storage scheme for compressed derivatives, which exploits the growing zero-blocks of higher derivatives. The second call multiplies the intermediate results with the matrices $A_k^{\xi_c}$ from the right and adds the result to the respective derivative in `l_derivativesBuffer`.

The array of function pointers `m_matrixKernels` allows us to use optimized matrix-matrix-multiplication kernels for the individual operations. Since the array-index for the multiplication from the left with $\hat{K}^{\xi_c}$ depends on both loop-indices, `l_derivative` and `l_c`, we are able to hardwire the current level of recursion and the current dimension in the respective matrix kernel. For the second call, we only hardwire the level of recursion in our matrix kernels, since we assume identically shaped matrices $A_k^{\xi_c}$ (see Fig. 21.1). We choose the individual matrix-matrix-multiplication kernels via auto-tuning based on the used architecture, order of convergence, and precision.

The remaining lines 45–53 shown in Fig. 21.4 add the current derivative's contribution to the time integrated DOFs `o_timeIntergrated` and permanently store the derivative if requested. Again we reduce the pressure on the memory subsystem by using streaming stores in the function `integrateInTime` for storing the derivative.

Volume Kernel: The first sum in the local update step (21.4) computes the contribution of the volume integration. Fig. 21.5 shows the volume kernel `computeVolume` performing this operation. `computeVolume's` first function parameter is the array `i_stiffnessMatrices` containing the locations of the three matrices K^{ξ_c} in memory. The second parameter `i_timeIntegrated` points to the time integrated DOFs of the element. Matrices $A_k^{\xi_c}$ are passed via `i_starMatrices` and the element's DOFs $Q_k^{n_k}$ via `io_dofs`.

```
1  void computeVolume(real** i_stiffnessMatrices,
2                     real*  i_timeIntegrated,
3                     real   i_starMatrices[3][STAR_NNZ],
4                     real*  io_dofs ) {
5  // temporary result
6  real l_temporaryResult[NUMBER_OF_ALIGNED_DOFS]
7    __attribute__((aligned(PAGESIZE_STACK)));
8
9  // iterate over dimensions
10 for( unsigned int l_c = 0; l_c < 3; l_c++ ) {
11   m_matrixKernels[l_c]( i_stiffnessMatrices[l_c],
12                         i_timeIntegrated,
13                         l_temporaryResult,
14                         NULL, NULL, NULL );
15   m_matrixKernels[3](   l_temporaryResult,
16                         i_starMatrices[l_c],
17                         io_dofs,
18                         NULL, NULL, NULL );
19 }}
```

FIG. 21.5

Volume kernel computing the contribution of the volume integration to a time step.

It is very likely that all input data already resides in cache, since we call the volume kernel directly after our time kernel shown in Fig. 21.4. The time kernel computes the time integrated DOFs in `i_timeIntegrated`. In addition, it also uses the three matrices $A_k^{\xi_c}$ in `i_starMatrices` and the DOFs in `io_dofs` for the derivative computation. Further, the matrices K^{ξ_c} are global and used repeatedly in the element-local update (21.4) of every element. This is supported by our execution-aware storage of the global matrices, reflecting the access patterns of the kernels. Here, the matrices K^{ξ_c} directly follow matrices $\hat{K}^{\xi_c}$ thus can benefit from hardware prefetching.

The structure of the volume kernel is similar to the derivative computation in the time kernel. Again, in lines 6–7 shown in Fig. 21.5, we define a page-size aligned array for the temporary data. Next, lines 10–19 iterate over the three dimensions ξ_c in the reference coordinate system and compute the corresponding volume contributions. Inside the loop, we call a set of optimized matrix-matrix-multiplication kernels, stored as function pointers in `m_matrixKernels`. This array exclusively belongs to the volume kernel and thus holds different function pointers in comparison to the corresponding array of the time kernel. The first call in lines 11–14 of Fig. 21.5 multiplies the time integrated DOFs from the left with matrices K^{ξ_c} and stores the intermediate results. Next, in lines 15–18, we multiply the results of the previous matrix-matrix multiplication with $A_k^{\xi_c}$ from the right and add the result to the DOFs.

By using function pointers `m_matrixKernels`, we are again able to hardwire individual matrix kernels for all four operations.

Local Surface Kernel: Our local surface kernel in Fig. 21.6 updates the DOFs with the second sum of (21.4). The first input parameter is the enumeration `i_faceTypes`, which encodes the types of the four faces. A pointer to the 52 global matrices $F^{-,i}$ and $F^{+,i,j,h}$, of which we only use the first four matrices $F^{-,i}$, is the second parameter. The next arguments pass the element-local data, which are the time integrated DOFs, the four matrices $A^-_{k,i}$, and the element's DOFs. The last two parameters

```
1  void computeLocalSurface(enum faceType i_faceTypes[4],
2                           real          *i_fluxMatrices[52],
3                           real          *i_timeIntegrated,
4                           real           i_fluxSolvers[4]
5                             [NUMBER_OF_QUANTITIES*NUMBER_OF_QUANTITIES],
6                           real          *io_dofs,
7                           real          *i_timeIntegrated_prefetch,
8                           real          *i_dofs_prefetch ) {
9  // temporary product
10 real l_temporaryResult[NUMBER_OF_ALIGNED_DOFS]
11   __attribute__((aligned(PAGESIZE_STACK)));
12
13 for( unsigned int l_face = 0; l_face < 4; l_face++ ) {
14   // Riemann problem for dynamic rupture faces is solved separately
15   if( i_faceTypes[l_face] != dynamicRupture ) {
16     // compute elements contribution
17     m_matrixKernels[l_face]( i_fluxMatrices[l_face],
18                              i_timeIntegrated,
19                              l_temporaryResult,
20                              NULL, NULL, NULL );
21
22     if (l_face == 1) {
23       m_matrixKernels[54](   l_temporaryResult,
24                              i_fluxSolvers[l_face],
25                              io_dofs,
26                              NULL, i_timeIntegrated_prefetch, NULL );
27     } else if (l_face == 2) {
28       m_matrixKernels[54](   l_temporaryResult,
29                              i_fluxSolvers[l_face],
30                              io_dofs,
31                              NULL, i_dofs_prefetch,           NULL );
32     } else {
33       m_matrixKernels[52](   l_temporaryResult,
34                              i_fluxSolvers[l_face],
35                              io_dofs,
36                              NULL, NULL,                      NULL );
37     }
38   } else {
39     if (l_face == 0) {
40       for( unsigned int l_p = 0; l_p < NUMBER_OF_ALIGNED_DOFS;
41                                  l_p += FP_VALUES_PER_CACHELINE ) {
42         _mm_prefetch( (const char*) i_timeIntegrated_prefetch+l_p,
43                                     _MM_HINT_T1 );
44       }
45     } else if (l_face == 1) {
46       for( unsigned int l_p = 0; l_p < NUMBER_OF_ALIGNED_DOFS;
47                                  l_p += FP_VALUES_PER_CACHELINE ) {
48         _mm_prefetch( (const char*) i_dofs_prefetch+l_p,
49                                     _MM_HINT_T1 );
50       }
51     }
52   }
53 }}
```

FIG. 21.6

Local surface kernel computing the element's contribution of the surface integration to a time step.

`i_timeIntegrated_prefetch` and `i_dofs_prefetch` are specific to the implementation on Knights Landing. Prefetching on Knights Landing is more important than on Intel Xeon processors because of the absence of a shared last level cache (LLC). Prefetching can help both Intel Xeon and Intel Xeon Phi based systems, a good examination of how to explore this is referenced in *For More Information* at the end of this chapter. Here, we also pass locations for the next element's DOFs and

time-integrated DOFs. We require those in the upcoming time kernel and therefore issue prefetching instructions throughout the kernel.

Considering the data-heavy arguments once again, the matrices $F^{-,i}$ most likely reside in cache, because we repeatedly call the local surface kernel for our elements. The time integrated DOFs in `i_timeIntegrated` and DOFs in `io_dofs` have been touched in the previous calls of the time and volume kernel, thus, most likely, are still in cache as well. Matrices $A^{-}_{k,i}$ are untouched in general. However, we assume that the prefetcher is able, at least for high-order simulations, to shadow the loads from memory.

In lines 10–11 shown in Fig. 21.6, we once again define storage for our two-way matrix-matrix multiplications. The loop over the four faces of the tetrahedron is realized in lines 13–53. We only continue with the computation of a face's contribution if the face is not a dynamic rupture face (Line 15), which is true for the vast majority of faces. Dynamic rupture faces use a specialized surface integration. Next, lines 17–20 multiply the time integrated DOFs from the left with matrices $F^{-,i}$ and store the result in our intermediate storage. The second call of our matrix kernels in lines 22–37 multiplies the intermediate result from the right with a matrix $A^{-}_{k,i}$ and updates the DOFs with the contribution of the current face.

Again, the class member `m_matrixKernels` allows us to hardwire optimized matrix-matrix-multiplication kernels for the five calls. However, the local surface kernel shares `m_matrixKernels` with the neighboring surface kernel, discussed in the next paragraph. Specific to Knights Landing, the differentiation between `m_matrixKernels[54]` and `m_matrixKernels[52]` allows us to perform execution-aware software prefetching. While the performed operation of the three different variants is identical (multiplication with $A^{-}_{k_i}$), the calls differ with respect to software prefetching. For the first and last face in lines 33–36, only element-local software prefetching is performed inside the matrix kernel. The second call in lines 23–26 prefetches time integrated DOFs of the next element, while the third call in lines 28–31 prefetches the DOFs of the upcoming element. In the case of dynamic rupture faces, the surface integration is performed separately and we perform dry software prefetches in lines 39–51.

Neighboring Surface Kernel: The neighboring elements' contribution to the surface integral (21.5), shown in Fig. 21.7, is our last ADER-DG kernel. While the time kernel, volume kernel, and local surface kernel all use element-local data only, the neighboring surface kernel is the only ADER-DG kernel accessing data of neighboring elements.

The function parameters of the kernel are given in lines 1–9. The first parameter `i_faceTypes` contains, analogous to the local surface kernel in Fig. 21.6, the types of the four faces. Argument `i_neighboringIndices` contains, in dependency of the local face i, the id of the neighboring face $j_k(i)$ and the orientation $h_k(i)$ with respect to the reference element. `i_fluxMatrices` again contains the 52 matrices $F^{-,i}$ and $F^{+,i,j,h}$.

The function parameter `i_timeIntegrated` in Fig. 21.7 points to the time integrated DOFs. If a face-neighbor provides derivatives, we have to assemble the time

```
1  void computeNeighSurface( enum faceType i_faceTypes[4],
2                            int           i_neighboringIndices[4][2],
3                            real         *i_fluxMatrices[52],
4                            real         *i_timeIntegrated[4],
5                            real          i_fluxSolvers[4]
6                              [NUMBER_OF_QUANTITIES*NUMBER_OF_QUANTITIES],
7                            real         *io_dofs,
8                            real         *i_faceNeighbors_prefetch[4],
9                            real         *i_fluxMatrices_prefetch[4] ) {
10
11 // temporary product
12 real l_temporaryResult[NUMBER_OF_ALIGNED_DOFS]
13   __attribute__((aligned(PAGESIZE_STACK)));
14
15 // perfetch the first cacheline per column for the DOFs
16 for( unsigned int l_var = 0; l_var < NUMBER_OF_QUANTITIES; l_var++ ) {
17   _mm_prefetch( (const char*) io_dofs + (l_var*
18                 NUMBER_OF_ALIGNED_BASIS_FUNCTIONS), _MM_HINT_T1 );
19 }
20
21 // iterate over faces
22 for( unsigned int l_face = 0; l_face < 4; l_face++ ) {
23   // absorbing: no contribution, dynamic rupture: separate
24   if( i_faceTypes[l_face] != outflow &&
25       i_faceTypes[l_face] != dynamicRupture ) {
26     // id of the flux matrix (0-3: local, 4-51: neighboring)
27     unsigned int l_id;
28
29     // compute the neighboring elements flux matrix id.
30     if( i_faceTypes[l_face] != freeSurface ) {
31       // derive memory and kernel index
32       l_id = 4                                          // jump: F^{-, i}
33            + l_face*12                                  // jump: i
34            + i_neighboringIndices[l_face][0]*3          // jump: j
35            + i_neighboringIndices[l_face][1];           // jump: h
36     }
37     else { // free surface: fall back to F^{-, i}
38       l_id = l_face;
39     }
40
41     real* l_faceNeigh_prefetch = NULL;
42     real* l_flux_prefetch      = NULL;
43     if (i_faceNeighbors_prefetch[l_face] != NULL) {
44       l_faceNeigh_prefetch     = i_faceNeighbors_prefetch[l_face];
45     } else {
46       l_faceNeigh_prefetch     = io_dofs;
47     }
48     if (i_fluxMatrices_prefetch[l_face] != NULL) {
49       l_flux_prefetch          = i_fluxMatrices_prefetch[l_face];
50     } else { // fall back to valid location if NULL is provided
51       l_flux_prefetch          = i_fluxMatrices[l_id];
52     }
53
54     // compute neighboring elements contribution
55     m_matrixKernels[l_id]( i_fluxMatrices[l_id],
56                            i_timeIntegrated[l_face],
57                            l_temporaryResult,
58                            l_flux_prefetch,l_faceNeigh_prefetch,NULL );
59
60     m_matrixKernels[53](   l_temporaryResult,
61                            i_fluxSolvers[l_face],
62                            io_dofs,
63                            l_flux_prefetch,l_faceNeigh_prefetch,NULL );
64   }
65 }}
```

FIG. 21.7

Neighboring surface kernel computing the neighboring elements' contribution of the surface integration to a time step.

integrated DOFs via (21.3) first, before passing them to `computeNeighSurface` in Fig. 21.7. For this purpose, prior to the call of `computeNeighSurface`, we may compute the time integrated DOFs from the derivatives based on a LTS bitmask, encoding the LTS configuration of an element with respect to its face-neighbors, and store them in a temporary array. Therefore, recently computed time integrated DOFs in `i_timeIntegrated` are most probably in cache. The location of the time buffers, which are used directly and not copied to the temporary array, depends on the data-locality of the neighboring elements. The next two parameters in lines 5–7 of Fig. 21.7 point to the matrices $A^{-}_{k,i}$ and the DOFs.

The hardware prefetcher most likely fails to predict the accesses of derivatives and buffers initially, due to our unstructured tetrahedral meshes. For high-order simulations, the neighboring accesses put the highest pressure on the memory subsystem. Here, we use software prefetching on Knights Landing again. For this purpose, we pass two additional parameters `i_faceNeighbors_prefetch` and `i_fluxMatrices_prefetch`. The first parameter `i_faceNeighbors_prefetch` is related to the time data touched by the neighboring surface kernel. Similarly, `i_fluxMatrices_prefetch` contains the locations of the four global flux matrices $F^{i,j,h}$, which are used in the neighboring surface kernel.

As in all other ADER-DG kernels, lines 12–13 define page-size aligned memory for our two-step matrix-matrix multiplications. The loop over the faces in lines 22–65 only processes a face if it is not a dynamic rupture face and no outflow boundary conditions are set (see lines 24–25). In the case of dynamic rupture faces, the surface integral is computed separately, and in the case of outflow boundary conditions, no neighboring surface-contribution is applied at all. We derive the face's id in `i_fluxMatrices` in lines 29–39. In the case of free-surface boundary conditions, we use $F^{-,i}$. In all other cases, we follow our ascending storage to get the right matrix $F^{+,i,j,h}$.

In lines 55–58, we multiply the time integrated DOFs from the left with matrices $F^{+,i,j,h}$ or $F^{-,i}$ in the case of free-surface boundary conditions and store the intermediate result. Finally, in lines 60–63, we multiply the intermediate result from the right with $A^{+}_{k,i}$ and update the DOFs with the face's contribution. After all faces are processed in Fig. 21.7, the element is at the next time step.

The class member `m_matrixKernels` is identical to the local surface kernel in Fig. 21.6. In addition to the four shared matrix-matrix-multiplication kernels performing the multiplications with $F^{-,i}$ from the left, we are able to hardwire additional kernels for the multiplications with $F^{+,i,j,h}$ and $A^{+}_{k,i}$.

Considering the Knights Landing specific software prefetches of the neighboring surface kernel, we load the first cacheline of the DOFs columns in lines 16–19. The prefetching performed in lines 41–63 is the most complex. The entries of `i_fluxMatrices_prefetch` and `i_faceNeighbors_prefetch` contain the location of data used in the upcoming operation. In the rare case of boundary conditions, the NULL constant is present and a special handling is performed. However, in the general case, an entry contains the location of the flux matrix and face-neighboring

time data used in the next iteration of the face loop in lines 22–65. For the last face (`l_face==3`) they contain the location of data used in the upcoming call of the neighboring surface kernel. Here, they contain the location of the flux matrix and face-neighboring time data used in the first iteration of the face loop in the next element's update. This means, analogous to the local surface kernel, that we prefetch data across the element boundaries and already consider where the ADER-DG scheme heads next.

INTEL ARCHITECTURE AS COMPUTE ENGINE

The discussion of the ADER-DG kernels in SeisSol made clear that the underlying algorithm is well-suited for a modern high-performance processor. The introduced update scheme requires dense compute capabilities as well as high memory bandwidth for selected data structures. These requirements are well fulfilled by the Intel® Xeon® E5 v3 processor family and, even more importantly, Knights Landing. In the upcoming sections, we will follow a bottom-up approach to address how hardware features such as SIMD units, shared caches, or high-bandwidth memory can be leveraged to run high-order seismic simulations with high efficiencies.

HIGHLY-EFFICIENT SMALL MATRIX KERNELS

Small matrix multiplication kernels form the computational back-bone of SeisSol. In order to achieve best possible computational effort and/or performance, a straightforward approach would be to either use a sparse matrix format or to densify all matrices and to call a high performance BLAS (basic linear algebra subroutines) implementation. However, both alternatives do not yield to the desired performance. This is because the involved matrix shapes are not within the ranges for which BLAS libraries are typically optimized. Furthermore, many computations take place at cache level, and therefore, cache bandwidth is a serious limiting factor when processing the matrix operations in sparse fashion.

We address this area of conflict by problem-specific code generation. The matrix shapes are known at compile time for each convergence order. Therefore, implementations which match the matrix structures can be derived. This can be done for both cases, sparse and dense. In the following, the corresponding implementation details and performance tricks are discussed. As SeisSol is implemented using FORTRAN and C++, we rely on a column major matrix representation in both dense and sparse kernels. The employed sparse format is called CSC (compressed sparse column). Furthermore, we rely on the regular BLAS notation: $C = \alpha A \cdot B + \beta C$, $C \in \mathbb{R}^{M \times N}$, $A \in \mathbb{R}^{M \times K}$, and $B \in \mathbb{R}^{K \times N}$. lda, ldb, and ldc define the length of the leading memory dimension of each matrix, and therefore lda $\geq M$, ldb $\geq K$, and ldc $\geq M$ need to hold true. Since we just need the straightforward cases of $\alpha = 1$ and $\beta \in \{0, 1\}$, we do not discuss the efficient integration of arbitrary α and β values into our kernels.

Note

SeisSol kernels are generated by the **LIBXSMM** open source project which is hosted at github: https://github.com/hfp/libxsmm. This library is already used in other scientific applications (e.g., CP2K—a quantum chemistry code which relies on block sparse matrices and NekBox—a strip-down of Nek5000 for box-shaped domains which uses the spectral element method to solve the Navier-Stokes equations) which demand small matrix multiplications.

SPARSE MATRIX KERNEL GENERATION AND SPARSE/DENSE KERNEL SELECTION

In the following sections, we will only cover dense matrix multiplications for implementing SeisSol's integration routines using LIBXSMM. However, when we recall the matrix shapes depicted in Figs. 21.1–21.3, it is essential to analyze if sparse matrix multiplication will further reduce SeisSol's time-to-solution as less calculations are performed. As these calculations are less structured, we expect lower flop rates. Therefore, the nonintuitive scenario of running with shorter time to solution and lower achieved peak performance might be possible. As mentioned above, the involved matrix sizes are very small, and therefore all operations take place at cache level. In case of a sparse implementation, loading indexing structures which represent the matrix's sparse structure (including their evaluation) occupy execution ports or pipeline slots in the processing core. These resources are therefore blocked and cannot be used for running arithmetic operations. This is a substantially different from large sparse matrices were the memory wall limitation allows "hiding" these management operations. In the past and on older Intel architectures, we addressed this problem by an offline structure analysis and code generation phase. This resulted in CSC-like kernels which encode the sparsity pattern in the instruction flow. We reevaluated this strategy leveraging AVX2 (Intel Xeon E5 v3 processors) and AVX-512 (Knights Landing) instruction set extensions. Unfortunately, runtime savings are very minor on Intel Xeon E5 v3 processors: 10% saving for low order runs and 1.5% in case of order six runs. In case of AVX-512 on Knights Landing processors, the sparse variants offer no advantage and even forbid prefetching techniques. The reason can be found in the design of the core. Both analyzed architectures have two load- and one store-port(s) at full vector processing unit (VPU) length. As we are operating out of L1 without significant reuse in case of the sparse variant, this limits us to at most 50% peak performance of the core as we can only store one register per cycle. The dense variants have register accumulators and can therefore exploit both VPUs per core and catch up, in terms of performance, with the sparse formulation. More information on this approach is contained in two papers from 2014 listed in *See More Information* at the end of this chapter. For this chapter, we now focus on the dense matrix work which yielded the most performance improvement. Our discussions assume double precision since this is needed for high-order runs of SeisSol.

DENSE MATRIX KERNEL GENERATION: AVX2

Let us begin the discussion of the actual kernel implementation by a deep dive into the code generation principles used. In order to ensure best performance independently from the compiler, the generated code includes the GEMM routines as inline assembly. This is supported by the Intel compiler suite as well as the GNU tool chain. Furthermore, the code generator supports arbitrary M, K, N, lda, ldb, and ldc. This flexibility is needed such that we can mix-and-match the various different (sub)-matrices shapes occurring especially in the ADER time and volume integration kernel to reduce the amount of zero-padding. For best performance, we align lda and ldc to the 32-byte memory alignment requirement of AVX2. This alignment ensures that the load performance inside the GEMM microkernel is not lowered by unaligned loads which might suffer from cacheline splits (vector loads and stores of data spanning across cacheline boundaries).

Whereas the dimensions of the GEMM microkernel for register blocking do not play an important role in case of large GEMM (we simply use the fastest one), for the small sizes needed in SeisSol, the microkernel's shape is essential. Especially the number of quantities enforces $N = 9$, which is not a standard case that is commonly used in GEMM high-performance implementations. However, it is possible to apply the design principles of large GEMMs to small GEMMs when selecting a suitable microkernel. These microkernels are based on an outer-product formulation which targets the following result blocks ($M' \times N'$) in matrix C: $\{16, 12, 8, 4, 2, 1\} \times 3$. The size of 16×3 is determined by the AVX2 instruction set extensions which offers 16 four-element-wide vector registers called `ymm0-15`, assuming double precision. To store this C-result buffer, 12 ($4 \cdot 3$) out of the 16 registers are needed. Additional three registers hold broadcasts of the kth row of three columns of B. The remaining last register is a ring buffer, containing 4 entries, for 16 rows of column k of A. This leads to an optimal usage of all 16 available registers. In order to match the required values of M in case of SeisSol's kernels, a concatenated series of loops is generated to compute a $M \times 3$ result in C. A generated outer loop over N is used to form the final $M \times 9$ matrix multiplication.

As the vector register length in AVX2 is four, best efficiencies are expected for $M \; mod \; 4 = 0$. If this is not the case, we handle the remainder by calling half vector length or scalar instructions. However, the major performance hit in such a scenario is due to unaligned loads. Fig. 21.8 demonstrates this for convergence order 3 ($M = 10$) and 5 ($M = 35$). Here we compare three different implementations. The "no padding" case does not require additional storage and operates directly on unaligned data. "ld padding" pads the leading dimensions of the matrices to the next bigger multiple of 4, which is 12 in case of order 3 and 36 for order 5. This leads to only aligned load and stores since the used M is still 10 and 35, respectively. In case of "zero padding," the application has to actively store zeros in A for rows exceeding the defined M. This enables the code to run with a minimal number of instructions, so the performance increases even further. SeisSol has been rewritten to support zero padding throughout

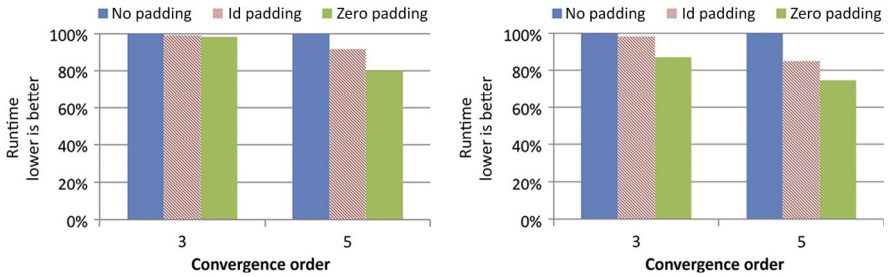


FIG. 21.8

Relative kernel performance measured on a single core of the Intel Xeon E5-2699v3 processor for convergence orders that result in a number of basisfunctions $B_{\mathcal{O}}$ which is not a multiple of four. The matrix pattern $B_{\mathcal{O}} \times B_{\mathcal{O}} \cdot B_{\mathcal{O}} \times 9 = B_{\mathcal{O}} \times 9$ is shown in the left plot, whereas the right one depicts the $B_{\mathcal{O}} \times 9 \cdot 9 \times 9 = B_{\mathcal{O}} \times 9$ shape.

the entire application for the best performance. Note that there is no memory consumption difference between "ld padding" and "zero padding."

DENSE MATRIX KERNEL GENERATION: AVX-512

The ideas discussed in the last section can be easily extended to use AVX-512. However, these new instructions offer several possibilities (e.g., fusing a memory broadcast as one operand into an arithmetic instruction) such that better performance can be achieved, especially in case of small GEMMs. Recall, we have twice as many and twice as long registers in AVX-512, so our AVX2 kernel would increase to 48×3 which can no longer be efficiently blocked to the sizes needed in SeisSol. Furthermore, the employed 2D-blocking is not optimal on Knights Landing. Due to Knights Landing's core being based on Intel's codenamed Silvermont architecture (see Chapters 2 and 4), the out-of-order pipeline is only two-issue wide with two FPUs in the backend. Therefore, any instruction that is not of arithmetic nature decreases the computational efficiency of the core. It's desirable to use a tall-and-skinny or short-and-wide blocking in the DGEMM kernel to only have one load instruction to A or B and then a series of Fused-Multiply-Adds (FMAs) which have a fused memory operand. As we have 32 registers available and we know that we always have a $N = 9$ in all matrix operations, we decided to work in all cases on all columns of B and C simultaneously. We take the route of "short-and-wide": we load 8 rows of A of column k into a register and then perform 9 FMA instructions which broadcast the kth row of all 9 columns on the fly. After having processed all columns of A/rows of B, we hold a 8×9 sub-matrix of C in 9 accumulator registers where they are stored back to all 9 columns of C.

This kernel is everything we need as building block. For convergence orders which result in a number of basis functions that are not a multiple of 8, we do not implement complicated remainder handling. Instead we could use AVX-512

masking registers to compute just the allowed values. That means all loads from A and all loads and stores referencing C are simply annotated by the masking register `k1` which can be initialized very cheaply at the beginning of a GEMM kernel. The mask variants of instructions do not run substantially slower than their unmasked counterparts, so only performance disadvantages because of unaligned memory accesses remain. However, as in case of AVX2 implementation, we rely on "zero padding" to ensure aligned, and therefore, fast memory accesses.

In order to obtain good performance on Knights Landing with its AVX-512 implementation, we need to apply four enhancements to the discussed microkernel of our small GEMM. They are required due to the fact that the enhanced Silvermont core still has limitations that need to be taken into consideration: (a) the FMA latency is 6 cycles, (b) only up to 16 bytes can be fetched per cycle from the instruction cache, (c) the memory pipeline of Knights Landing is in-order, and (d) there is no shared LLC.

Let us start with the simplest point, namely (d). This can be handled by adding prefetch instructions for the matrix operands of the next tetrahedron. We will discuss the impact of having these prefetch instructions for Knights Landing in the next section when covering the performance of the entire update kernel. (a), (b), and (c) need to be addressed by implementing the kernel very thoughtfully. The problem with (a) is that our innermost kernel consists of 9 FMA instructions which presumably run in throughput scenarios in 4.5 cycles as there are 2 VPU per out-of-order core. Since this is "faster" than the 6 cycles latency per VPU, we will run into reservation station over-subscription as the same nine registers (e.g., `zmm23-31`) will be reused in the next iteration of the microkernel. The solution is straightforward: we introduce a second temporary accumulator for C, `zmm14-22`, which is used in every other iteration. This ensures that the same register is only reused after at least 9 cycles. Before storing back to C we need to merge `zmm23-31` and `zmm14-23`; however the overhead in case of a larger K is minor. Issue (b) is problematic since we cannot afford to restructure our data in an optimal way as it is normally done for large GEMMs. We therefore have strided accesses (stride ldb) when reading B in broadcast fashion. Even for mid-order runs, ldb is bigger than 16, resulting in an offset larger than $128 \cdot 8$ byte between columns (AVX-512 offers a compressed displacement encoding which multiplies the offset by the datatype size). In such a case, the length of the FMA instruction increases from 7 to 10 bytes, and two instructions can no longer be fetched on a sustained basis from the instruction cache. However, the instruction size can be reduced to 8 byte per FMA if the x86 SIB (scale index base) addressing mode is utilized. Since we have spare general purpose registers and we need to express nine streams, we can load ldb to a register and use different base registers to assemble all nine column streams. Due to the scaling in the SIB mode, $\{1,2,4,8\}\times$ ldb can be added for free. The different base register hold pointers to the first, fourth, and seventh column of B. In order to fit into eight byte length, we would need to increase these pointers every 128th k by 1024 bytes, however in SeisSol this never happens as we have $M < 128$. Finally, (c) is addressed by pipelining the loads of rows per column k of A. This is easily doable as registers `zmm0-13` are still unused. To even hide the L2 latency of 17 cycles we use a 6-register ring-buffer of A column-vectors.

KERNEL PERFORMANCE BENCHMARKING

Before running benchmarks using the entire SeisSol code on the Intel Xeon E5 v3 processor and Knights Landing, we study the dense small GEMM performance out of the L1 cache. Additionally, we analyze the performance of the kernel running on a single core of the first generation Intel Xeon Phi coprocessor (codenamed Knights Corner). The details of the test systems are as follows:

Haswell (HSX) one Intel® Xeon® E5-2699 v3 processor with 18 cores, 1.9 GHz (running at AVX-base frequency), 64 GB of DDR4-2133.
Knights Corner (KNC) one Intel® Xeon Phi™ 7120A coprocessor in native mode with 61 cores, 1.24 GHz, 16 GB of GDDR5, one core reserved for OS.
Knights Landing (KNL) one Knights Landing preproduction processor with 68 cores, 1.2 GHz AVX-base core-clock, 1.7 GHz mesh-clock, 16 GB MCDRAM@7.2 GT, 96 GB DDR4-2400, cache mode=flat, cluster mode=quadrant, one core reserved for OS.

Fig. 21.9 summarizes the kernel performance for SeisSol, as pointed out above, most important kernels, $B_{\mathcal{O}} \times B_{\mathcal{O}} \cdot B_{\mathcal{O}} \times 9 = B_{\mathcal{O}} \times 9$ and $B_{\mathcal{O}} \times 9 \cdot 9 \times 9 = B_{\mathcal{O}} \times 9$. Comparing Knights Landing to the Intel Xeon E5 v3 processor shows efficiencies which are considerably lower. This is not related to the fact AVX-512 is used instead of AVX2 and the wider register set is less efficient or cannot be exploited that well. In fact, our AVX2 code runs roughly 3× slower on Knights Landing than the AVX-512 variant. The main reason for the lower peak efficiency is the two-issue-wide execution pipeline. Our inner most kernel (not accounting for load and stores to C and pointer advancing or loop structures) has in an unrolled fashion two loads of A and 18 FMA instructions. Therefore has a hard limit at 90% peak efficiency. Including all mentioned overheads we end up with roughly 80% as theoretical maximum. Even

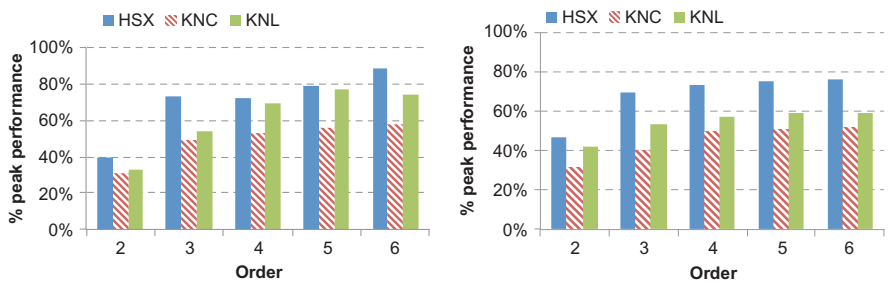


FIG. 21.9

Standalone matrix kernel performance running out of a hot L1 cache for Haswell, Knights Corner and Knights Landing. *Left*: kernel performance for $B_{\mathcal{O}} \times 9 \times B_{\mathcal{O}}$ matrix multiplication shapes; *right*: kernel performance for $B_{\mathcal{O}} \times 9 \times 9$ matrix multiplication shapes.

when considering 90% peak as maximum, our kernel runs at close to 90% of its attainable best performance which is the same number we measured on the Intel Xeon E5 system. The smaller kernel benchmarked (right plot) runs at lower performance as the A load pipeline and the additional accumulator cannot be efficiently used due to the very limited size of the operation ($k = 9$).

We also implemented a single-precision version of the kernels and LIBXSMM supports single precision as well. However, for high-order runs of SeisSol, double precision is necessary and hence the code generation works analogously, so we do not cover single precision here.

INCORPORATING KNIGHTS LANDING'S DIFFERENT MEMORY SUBSYSTEMS

As discussed earlier, SeisSol separates the wave propagation solver into two macro-kernels: the time kernel which is combined with the element local volume kernel and element-local part of the surface kernel, and the contribution of the face-neighboring elements. This offers opportunities for leveraging Knights Landing's two memory subsytems. This opportunity is due to different access frequencies to data structures in question and therefore varying bandwidth requirements. In the case of high-order simulations, the access frequency to Q_k, $\mathcal{B}_k$, or $\mathcal{D}_k$ and the element-local $A_k^{\xi_c}$, $A_k^{-,i}$ in the computation of the local contributions is very low, as the data causing the majority of the compute ($\hat{K}^{\xi_c}, K^{\xi_c}, F^{-,i}$ and temporarily buffers) can be cached in each tile. In contrast, computing the neighbor contributions, using $\mathcal{B}_{k_i}$ or $\mathcal{D}_{k_i}$ requires significantly more bandwidth than $A_k^{+,i}$ for higher orders as they are bigger but have the same access frequency.

These access patterns can be exploited to overcome potential size limitations of the 16 GB MCDRAM by placing the in-frequently accessed data structures in DDR4. As the bandwidth requirements of SeisSol decrease with an increasing order and the memory footprint increases with higher orders, this is a perfect coincidence. We are using the memkind library (see Chapter 3) to place different data structures in MCDRAM. This is demonstrated in Fig. 21.10. We implemented a custom memory allocator on top of memkind to allow compilations on installations without the library and to have a clean programmatic selection as to which memory type should be used through the memkind enum type. Note that the computational routines of SeisSol do not need to be modified as just a different pointer location is fed into them.

In summary, $\mathcal{B}_k$ and/or $\mathcal{D}_k$ are physically stored in MCDRAM, whereas $A_k^{\xi_c}, A_k^{-,i}, A_k^{+,i}, Q_k$ reside in the DDR4 portion of the address space for orders $\mathcal{O} = 5$ and $\mathcal{O} = 6$. Additionally, we hold all global matrices, $\hat{K}^{\xi_c}, K^{\xi_c}$, $F^{-,i}, F^{+,i,j,h}$, in MCDRAM as well, as we expect L2 cache evicts for higher orders, so we can leverage the MCDRAM's high bandwidth to mitigate cache miss penalties. For lower orders, two to four, we allocate more data structures in MCDRAM, in fact

```
namespace seissol {
namespace memory {
  enum Memkind {
    Standard = 0,
    HighBandwidth = 1
  };
}

#define PAGESIZE_HEAP 2097152
#define PAGESIZE_STACK 4096
#define MEMKIND_GLOBAL     seissol::memory::HighBandwidth
#define MEMKIND_TIMEDOFS seissol::memory::HighBandwidth
#define MEMKIND_CONSTANT seissol::memory::Standard
#define MEMKIND_DOFS       seissol::memory::Standard

void* seissol::memory::allocate(size_t i_size, size_t i_alignment,
                                enum Memkind i_memkind){

  void* l_ptrBuffer;
  bool error = false;

  /* handle zero allocation */
  if ( i_size == 0 ) {
    l_ptrBuffer = NULL;
    return l_ptrBuffer;
  }
#ifdef USE_MEMKIND
  if( i_memkind == 0 ) {
#endif
    if (i_alignment % (sizeof(void*)) != 0) {
      l_ptrBuffer = malloc( i_size );
      error = (l_ptrBuffer == NULL);
    } else {
      error = (posix_memalign( &l_ptrBuffer,
               i_alignment, i_size ) != 0);
    }
#ifdef USE_MEMKIND
  } else {
    if (i_alignment % (sizeof(void*)) != 0) {
      l_ptrBuffer = hbw_malloc( i_size );
      error = (l_ptrBuffer == NULL);
    } else {
      error = (hbw_posix_memalign( &l_ptrBuffer,
               i_alignment, i_size ) != 0);
    }
  }
#endif
  if (error) {
    logError() << "The seissol::memory::allocate failed.";
  }
  return l_ptrBuffer;
  }

 Examples how this allocator is used:
[...]
real* globalMatrixMem =(real*)(seissol::memory::allocate(
seissol::model::globalMatrixOffsets[seissol::model::numGlobalMatrices]
*sizeof(real), PAGESIZE_HEAP, MEMKIND_GLOBAL ));
[...]
m_internalState.dofs  = (real(*)[NUMBER_OF_ALIGNED_DOFS])
seissol::memory::allocate( (m_totalNumberOfCopyCells
+m_totalNumberOfInteriorCells)*sizeof( real[NUMBER_OF_ALIGNED_DOFS] ),
PAGESIZE_HEAP, MEMKIND_DOFS );
```

FIG. 21.10

Custom memory allocator which allows for a programmatically selection of the memory target.

order	Q_k	$\mathcal{B}_k, \mathcal{D}_k$	$A_k^{\xi_c}, A_k^{-,i}, A_k^{+,i}$	$\hat{K}^{\xi_c}, K^{\xi_c}, F^{-,i}, F^{+,i,j,h}$
2	MCDRAM	MCDRAM	MCDRAM	MCDRAM
3	MCDRAM	MCDRAM	MCDRAM	MCDRAM
4	DDR4	MCDRAM	MCDRAM	MCDRAM
5	DDR4	MCDRAM	DDR4	MCDRAM
6	DDR4	MCDRAM	DDR4	MCDRAM

FIG. 21.11

Placements for all orders and the different data structures of SeisSol; DDR4/MCDRAM denotes if a particular data structure is placed in DDR4/MCDRAM.

orders $\mathcal{O} = 2$ and $\mathcal{O} = 3$ have to run out of MCDRAM for the best performance. The table in Fig. 21.11 summarizes the used placements when running on Knights Landing in flat memory mode.

One challenge compared to the Intel Xeon E5 v3 processor when using Knights Landing is the absence of a shared LLC. Instead, Knights Landing features a distributed LLC which is implemented by a 2D mesh of up to 36 1 MB large slices of L2 caches. These segments are kept coherent by a distributed tag directory. This cache-layout can be harmful in SeisSol's neighboring contribution kernel for two reasons: (a) the 48 flux matrices $F^{+,i,j,h}$ approach (500 KB for order five) or even exceed the size (1.5 MB for order $\mathcal{O} = 6$) of a slice of L2 and (b) due to the unstructured and mesh-dependent access pattern to the flux matrices, the hardware prefetcher is not able to preload the needed data into the L2. Both factors cause a high rate of coherency communication which will result in high access latencies. However, this coherency traffic can be optimized by leveraging MCDRAM, where we still have plenty of bandwidth unused, especially when running high-order simulations. Therefore, we place several copies of the global matrices, one per two tiles, in MCDRAM. This ensures that the mesh traffic gets equally distributed and the access latency may not be limited by one coherency agent in the entire mesh holding the directory entries for one particular flux matrix. This procedure is further enhanced by incorporating prefetches into selected matrix kernels. We are using a modified version in the neighbors' contribution to preload the next $\mathcal{B}_{k_i}$ or $\mathcal{D}_{k_i}$. For the best performance, these prefetches are widely scattered throughout all eight involved matrix operations.

The results depicted in Fig. 21.12 show the benefits of the aforementioned tweaks, when running our performance proxy for the Mount Merapi scenario in flat memory mode coupled with quadrant cluster mode, using order $\mathcal{O} = 6$. This plot shows the scaling behavior of SeisSol's local contribution and neighbor element's contribution kernels. In the case of the latter one, we distinguish between a vanilla version and one incorporating the optimizations highlighted earlier. Additionally, we include the full application scaling. Duplication of data structures and prefetches roughly double the performance of the neighbor contribution and nearly perfect scaling can be achieved. In case of all operations, the biggest drop in performance is noticeable within a tile when using both cores. The reason for this is the shared

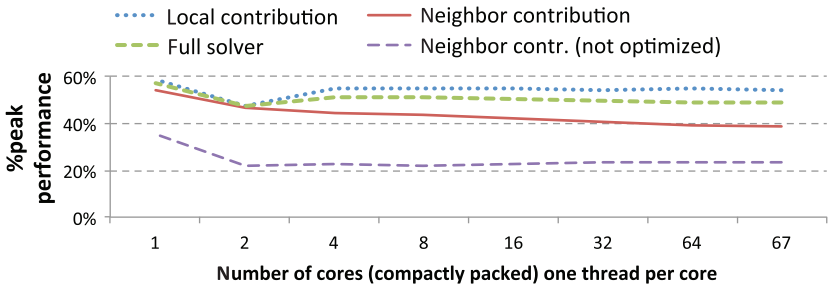


FIG. 21.12

Scaling of a setup with Merapi characteristics on Knights Landing using global time stepping measured by a performance proxy application for single-node SeisSol executions with errors of less than 1%. Shown is the separate performance of the element local contribution and the contribution of the face-neighboring elements and the full solver for order $\mathcal{O} = 6$. Additionally, we show the scaling of the neighboring elements' contribution to the surface kernel without our optimization for Knights Landing's mesh and distributed LLC.

L2 cache per tile which allows for reading one line per cycle and writing a half line per cycle. As the kernel computing the neighbor contribution reads more data per element (flux matrices, time integrated DOFs/time derivatives, flux solvers) than the local kernel, its performance drop is bigger. Fortunately, for higher orders, for example, $\mathcal{O} = 6$, the local contribution accounts for roughly 70% of SeisSol's overall runtime, and therefore the full application scaling is aligned with the calculation of the local contribution.

PERFORMANCE EVALUATION

This chapter evaluates performance of SeisSol in two different setups. The first setup simulates seismic wave propagation in Mount Merapi using a geometrical complex topography. Additionally, this wave propagation setup demonstrates algorithmic improvements due to our clustered LTS scheme. The second configuration is a setup of the 1992 Landers earthquake. In addition to topography data, this configuration includes the complex Landers fault system as internal boundary. Here, we solve a multiphysics problem, coupling dynamic rupture, and seismic wave propagation.

MOUNT MERAPI

Mount Merapi is the most active volcano in Indonesia and has erupted regularly since 1548. The Mount Merapi setup uses a tetrahedral mesh consisting of 1,548,496 elements. These elements are aligned to the surface topography, the spherical outflow boundary and the spherical material contrast inside the volcano. The seismic point

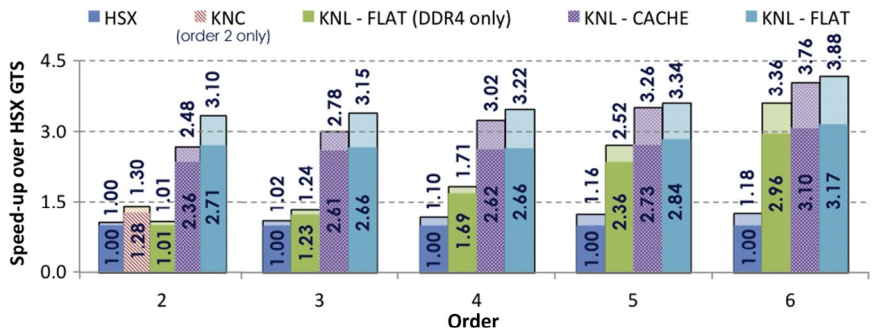


FIG. 21.13

Using global time stepping: normalized time-to-solution speed-up in the Mount Merapi scenario for Haswell, Knights Corner and Knights Landing over Haswell global time stepping. Stacked bars: number above the bar is with turbo, numbers on the bars are without turbo.

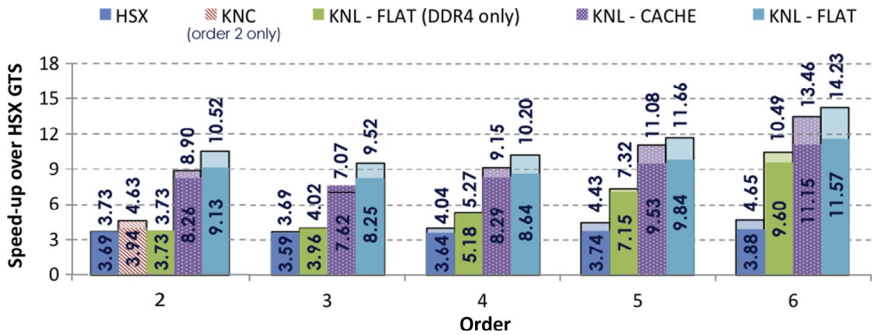


FIG. 21.14

Using rate-2 local time stepping: normalized time-to-solution speed-up in the Mount Merapi scenario for Haswell, Knights Corner and Knights Landing over Haswell global time stepping. Stacked bars: number above the bar is with turbo, numbers on the bars are without turbo.

source is located at mean sea level directly below Mount Merapi's peak. We show our performance results relative to Haswell when executing the Mount Merapi scenario using global time stepping (Fig. 21.13) and a LTS (Fig. 21.14) scheme which theoretically can speed-up the computations by about $4\times$. Since the mesh has a huge number of elements, the Knights Corner coprocessor cannot execute simulations for orders larger than two as the 16 GB GDDR5 memory does not offer enough space. In contrast, Knights Landing does not face such memory limitations as the MCDRAM is backed by a large DDR4 region. This memory can be leveraged by our out-of-core implementation, for example, for order $\mathcal{O} = 6$, the total consumed memory is 26 GB with 7.3 GB used in MCDRAM when running with GTS. If LTS is used instead, we see an increase to 30 GB total and 11 GB of MCDRAM. As we can see, only the memory footprint in MCDRAM grows. The reason is that in case of GTS every

element k only stores a buffer B_k for read-only access by face neighbors. In contrast to that, an element k' in LTS configurations might have to store a buffer $\mathcal{B}_{k'}$, or derivatives $\mathcal{D}_{k'}$, or both $\mathcal{B}_{k'}$ and $\mathcal{D}_{k'}$. As the effective usage of the flat memory mode required several changes to the source for SeisSol we also want to analyze the transparent usage of MCDRAM through the cache memory mode. We see that this ease of use comes at a very low price of $\approx$10% reduction compared to the flat memory mode results.

In addition to the base clock performance, we analyzed additional possible speed-ups over Haswell @1.9 GHz in Figs. 21.13 and 21.14 as shown by the upper numbers over the stacked bars. On Knights Corner, only a little step-up is measured when allowing up-to 1.33 GHz turbo clock if the power consumption permits. In contrast on Haswell, the turbo technology allows up to 2.6 GHz which is obviously not realizable due to the highly efficient kernels in SeisSol, but 10–15% improvement is possible. Knights Landing is able to clock up to 1.5 GHz if the power stays below the TDP power limit. This frequency is also not possible at a sustained level, and we therefore measure a 15–20% speed-up when using turbo. As a bottom line, we can conclude that Knights Landing can execute the Mount Merapi scenario up to 14.23$\times$ faster than the Haswell GTS baseline. This is 3.06$\times$ faster than the algorithmically optimized Haswell LTS version. Due to the huge memory footprint, a comprehensive comparison to performance on Knights Corner is not possible. Therefore, Knights Corner can offer only order 2 results for comparisons.

1992 LANDERS

1992 Landers was a magnitude 7.3 earthquake on Jun. 28, 1992, near Landers, California, which is about 104 miles (168 km) from Los Angeles. Our second setup simulates the 1992 Landers earthquake and employs a total of 466,574 tetrahedrons for spatial discretization. In this setup, the faces of our tetrahedral elements are aligned to the surface topography, the outflow boundary of the computational domain, and the internal boundary discretizing the fault system. The setup prescribes a stress field at the internal boundary, and rupture propagation is initiated via a nucleation patch. An example of this setup after 13 s of simulated time is illustrated in Fig. 21.15.

At this point we can only conduct global time stepping runs in case of dynamic rupture simulations. Fig. 21.16 depicts the GTS performance of our 1992 Landers setup. Due to dynamic rupture internal boundary conditions, this scenario is a multi-physics setup. We expect a slight performance decrease as the rupture process handling is highly scalar and adds overhead to the pure wave propagation runs (as the discussed Mount Merapi before). It is worth noting that the higher single-thread performance of Knights Landing is resulting in better performance compared to Knights Corner. Knights Landing is able to maintain 92% of the wave propagation speed-up over Haswell, whereas Knights Landing was limited to only 83%. Therefore the performance on Knights Landing for this complicated setup with irregular subparts is comparable to the characteristics observed on Haswell. In summary, the time-to-solution speed-up on Knights Landing for executing the 1992 Landers earthquake

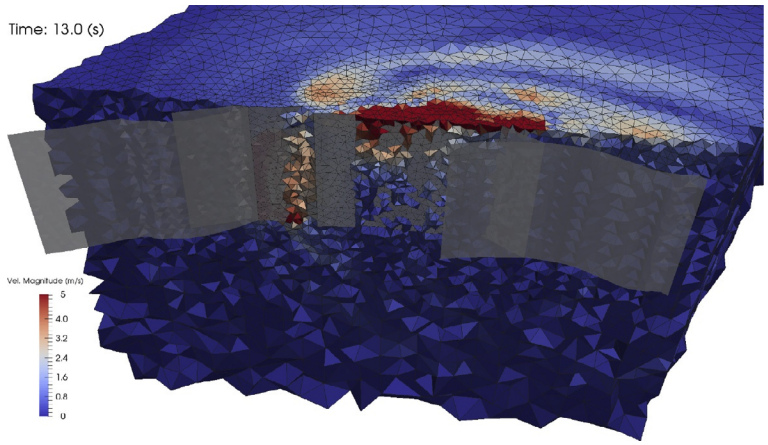


FIG. 21.15

Visualization of the velocity magnitude for the 1992 Landers setup after 13 s of simulated time. The structure of the fault system is shown in what looks like a ribbon across the visualization.

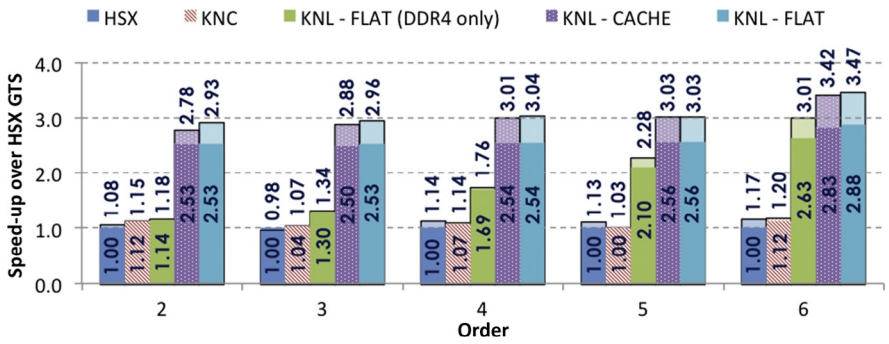


FIG. 21.16

Using global time stepping: Normalized time-to-solution speed-up over Haswell for Knights Corner and Knights Landing for the 1992 Landers scenario. Stacked bars: number above the bar is with turbo, numbers on the bars are without turbo.

simulations is 2.93–3.47× depending on the chosen order. We do not see a noticeable difference between flat memory and cache memory mode as the setup fully fits into MCDRAM.

SUMMARY AND TAKE-AWAYS

In this chapter we reviewed our end-to-end, optimization of the high-order finite element software SeisSol. The unique combination of the described individual steps allowed us to accelerate seismic simulations on the latest Intel architectures.

First, our chapter took a step back looking at the numerics. Here, we arranged our operators in an optimal way and derived a simple two-step scheme for the update of an element. The presented formulation maps the algorithmic complexity of ADER-DG, even in LTS configurations, to hardware properties. Afterwards, we described our high-level implementation of the ADER-DG integrators. As part of this description, we discussed data-reuse, encouraged by the arrangement of the operators. Here, we also presented our high-level software prefetching for the Knights Landing processor. By feeding our knowledge of future data accesses, performed in the ADER-DG scheme, to hardware, our software prefetches support concurrency of data transfers and computation.

In the next section, we covered the low-level implementation of our matrix kernels and thus moved closer to hardware. We achieved performance portability by using a code generation approach based on the libxsmm library. Our evaluation of the standalone matrix kernels showed high-order peak efficiencies above 50% on all considered architectures. Afterwards, we discussed our approach of exploiting high-bandwidth memory. Knights Landing's MCDRAM functioned as an example, and we derived memory placements of our heavy data structures based on functionality in the ADER-DG scheme and the order of convergence.

Finally, in the last section of this chapter, we evaluated the achieved, combined performance of our optimization using SeisSol for two different setups. Here, we presented results for seismic wave propagation in the stratovolcano Mount Merapi and a setup of the 1992 Landers earthquake. Our results show that we are able to transfer the observed, high peak efficiencies to the application level. This also holds when using our clustered LTS scheme for seismic wave propagation or running dynamic rupture workloads.

Summarizing take-aways and best known methods, we consider the following steps to be important when tuning an application:

- Step back and analyze the numerics. Bring together algorithmic complexity with hardware properties.
- After defining the application layout, define the needed operators. Optimize them. Use performance monitoring tools or performance models.
- Find an abstraction of the low-level kernels from the actual hardware: Guarantee performance portability to future architectures.
- Write macro kernels/proxy applications to mimic reasonable parts of the application with high accuracy. Verify/tune the application patterns using these proxies.
- Finally, combine all your efforts.

FOR MORE INFORMATION

- The version of SeisSol which was used to produce the results of this chapter can be found on github: https://github.com/SeisSol/SeisSol/releases/tag/201511.
- SeisSol kernels are generated by the LIBXSMM open source project which is hosted at github: https://github.com/hfp/libxsmm.
- Alexander Breuer, Alexander Heinecke, and Michael Bader. Petascale LTS for the ader-dg nite element method. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium, IPDPS16*, May 2016.
- Alexander Breuer, Alexander Heinecke, Michael Bader, and Christian Pelties. Accelerating seissol by generating vectorized code for sparse matrix operators. In *Parallel Computing - Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 347–356. IOS Press, Apr. 2014.
- Alexander Heinecke, Alexander Breuer, Sebastian Rettenberger, Michael Bader, Alice-Agnes Gabriel, Christian Pelties, Arndt Bode, William Barth, Xiang-Ke Liao, Karthikeyan Vaidyanathan, Mikhail Smelyanskiy, and Pradeep Dubey. Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC14*, pages 3–14, New Orleans, LA, USA, Nov. 2014. IEEE. Gordon Bell Finalist.
- Diana Guttman, Meenakshi Arunachalam, Vlad Calina, and Mahmut Taylan Kandemir, *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, Chapter 21, *Prefetch Tuning Optimizations*, pp 401–419, editors James Reinders and Jim Jeffers, Morgan Kaufmann, 2015, ISBN 978-0-12-803819-2.

CHAPTER

Weather research and forecasting (WRF)

22

What is new with Knights Landing in this chapter?

MCDRAM boosts performance in both cache and flat memory modes. AVX-512 offers higher performance than AVX2.

The Weather Research and Forecasting (WRF) Model is a next-generation intermediate scale (mesoscale) numerical weather prediction (NWP) system designed for both atmospheric research and operational forecasting needs. It is comprised of about a half million lines of code, predominantly Fortran. WRF employs a two-level domain decomposition, first over MPI ranks and then within each MPI rank over threads (OpenMP). This decomposition provides large amounts of hybrid MPI/OpenMP parallelism that is well suited to the many-core architecture of Knights Landing.

The results on Knights Landing are a notable example of the value of avoiding *offload* programming models. The relatively flat execution profile of WRF code is very limiting to any offload approach because offloading requires focused optimization efforts for each kernel targeted. In sharp contrast with offload models, all the WRF computation kernels benefit from running on Knights Landing, and they all benefit from the many cores, AVX-512, and MCDRAM.

WRF OVERVIEW

WRF was first released in 2000 and today has a worldwide community of registered users of over 30,000, from over 150 countries. WRF is used by the U.S. National Weather Service and weather services in several other countries. WRF has been ported to virtually every type of HPC system on the Top500 List of Supercomputers. WRF is supported and freely distributed by the National Center for Atmospheric Research (NCAR), they hold user workshops and tutorials, and coordinate the efforts of the large and distributed community of scientific contributors. WRF began in the late 1990s as a collaborative partnership principally among the NCAR, the National Oceanic and Atmospheric Administration (NOAA), the Air Force Weather Agency (AFWA), the Naval Research Laboratory, the University of Oklahoma, and the Federal Aviation Administration (FAA). WRF is currently in operational use at NCEP, AFWA, and other centers.

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00022-3

WRF is structured with both a data assimilation system and a software architecture facilitating parallel computation and system extensibility. WRF serves a wide range of meteorological applications across scales from tens of meters to thousands of kilometers. The WRF system contains two dynamical solvers, referred to as the ARW (Advanced Research WRF) core and the NMM (Nonhydrostatic Mesoscale Model) core.

WRF is used for NWP, climate studies, and atmospheric research. This includes forecasting hurricanes and other high-impact weather, wildfire modeling, wind, and solar renewable energy simulations.

WRF EXECUTION PROFILE: RELATIVELY FLAT

Computational cost in WRF is spread over a relatively flat profile comprising the *dynamics* module, a computational fluid dynamics finite-volume solver using explicit finite-difference approximation, and the *physics* module that represent atmospheric processes such as radiative transfer, convection and other moist processes, surface land/sea surface, and boundary layer effects. Roughly half the execution time is consumed by memory bandwidth limited routines in the *dynamics* module with the other half being dominated by floating-point intensive calculations in the *physics* module. The flatness of WRF's profile argues against *offload* programming for coprocessors, which is best suited for applications where a majority of an application's cost is over a small section of code comprising of a few thousand lines of code. Like Intel® Xeon® processors, Knights Landing supports industry standard, mature, and widely supported programming models (MPI, OpenMP, and a vector ISA). Unlike Knights Corner, Knights Landing nodes can be self-hosted avoiding the need for host/coprocessor data movement.

HISTORY OF WRF ON INTEL MANY-CORE (INTEL XEON PHI PRODUCT LINE)

WRF was first demonstrated on the prototype Knights Ferry at the Supercomputing Conference in Nov. 2011. WRF is the first NWP model to have been ported to and run in its entirety on Intel's MIC architecture.

Prior to running on Intel's MIC architecture, WRF had been run on systems with only modest numbers of cores per node, so optimization work to overcome barriers to thread scaling was important. Thread scaling analyses used the Intel® VTune™ Amplifier performance analyzer to identify and address scaling bottlenecks such as thread load imbalance. The Intel® Inspector tool proved especially useful at uncovering race conditions and other threading errors that surfaced on large numbers of threads.

WRF was designed to perform well on vector supercomputers that were still in use when WRF was being developed. WRF was designed with a structure-of-arrays

data organization generally favorable to vector instruction sets on modern processors today including AVX-512. Further detailed analysis of compiler vector reports and run-time analysis using the VTune performance analyzer yielded additional benefits from AVX-512 instructions on MIC.

A WRF/MIC working group of Intel architects, performance engineers, and WRF developers characterized and improved performance of the WRF model on the Knights Corner. These improvements to WRF were provided back to the community in Apr. 2013 with version 3.5 of WRF being the first community release of WRF with support for Intel Xeon Phi products. Support for the Intel Xeon Phi product, now Knights Landing, has continued with version 3.8 in spring 2016. Many of the same Intel® Xeon Phi™ enablement techniques employed by the authors for WRF were also used for NOAA's Nonhydrostatic Icosahedral Model, the subject of Chapter 2 in Volume Two of our Pearls books — see *For More Information* at the end of this chapter.

OUR EARLY EXPERIENCES WITH WRF ON KNIGHTS LANDING

In this chapter, we take a quick look at the performance of the full WRF model using the 12 km resolution Continental United States (CONUS12km) benchmark, a widely cited industry standard workload. Results are provided to explain the nearly 3× speedup observed over Knights Corner. In particular, the impact of the high-bandwidth 16 GB MCDRAM system on the Knights Landing node is discussed along with the benefits of the AVX-512 instruction set. Use of both all-to-all and quadrant cluster modes, with flat memory mode, is also discussed.

Since this book is about Knights Landing, we focused our attention on single node experiences. We measured performance of WRF on a single Knights Landing. Our particular preproduction Knights Landing had 68 cores running at 1.4 GHz, 16 GB MCDRAM memory, and 96 GB DDR memory.

WRF, when running the CONUS12km benchmark, fits entirely inside the 16 GB of MCDRAM. We use "numactl" (see Chapter 3) in flat memory modes to fully utilize the MCDRAM. We compare that with cache mode and we observed same performance for both modes. We use "Average Time Step" to calculate performance and ignore WRF initialization and I/O costs.

THE CONUS12km BENCHMARK

The 12 km Continental United States (CONUS) benchmark is 3 h of a 48-h forecast of a baroclinic cyclone and a frontal boundary that extended from north to south across the entire United States on Oct. 24, 2001. "12 km" refers to the spacing between grid cells in the horizontal dimensions of the WRF grid. The CONUS12km benchmark domain is 425 × 300 horizontal grid points with and 35 vertical levels. The CONUS12km benchmark is small by today's operational forecasting standards

but is useful here since it approximates the size of the domain that might be computed on one node of cluster at a forecast center.

The WRF domain is a three-dimensional grid representing the atmosphere over a rectangular geographic region of interest. As illustrated in Fig. 22.1, a WRF domain is parallelized using a two-level decomposition: first over subdomains called *patches* that are assigned to MPI tasks. Patches are further subdivided into smaller subdomains called *tiles* that are assigned one-to-one to threads within a multithreaded MPI task. WRF's default patch and tile configurations may be overridden at runtime either via the namelist.inut configuration file or using environmental variables.

On Knights Landing, we ran WRF CONUS12km with a single MPI rank containing 120 OpenMP threads assigned as two threads per Knights Landing core. Thus, each Knights Landing core works on two WRF tiles, and each hardware thread within that core works one WRF tile. We use 60 of the 68 available cores to evenly divide the 300-row south-north dimension into five rows per tile. Tests showed that using the full 68 cores gave no additional benefit because of load imbalance.

For the CONUS12km workload, tiles were decomposed on our Intel Xeon Phi system with the south-north (y) dimension by core and the west-east (x) dimension by thread. The threading that occurs in the x-dimension is over hyperthreads on a single core. Decomposing in the x-direction means the vectors get shorter. We require that the WRF tile size be a minimum of five grid cells in both the x- and y-direction. An example is shown in Fig. 22.2.

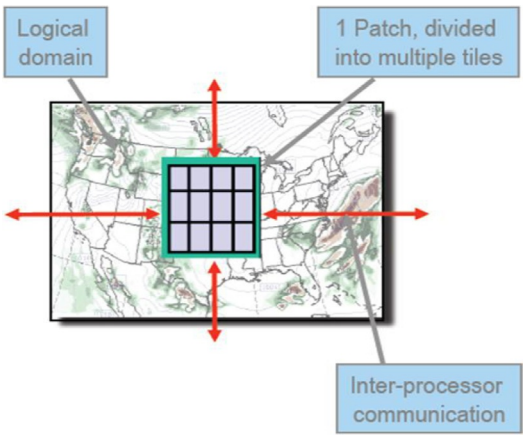


FIG. 22.1

WRF CONUS12km decomposition.

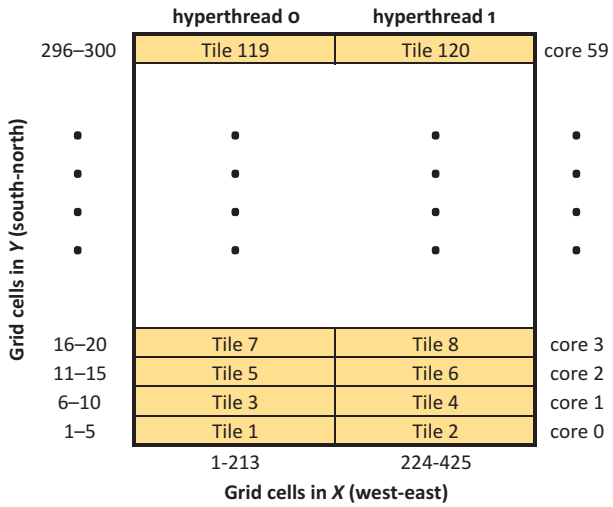


FIG. 22.2

WRF CONUS12km layout on Knights Landing.

COMPILING WRF FOR INTEL XEON AND INTEL XEON PHI SYSTEMS

This chapter uses the WRF V3.6 code (released 04/18/2014) from the NCAR WRF site. The mainly Fortran code was compiled with the Intel compilers in the Intel® Composer XE version 2015.2.164. OpenMP pragmas were used to express thread parallelism while MPI was utilized to express task parallelism. The executable was linked with the Intel® MPI Library version 5.0.2.044.

The WRF source code was compiled with the options:

```
-auto -ftz -fno-alias -fp-model fast=1 -no-prec-div
-no-prec-sqrt -FR -convert big_endian -auto
-align array64byte -openmp -fpp -O3
-fimf-domain-exclusion=15 -fimf-precision=low
```

The vector architecture "-xMIC-AVX512" option was added to generate AVX-512 instructions for the Intel Xeon Phi processor while the "-xCORE-AVX2" was added when compiling for our AVX2 Intel Xeon processors.

CONFIGURATION DETAILS

WRF was run on the Knights Landing computational node as a single MPI client that utilized 120 OpenMP threads, or two threads per core, on our experimental setup described in Fig. 22.3. The "KMP_PLACE_THREADS=60C,2T" specifies that the 120 threads will be placed two per core on 60 cores. Also note how stack size is set to a reasonable level using the KMP_STACKSIZE with the "-s unlimited" ulimit.

```
source /opt/intel/composer_xe_2015.2.164/bin/compilervars.sh intel64
#source /opt/intel/impi_5.0.3/bin64/mpivars.sh
ulimit -s unlimited
export KMP_STACKSIZE=512m
export KMP_PLACE_THREADS=60C,2T
export KMP_AFFINITY=compact,granularity=thread,verbose
export KMP_BLOCKTIME=infinite
export KMP_LIBRARY=turnaround
export WRF_NUM_TILES_X=2
export WRF_NUM_TILES_Y=60
numactl -m 1 mpiexec.hydra -np 1 ../wrf-Knights Landing .exe
```

FIG. 22.3

WRF CONUS12km environmental configuration script as used on Knights Landing.

WRF CONUS12km BENCHMARK PERFORMANCE

The CONUS12km workload is a computational-only benchmark. Therefore, we ignore the first timestep and measure performance for 149 time steps to exclude I/O and initialization costs from our measurements. Different time steps within the series cost differently; for example 5 of the 149 steps involve radiative transfer physics that costs 3.5 × more than other steps on Knights Landing for this workload. Performance comparisons are based upon average time step based on wall clock time on an unloaded computational node.

The WRF consus12km benchmark completely fits in the 16 GB MCDRAM on Knights Landing. CONUS12km has the same performance in flat memory mode (using numactl to place all data in MCDRAM) and cache memory mode. WRF CONUS12km has the same performance (0.24 s) in all-to-all and quadrant cluster modes, which suggests that the code is not highly latency sensitive.

Knights Landing is the best performing processor for the WRF CONUS12km benchmark. Knights Landing is 1.7 × faster than a dual socket Intel® Xeon™ E5-2697v4 codenamed *Broadwell* processor and 2 × faster than a dual socket Intel Xeon E5-2697v3 codenamed *Haswell* Processor. WRF CONUS12km is also 3 × faster on Knights Landing than Knights Corner. WRF CONUS12km performance comparison with other Intel Architecture machines is shown in Fig. 22.4.

MCDRAM BANDWIDTH

More bandwidth for Knights Landing from MCDRAM is the most important factor contributing to the superior CONUS12km WRF performance on Knights Landing. By profiling WRF CONUS12km on Knights Landing, we observe that WRF CONUS12km is a bandwidth bound application with bursts of high demands on bandwidth. Using the VTune "*memory access*" collection option, we can get a bandwidth profile for WRF as seen in Fig. 22.5. The five valleys in Fig. 22.5 are the five radiation time steps and the peaks are the nonradiation time steps mostly

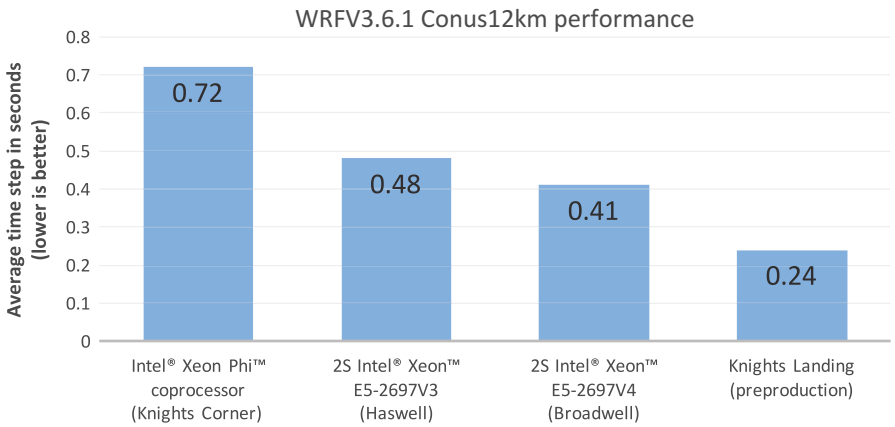


FIG. 22.4

Comparative WRF performance across different processors.

FIG. 22.5

DRAM and MCDRAM bandwidth utilization for all WRF timesteps.

executing the microphysics schemes. The first 24 timesteps are the nonradiation timesteps followed by the slower radiation timestep followed by the second set of 24 nonradiation timesteps followed by the slower radiation timestep and so on. Further zooming into the first set of valleys and peaks reveals another set of bandwidth valleys and peaks as seen in Fig. 22.6 (very hard on eyes without a big screen). The VTune graphs are more compelling and readable when using VTune; Fig. 22.7 shows the data in a form that is more interesting in print.

To understand the impact of bandwidth on Knights Landing, let us focus on the third timestep and understand its behavior with respect to bandwidth. The third WRF CONUS12km timestep takes 0.23 seconds to complete. By instrumenting the source code, and using SEP (Intel® Sampling Enabling Product), we observe that the total

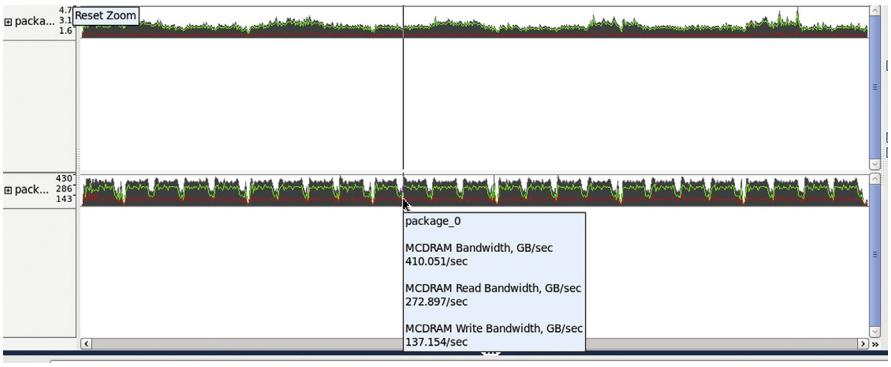


FIG. 22.6

DRAM and MCDRAM bandwidth utilization for first 24 timesteps of WRF CONUS12km.

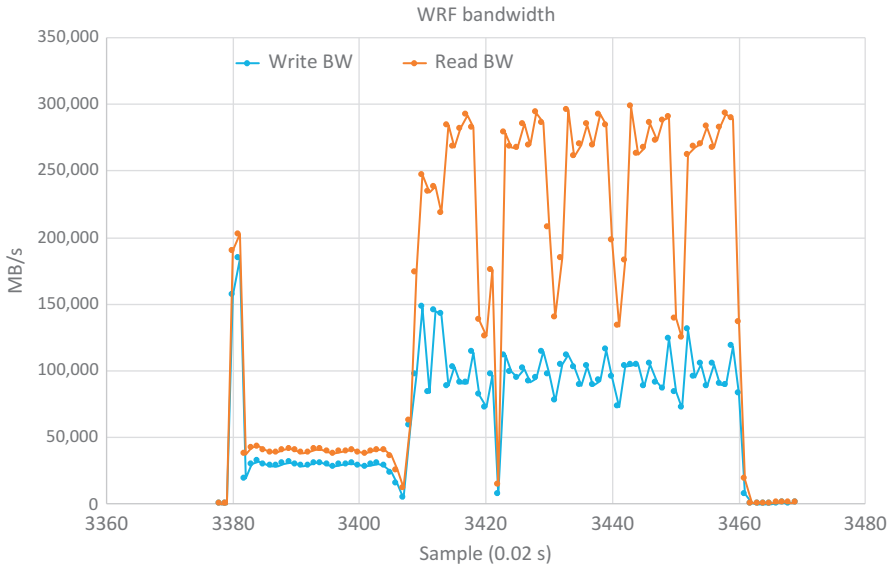


FIG. 22.7

Plot of data from prior two VTune graphs showing bandwidth demands over time.

bandwidth for the third timestep (duration of 0.23 s) is 293.02 GB/s composed of 213.52 GB/s read bandwidth & 79.50 GB/s write bandwidth. The read bandwidth to write bandwidth ratio is 2.7:1. These measurements were on preproduction hardware and may differ from production versions of Knights Landing.

VECTORIZATION: BOOST OF AVX-512 OVER AVX2

Knights Landing supports AVX-512 instructions, and we enable those instructions for WRF by compiling it with –xMIC-AVX512. These 512-bit instructions are wider than prior processors. To see the performance impact of AVX-512 instructions of WRF on Knights Landing, we compiled WRF with AVX2 (256 bit instructions) and AVX-512 and ran on the same Knights Landing, with the same 120 thread configuration as before, to observe the improvement. Fig. 22.8 shows that enabling AVX-512 instructions gives a 1.3 × performance improvement, which is significant from a performance perspective. Knights Landing has double the vector width per instruction compared to Broadwell, which supports AVX2 and not AVX-512 instructions.

On inspection with VTune, the code generated for WRF CONUS12km for Knights Landing vectorizes well. The hot functions for WRF CONUS12km on Knights Landing are shown in Fig. 22.9.

The Cycles Per instruction (CPI) in Fig. 22.9 are at a core granularity. Functions exhibiting a CPI up to 2.00 are actually good and expected behavior. Additional investigation is warranted for functions where CPI is greater than 2.00.

A reported Low Vectorization intensity would mean that we have work to do. However, a High Vectorization intensity is misleading because there may be significant vectorization not occurring since the counters on Knights Landing (and Haswell, and Broadwell) do not account for masking.

Using VTune, we measured the Vectorization intensity. *Vectorization intensity* or *Vector VPU Compute Percentage* measures the fraction of SIMD micro-ops that performed packed vector operations of any vector length and any mask. Unfortunately, Knights Landing counters (used by VTune) do not account for masking. Therefore, a vector instruction that only does one operation (7 double, or 15 single, operations being masked off) will count as fully vectorized. This is also true of earlier AVX and AVX2 support including that found in Haswell and Broadwell. The Vector Advisor (Chapter 10) discusses alternate ways to analyze vectorization accurately. Therefore, the higher vectorization intensity reported in Fig. 22.9 does not tell us that we are fully vectorized.

Instruction set architecture	Avg. time step in seconds
AVX2	0.32
AVX-512	0.24

FIG. 22.8

Performance of AVX-512 vs. AVX2 both executed on Knights Landing.

WRF function	Hotspot (%)	CPI rate per core	Vectorization intensity (%)
advect_scalar_pd	7	1.51	99.8
advance_uv	6	2.73	99.3
advect_scalar	5	2.11	100.0
advance_w	4	2.73	99.9
wsm52d	4	1.18	100.0
advance_mu_t	4	2.65	99.9
ysu2d	3	1.88	99.6
zero_tend	3	2.71	100.0
phy_prep	3	1.47	88.5
rk_update_scalar	3	3.50	100.0
calc_p_rho	2	4.59	100.0
sumflux	2	5.91	100.0
cal_deform_and_div	2	3.21	97.7
advect_w	2	0.86	99.6
coriolis	2	2.85	100.0
horizontal_pressure_gradient	2	2.86	100.0
calc_cq	2	2.41	100.0
small_step_prep	2	4.40	99.8
set_physical_bc3d	2	3.71	100.0
rhs_ph	2	2.82	99.8

FIG. 22.9

WRF CONUS12km hotspots and vectorization intensity on Knights Landing.

CORE SCALING

WRF is a highly parallel application with 95.7% of the code being parallel on Knights Landing and only 4.3% being serial on Knights Landing, excluding I/O and Initialization. Using VTune, we can calculate the fraction of the code which is parallel and the fraction which is serial as shown in Fig. 22.10.

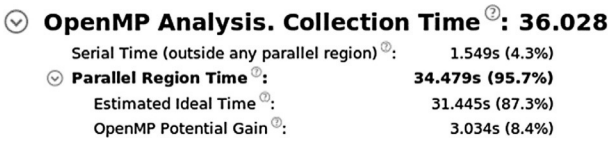


FIG. 22.10

WRF CONUS12km hotspots and vectorization intensity on Knights Landing.

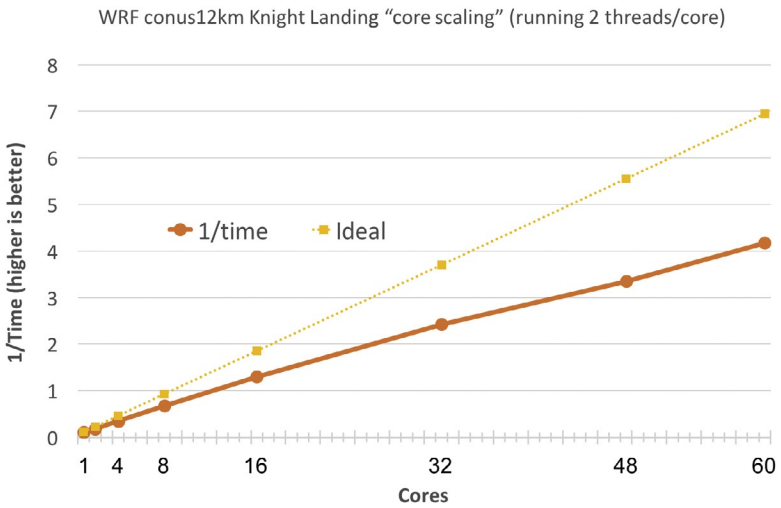


FIG. 22.11

WRF CONUS12km core scaling on Knights Landing vs. ideal scaling.

The Knights Landing architecture is well suited for an application like WRF. WRF CONUS12km has excellent core scaling as seen in Fig. 22.11. WRF CONUS12km scales well up to 16 cores, and the scaling curve starts to drift further away from ideal. More investigation is need to root-cause the specific reasons for the observed scalability. We observe a scaling of 36× on 60 cores versus a single core.

SUMMARY

Knights Landing does better than other processors because of its higher memory bandwidth (MCDRAM), superior vector capabilities (AVX-512), and higher thread counts (120 in our case).

Overall, we found that running on Knights Landing delivered a 3× performance improvement over the previous generation Knights Corner Intel Xeon Phi processors. While these are early results, we believe the MCDRAM memory subsystem contributed much to the performance improvement. Of course, this was possible only because the per-core AVX-512 vector units in Knights Landing were able to process the data at the rate possible due to the higher bandwidth.

FOR MORE INFORMATION

Here are some additional reading materials related to this chapter.

- The Weather Research & Forecasting Model, http://www.wrf-model.org.
- National Center for Atmospheric Research, https://ncar.ucar.edu/.

- National Oceanic and Atmospheric Administration (NOAA), http://www.noaa.gov/.
- Skamarock, W.C., Klemp, J.B., Dudhia, J., Gill, D.O., Barker, D.M., Duda, M.G., Huang, X.Y., Wang, W. and Powers, J.G., 2008. *A description of the Advanced Research WRF Version 3*, NCAR technical note, NCAR/TN–475+STR, Mesoscale and Microscale Meteorology Division. National Center for Atmospheric Research, Boulder, Colorado, USA.
- Michalakes, J., Dudhia, J., Gill, D., Henderson, T., Klemp, J., Skamarock, W., and Wang, W., 2005. *The Weather Research and Forecast Model: Software Architecture and Performance*. Proceedings of the Eleventh ECMWF Workshop on the Use of High-Performance Computing in Meteorology. Eds. Walter Zwieflhofer and George Mozdzynski. World Scientific, pp. 156–168.
- Iacono, M. J., Berthiaume, D., and Michalakes, J., 2014. *Enhancing Efficiency Of The RRTMG Radiation Code With GPU And MIC Approaches For Numerical Weather Prediction Models*. 14th Conf. on Atmospheric Radiation, Boston, MA, Amer. Meteor. Soc., p. 156.
- Henderson, T., Michalakes, J., Gokhale, I., and Jha, A., 2005. Chapter 2—Numerical Weather Prediction Optimization, in High-Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, ISBN 978-0-12-803819-2.

CHAPTER

N-Body simulation

23

In this chapter, we present performance optimization methods applicable to *N*-body simulations on Knights Landing. We discuss the optimization of arithmetic expressions, data structures, thread parallelism, and memory traffic on Knights Landing. Our results demonstrate that the *N*-body simulation, previously optimized for parallelism on Knights Corner, achieves most of the performance improvements available with Knights Landing without any code adaptation, but recompilation is necessary to take advantage of AVX-512.

What is new with Knights Landing in this chapter?

MCDRAM flat mode + numactl can be used to prove our optimizations eliminated our bandwidth problems; AVX-512 vectorization and transcendental capabilities.

PARALLEL PROGRAMMING FOR NONCOMPUTER SCIENTISTS

Computational applications are the primary tools of researchers and engineers across a broad spectrum of disciplines. However, for most professionals in fields outside of computer science, the task of following the latest trends in computing hardware has understandably lower priority than ongoing research in their own domain. This means that a large number of experts in computing-related domains who had received their education in programming 10 or more years ago are disadvantaged by not using the critical innovations introduced in processors over the last decade (e.g., short vector support, multi- and many-core processing, and heterogeneous computing).

This chapter focuses on an illustrative problem that should ignite the interest of noncomputer scientists in modern programming; it is compact enough to be easily understood but has broad appeal and real-world applications. Most importantly, the problem demonstrates the tremendous difference in performance one may experience by incorporating common modern programming methods into an application.

Adapting a computing application to modern processors in this way also prepares it for future generations of computing architectures—the application presented here was originally developed for Knights Corner coprocessors (see *For More Information* at the end of this chapter) but is able to leverage much of the computing power of the newer Knights Landing architecture (see Chapter 4) "out of the box."

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00023-5

STEP-BY-STEP IMPROVEMENTS

In this chapter, we discuss step-by-step the application of parallel programming to achieve very significant speed-up on this n-body simulation code. Our results are shown in Fig. 23.1. The performance measurements were obtained for $N = 1048576$, and we report the average of 8 out of 10 timesteps (the first two time steps are omitted to account for thread initialization overheads). Fig. 23.2

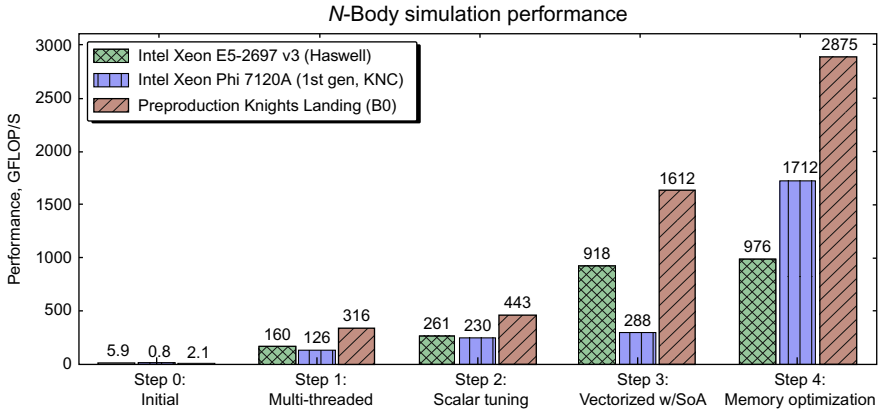


FIG. 23.1

Performance measurements step-by-step. Performance is measured in GFLOP/s (higher is better) and is calculated as $20 \times 10^{-9} N^2 / T$, where N is the number of particles, T is the measured duration of a time step (in seconds), the factor 20 reflects a convention widely adopted in literature on N-body simulations that each particle-particle interaction requires 20 floating-point operations, and 10^{-9} converts FLOP/s to GFLOP/s.

	Packages (sockets)	Cores/ package	Clock speed	Memory	OS
Intel Xeon E5-2697 V3 processor	2	14	2.7 GHz	128 GB DDR3	Centos 6.5
Intel Xeon Phi 7120A coprocessor (Knights Corner)	1	61	1.2 GHz	16 GB DDR4	Centos 6.5
Knights Landing (flat memory mode, quadrant cluster mode)	1	64	1.3 GHz	16 GB MCDRAM + 96 GB DDR4	CentOS 7.1

FIG. 23.2

System configurations utilized for results in this chapter.

shows the system configurations utilized for results in this chapter. Knights Landing was run in memory mode = flat, and cluster mode = quadrant, with memory allocation in MCDRAM for these results. Later, we will show an interesting test for memory-bound vs CPU-bound behavior that works by setting the default allocation to the MCDRAM or DDR4 (see Section "Impact of MCDRAM on Performance"). Additionally, we have performed tests with memory mode = cache and with cache mode = all-to-all and found that these modes lower performance of the fully optimized code by up to 3%. Results of these tests are not included in this chapter.

On all systems, the code was compiled with the Intel C++ compiler version 16.0.1.150. To compile the code for the Intel Xeon E5 v3 processor, we used the compiler argument "–xHost" (compilation was done on the same system as where benchmark was run), for Knights Corner we used "–mmic" argument and for Knights Landing the "–xMIC–AVX512" argument.

N-BODY SIMULATION

The *N*-body problem is the task of predicting how *N* point masses in free space in a closed system move in response to gravitational or electrostatic forces. For the 2-body problem, a general solution in terms of algebraic functions is well known. However, for solving the *N*-body problem with *N* greater than 2, computation is the only feasible approach.

PHYSICS BACKGROUND

The *N*-body problem has applications in multiple scientific domains (e.g., planetary systems and galaxy formation in astrophysics or the dynamics of molecular systems, crystals, and plasmas in microphysics), with bodies of different scales (e.g., planets or electrons/protons) playing the roles of the point masses. While different domains will have different physics, the underlying mathematical relationships and fundamental computing kernels are common between them.

MATHEMATICAL MODEL

The gravitational force $\vec{F}_i$ acting on a particle i due to its interaction with all other particles may be expressed in vector form as

$$\vec{F}_i = \sum_{j \neq i} \frac{M_i M_j \left(\vec{R}_j - \vec{R}_i\right)}{\left|\vec{R}_j - \vec{R}_i\right|^3}, \vec{F}_i = \sum_{j \neq i} \frac{M_i M_j \left(\vec{R}_j - \vec{R}_i\right)}{\left|\vec{R}_j - \vec{R}_i\right|^3}$$

where $\vec{R}_i$ is the location vector of the ith particle and M_i is its mass. Similarly, the electrostatic force acting on a particle due to all other particles in the system is

$$\vec{F}_i = \sum_{j \neq i} \frac{Q_i Q_j \left(\vec{R}_j - \vec{R}_i\right)}{\left|\vec{R}_j - \vec{R}_i\right|^3}$$

where Q_i is the electric charge of the ith particle. In both expressions,

$$\left|\vec{R}_j - \vec{R}_i\right| = \sqrt{\left(R_{j,x} - R_{i,x}\right)^2 + \left(R_{j,y} - R_{i,y}\right)^2 + \left(R_{j,z} - R_{i,z}\right)^2}$$

Regardless of the nature of the forces, the motion of particles is governed by the equation

$$M_i \ddot{\vec{R}}_i = \vec{F}_i$$

This second order ordinary differential equation assumes that the coordinate vector is a function of time: $\vec{R}_i \equiv \vec{R}_i(t)$, and the force vector $\vec{F}_i \equiv \vec{F}_i(t)$ is calculated using the coordinate vectors at the corresponding instant t. Given initial conditions on particle coordinates, $\left\{\vec{R}_i(0), i = 1 \ldots N\right\}$, and velocities $\left\{\dot{\vec{R}}_i(0), i = 1 \ldots N\right\}$, the N-body problem has a unique solution.

TIME COMPLEXITY

Because the force equation has $N - 1$ terms and there are N such equations, the computation of forces has asymptotic time complexity of $O(N^2)$. An algorithm that uses the exact expression for interaction forces is called the *direct* N-body algorithm.

For astrophysical applications, where N often has value in the hundreds of billions, direct computation of the forces is impractical. In this case, approximations may be used where remote parts of the gravitational system are approximated (e.g., with the multipole expansion). In the case of electrostatic interaction, where there are generally equal amounts of positive and negative charge in the system, the physics of interaction is actually simplified due to shielding. Effectively, shielding allows one to neglect, or approximate with fast-decaying expressions, the forces acting from particles beyond a certain shielding distance. Efficient N-body simulations based on these approximations have $O(N \log N)$ complexity.

While practical methods for *algorithmic* acceleration of the N-body simulation are essential in real-life applications, we do not include them here. For pedagogical reasons, we stick with the direct N-body algorithm and its $O(N^2)$ time complexity. The reader is reminded that the optimization of algorithms with better time complexity is generally more challenging. At the same time, the optimization methods demonstrated here (multithreading, scalar tuning, and spatial and temporal data access locality) are general and applicable to more advanced algorithms.

EVOLUTION IN TIME

For each particle, the equation of motion is an initial condition problem for a second order ordinary differential equation. It may be transformed into a system of two first-order differential equations, one for position vector $\vec{R}_i(t)$ and the other for velocity

vector $\vec{V}_i(t) \equiv \dot{\vec{R}}_i(t)$. The resulting system may be solved numerically with a variety of methods; of which we choose the easiest: the forward Euler method. Given a time step $\triangle t$, one may approximate the evolution of position and coordinates as follows:

$$\vec{V}_i(t+\triangle t) \approx \vec{V}_i(t) + \frac{1}{M_i}\vec{F}_i(t)\triangle t$$

$$\vec{R}_i(t+\triangle t) \approx \vec{R}_i(t) + \vec{V}_i(t+\triangle t)\triangle t$$

In a realistic calculation, one may choose to get better asymptotic accuracy by using, for example, the Verlet method or a high-order Runge-Kutta method, or, depending on the needs of the application, a backward (implicit) method to improve stability for large steps.

The usage of a simple integration scheme is easily justified: even a complex solver would typically have lower complexity than the force calculation; for a large number of particles, the solver implementation has little impact on overall performance. Furthermore, we have found that with single precision and with $\triangle t = 10^{-2}T$, the simple integration scheme reproduces the exact analytical solution of the 2-body problem with less than 3% error for as many as 30 orbits (here T is the orbital period).

OPTIMIZATION

Throughout the rest of this chapter, we express the performance of each new implementation in terms of GFLOP/s, derived using the expression:

$$P = \frac{20 \cdot N^2 \cdot 10^{-9}}{T_w} \text{GFLOP/s}$$

where T_w is the of execution time of the function `MoveParticles`, measured in seconds. We assume that every particle-particle interaction requires 20 floating-point operations, an approximation widely adopted in the available literature. Note that depending on the availability of operations such as fused multiply-add (FMA) and reciprocal square root in hardware and on the specifics of an implementation, the actual number of processor instructions issued may vary (see the reference to the *High-Performance Parallelism Pearls* chapter in *For More Information*).

INITIAL IMPLEMENTATION (OPTIMIZATION STEP 0)

We define a container for a single particle as a structure:

```
Struct ParticleType { float x, y, z, vx, vy, vz; };
```

and define an array of these structures to contain all particles:

```
ParticleType* P = new ParticleType(N);
```

Such a data structure is a logical starting point, because a single particle is the building block of all the information contained in the simulation.

```
void MoveParticles(int N, ParticleType* p, float dt) {
  // Loop over particles that experience force
  for (int i = 0; i < nParticles; i++) {
    // Components of the gravity force on particle i
    float Fx = 0, Fy = 0, Fz = 0;
    // Loop over particles that exert force
    for (int j = 0; j < nParticles; j++) {
      // Avoid singularity and interaction with self
      const float softening = 1e-20;
      // Newton's law of universal gravity
      const float dx = particle[j].x - particle[i].x;
      const float dy = particle[j].y - particle[i].y;
      const float dz = particle[j].z - particle[i].z;
      const float drSquared  =
                    dx*dx + dy*dy + dz*dz + softening;
      const float drPower32  = pow(drSquared, 3.0/2.0);
      // Calculate the net force
      Fx += dx / drPower32;
      Fy += dy / drPower32;
      Fz += dz / drPower32;
    }
    // Accelerate particles in response to the gravity force
    particle[i].vx += dt*Fx;
    particle[i].vy += dt*Fy;
    particle[i].vz += dt*Fz;
  }
  // Move particles according to their velocities
  for (int i = 0 ; i < nParticles; i++) {
    particle[i].x  += particle[i].vx*dt;
    particle[i].y  += particle[i].vy*dt;
    particle[i].z  += particle[i].vz*dt;
  }
}
```

FIG. 23.3

Initial *N*-body simulation implementation.

Fig. 23.3 contains the initial routine that implements the force equation and the forward Euler scheme for time evolution. The small floating-point number added to the denominator of the force equation is a "softening factor," which serves two purposes:

1. It prevents floating-point overflow if two particles approach each other too closely, and
2. When $j = i$, the computation can proceed without a floating-point exception and naturally results in the force being equal to 0. As a consequence, the code correctly handles the special case of $j = i$ without a performance-damaging branch.

THREAD PARALLELISM (OPTIMIZATION STEP 1)

The code of Fig. 23.3 delivers approximately the same performance on a modern dual-core notebook computer as on a cutting-edge recent generation Intel Xeon E5 v3 server with 28 cores across two sockets (two *packages* in OpenMP terminology); on a highly parallel platform like Knights Landing, its performance is even more disappointing initially.

The primary reason for the inability of this code to leverage powerful processors is obvious even without specialized tools. The code uses only one thread for computation, and, as a consequence, it will not take advantage of the multiple processing cores available in advanced processors.

To distribute the calculation across multiple cores, we need a parallel framework, and we choose OpenMP. We could parallelize the innermost *j*-loop, but it would be unwise due to the high fork-join synchronization costs required for each *i*-iteration, so this leaves the outer *i*-loop as the best candidate for parallelism.

Enabling parallelism with OpenMP in our implementation of the *N*-body simulation is trivial, requiring just one line of code before the *i*-loop as shown in Fig. 23.4.

In addition, we tune the execution environment by preventing OpenMP thread migration across cores by setting the environment variable `KMP_AFFINITY` as follows:

```
export KMP_AFFINITY=compact,granularity=fine
```

This works on all tested platforms and benefits performance by improving data access locality. When OpenMP threads move from one core to another, they lose the data they had stored in the L1 and L2 caches, which incurs a performance penalty. Setting the OpenMP thread affinity pins threads to their respective logical processors placing threads with adjacent numbers on adjacent logical processors or cores. This maintains the locality of cache access.

The speedup, machine-by-machine, brought about by multithreading is enormous: 27× on 28 cores from Intel Xeon processors; 160× on a 61-core Knights Corner coprocessor; and 150× on a 64-core Knights Landing processor. Speedup equal to the number of processing cores is the ideal result for a compute-intensive application such as this one. On Knights Corner coprocessors, the speedup is double the number of cores because a single thread per core could only issue a floating-point operation every other cycle. Knights Landing does not have the same limitation. On Knights Landing, the speedup is definitely super-linear. We assume this is either due to the usage of multiple threads per core, or because with the utilization of all cores, a greater amount of aggregate L2 cache became available.

```
  // Loop over particles that experience force
#pragma omp parallel for schedule(guided)
  for (int i = 0; i < nParticles; i++) {
    // … Rest of code without change
  }

  // Outside the parallel region:
  // Move particles according to their velocities
  // O(N) work, so using a serial loop
  for (int i = 0 ; i < nParticles; i++) {
    particle[i].x  += particle[i].vx*dt;
    particle[i].y  += particle[i].vy*dt;
    particle[i].z  += particle[i].vz*dt;
  }
```

FIG. 23.4

Parallelization with OpenMP.

There are clear indications that a large fraction of performance is left untapped in all of the platforms. First, the fact that the Knights Corner coprocessor delivers less performance than the Intel Xeon platform despite its higher theoretical peak suggests a suboptimal implementation. Second, we can gauge the degree to which the code is suboptimal by examining the performance expressed as a percentage of this theoretical peak. The Knights Corner coprocessor has a theoretical peak of 2400 GFLOP/s in single precision—but delivers just over 120 GFLOP/s—and the Knights Landing processor has a theoretical peak of 4600 GFLOP/s but delivers just under 400 GFLOP/s. Comparing measured performance to the theoretical peak has caveats: the latter is based on the assumption that FMA instructions are executed every cycle, while the actual application has some standalone additions and multiplications and also computes the power function. Even though the target performance for the *N*-body code may be lower than the FMA-based estimates by a factor of 2, these initial results have more than an order of magnitude shortage of delivered performance, which prompts further investigation.

SCALAR PERFORMANCE TUNING (OPTIMIZATION STEP 2)

The optimization report produced by the Intel compiler is an invaluable tool for performance tuning and can be generated by using the compiler argument "-qopt-report=n" with n specifying verbosity. The report contains information about each function and loop in the compiled file, and a programmer can systematically identify problem areas by looking for common "red flag" warnings, which include nonvectorized loops; gather/scatter or nonunit stride access memory; unaligned accesses; multiversioned loops; vectorized loops with peeling; type conversion operations; and imperfect loop nesting.

An invaluable tool for performance tuning is the "-qopt-report=n" optimization report from the Intel compilers.

We set verbosity to 5 and focus on the innermost *j*-loop (innermost loops are of the highest importance because they are executed the greatest number of times) and find this statement:

```
remark #15417: vectorization support: number of FP up converts: single
precision to double precision 2 [ nbody-step-1.cc(42,30)
```

Code line 42 referred to by the listing reads

```
const float drPower32 = pow(drSquared, 3.0/2.0);
```

In C++, programmers have come to expect standard tools to use function overload by argument type (i.e., that pow(x, a) invokes an implementation of the power function with precision matching the types of x and a). However, the Math library functions are not overloaded in the global namespace, so pow(x, a) with single-precision arguments in fact invokes a double-precision function. To fix the problem, we should rewrite the expression either by using the single-precision function name:

```
const float rr  =
       1.0f/sqrtf(dx*dx + dy*dy + dz*dz + softening);
const float drPower32  = rr*rr*rr;

// Calculate the net force
Fx += dx * drPower32;
Fy += dy * drPower32;
Fz += dz * drPower32;
```

FIG. 23.5

Strength reduction and precision control.

```
const float drPower32 = powf(drSquared, 3.0f/2.0f);
```

or by accessing the overloaded power function in the namespace std:

```
const float drPower32 = std::pow(drSquared, 3.0f/2.0f);
```

Another often ignored convention is that floating-point literal constants in C and C++ are interpreted as double-precision numbers, and the suffix -f is required to treat them as single-precision numbers.

It may look like a contrived imperfection in the original code, but this optimization improves performance and removes the type conversion warning from the optimization report, an important reminder to look carefully at what is actually being computed, and not to trust "common knowledge" adopted by repetition.

In this particular case, the use of the power function is not justified, and we can replace it with an approximate equivalent: a reciprocal square root (which is implemented in Intel processors as a high-throughput vector instruction) followed by three multiplications. We can similarly replace the three divisions with three approximately equivalent multiplications (as shown in Fig. 23.5). This optimization is known as strength reduction.

In addition to function tweaks, there are opportunities to trade arithmetic accuracy for performance using compiler argument "–fp–model fast=2," which allows more aggressive compiler optimization and requests low-accuracy implementations of transcendental functions, resulting in additional speedup.

While the performance impact of each optimization technique: precision control (the –f suffix), strength reduction (using the reciprocal square root), and optimization level (the "–fp-model fast=2" option) may be illustrated separately, we report only the net result. On all platforms, these techniques combined lead to a performance increase of almost a factor of 2.

VECTORIZATION WITH SOA (OPTIMIZATION STEP 3)

As we are still an order of magnitude behind the target performance values set by the theoretical peak performance estimates, we continue to explore the optimization report for more clues. The next red flag is raised by the statements:

```
remark #15460: masked strided loads: 3
remark #15415: vectorization support: gather was generated for the
variable particle: strided by 6 [ nbody-step-2.cc(36,24) ]
```

A total of three reports like the above point us to code lines 36, 37, and 38 where expressions like these are executed:

```
const float dx = particle[j].x - particle[i].x
```

Before running the subtract operation, the processor must load the data values from the main memory (or cache) into the vector register. In the case of the expression `particle[j].x`, consecutive values of *x* for different values of *j* are spaced by a stride of 6. A better solution would be to have consecutive values of *x*, *y*, and *z* (as well as *vx*, *vy*, and *vz*) placed sequentially in memory. This optimization is illustrated in Fig. 23.6.

To remedy the situation, we introduce a new data structure:

```
Struct ParticleSetType{float *x, *y, *z, *vx, *vy, *vz;};
```

Instead of storing an array-of-structures, we now store a structure-of-arrays (SoA), enabling unit-stride access to the particle data. A programmer well trained in the pre-SIMD era may rightfully criticize the new approach for the lack of object-oriented thinking. Indeed, this is where the difference in mindset of programming for performance and programming for good style becomes apparent; though potentially confusing at first, using the optimized data structure brings about a significant performance improvement of factors up to 4 on all platforms except for Knights Corner where the speedup is 25%. This is likely because at our problem size, the Knights Corner performance was suppressed due to inefficient data reusage in caches. Indeed, we have confirmed (not published here) that for smaller problem sizes, Knights Corner performance also experienced a boost due to SoA by a small factor.

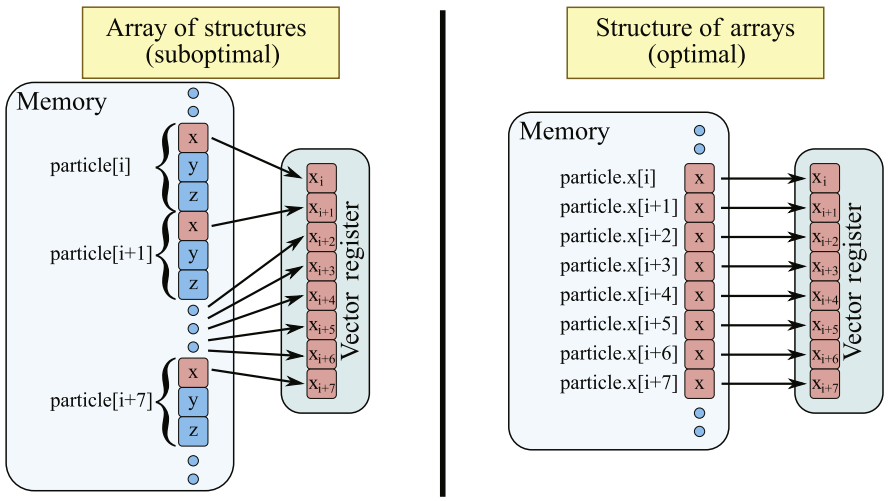


FIG. 23.6

Benefit of a structure-of-arrays (SoA) over array-of-structures (AoS). For simplicity, velocity components are not shown.

We also allocate data containers on a 64-byte boundary using the special allocator `_mm_malloc` and place the compiler hint

```
#pragma vector aligned
```

before the innermost loop. As a result, the compiler assumes aligned access to the particle data. The need for this optimization is apparent from the optimization report: the compiler reports creating peel loops, which is unnecessary if the data containers are aligned.

Finally, to get all benefits of vectorization, we used the compiler hint

```
#pragma omp simd
```

to vectorize the update to particle coordinates. According to the optimization report, this loop was not vectorized before the addition of this hint.

MEMORY TRAFFIC (OPTIMIZATION STEP 4)

By this point, we have achieved two orders of magnitude speedup with multithreading and an additional factor of 5–9 × increase in performance from scalar tuning and vectorization improvements. We are operating at roughly 50% of the theoretical peak performance on Knights Corner and Knights Landing. How can we tell whether this is sufficient, or if additional tuning is required?

First, we compile the code with the arguments "`-no-vec -no-simd -no-openmp-simd`" and compare performance to the vectorized code. This test reveals that the speedup due to vectorization is significantly less than the SIMD width (8 on the Intel Xeon processor and 16 on Intel Xeon Phi products), suggesting that data traffic is not efficient enough to saturate the vector processing units. Second, we study the change in performance behavior as we vary the problem size. This test reveals that performance decreases (on all platforms) for large problem sizes and again points to a need to optimize memory traffic.

The problem with memory traffic becomes apparent when we analyze the *j*-loop in the application, which reads three 4-byte quantities for each particle (x, y, and z coordinates). For values of N in the hundreds of thousands, this amounts to Megabytes of data accessed in each loop. Compared to the size of the L2 cache per core (256–512 kB on our platforms), and accounting for the fact that each core hosts multiple (2–4) threads, we can easily see that the data set accessed by the inner loop does not fit into this cache. As a result, memory traffic will "spill" into the Last-level cache (LLC) on the Intel Xeon processor, or into the MCDRAM cache or the main memory on Knights Landing.

To improve the situation, we must reduce the amount of data processed in the innermost loop. Two classes of techniques are available for this: loop tiling (which, in turn, can be implemented as cache blocking or as unroll-and-jam) and cache-oblivious recursion. For simplicity, we choose loop tiling and, considering the compiler's suggestion to permute loops, we perform an unroll-and-jam implementation. This technique requires two steps: (1) strip-mining the outer *i*-loop and (2) permuting the middle

and the inner loop. Vectorization in this case will be moved to the innermost loop, which is now in *i*. See Fig. 23.7 for the resulting code.

This version of the code reads data for the particles that fit into the vector register, and this data will be used a total of *N* times for each value of *j*. Therefore, for each floating-point number read from the LLC, MCDRAM or from main memory, this

```
#ifdef __MIC__
  const int tileSize = 16;
#elif KNLTILE
  const int tileSize = 16;
#else
  const int tileSize = 8;
#endif
  // Loop over particles that experience force
#pragma omp parallel for schedule(guided)
  for (int ii = 0; ii < nParticles; ii += tileSize) {
    // Components of the gravity force
    // on particles ii through ii+tileSize
    float Fx[tileSize], Fy[tileSize], Fz[tileSize];
    Fx[:] = Fy[:] = Fz[:] = 0.0f;
    // Avoid singularity and interaction with self
    const float softening = 1e-20f;
    // Loop over particles that exert force
#ifdef KNLTILE
#pragma unroll(tileSize)
#endif
    for (int j = 0; j < nParticles; j++) {
      // Loop within tile over particles that experience force
#pragma vector aligned
      for (int i = ii; i < ii + tileSize; i++) {
         // Newton's law of universal gravity
         const float dx = particle.x[j] - particle.x[i];
         const float dy = particle.y[j] - particle.y[i];
         const float dz = particle.z[j] - particle.z[i];
         const float rr =
             1.0f/sqrtf(dx*dx + dy*dy + dz*dz + softening);
         const float drPowerN32  = rr*rr*rr;
         // Calculate the net force
         Fx[i-ii] += dx * drPowerN32;
         Fy[i-ii] += dy * drPowerN32;
         Fz[i-ii] += dz * drPowerN32;
      }
    }
    // Accelerate particles in response to the gravity force
    particle.vx[ii:tileSize] += dt*Fx[0:tileSize];
    particle.vy[ii:tileSize] += dt*Fy[0:tileSize];
    particle.vz[ii:tileSize] += dt*Fz[0:tileSize];
  }
```

FIG. 23.7

Tiled *N*-body code.

tiled code will perform 20*(SIMD width)/3 ≈ 100 floating-point operations, which is generally sufficient to saturate the vector processing units.

The performance impact of this technique is small on the Intel Xeon processor, around 6%, due to the large LLC symmetrically shared between all cores. However, on Intel Xeon Phi products, we achieve an improvement of 500% (really!) on Knights Corner and 80% on Knights Landing.

Compared to cache-oblivious recursion, loop tiling is not a highly portable implementation; different tile sizes and loop orders may be optimal on different platforms. However, tuning a tiled code for a new platform is not technically complicated: one can determine the best tile size from a performance model, or simply test several sizes and pick the best.

IMPACT OF MCDRAM ON PERFORMANCE

The purpose of memory optimization undertaken in the previous step was to eliminate the dependence of simulation performance on the memory subsystem performance. In other words, we strive to make our application *compute-bound* as opposed to *memory-bound*. To test the success of our efforts, we used the ability of the Knights Landing to run the application either in the system DDR4 memory, or in the high-bandwidth onboard MCDRAM.

To perform this test, we ran the application with the tool `numactl` (see Chapter 3), which allowed us to set the NUMA policy to store all application data in different memory regions. This was possible because the dataset of this application, only 24 MiB in size, completely fits in the MCDRAM.

(1) To bind application data to NUMA node 0 (DDR4), we ran

```
numactl -membind 0 ./app-CPU
```

(2) To bind the application data to NUMA node 1 (MCDRAM), we ran

```
numactl -membind 1 ./app-CPU
```

Fig. 23.8 shows the results of our measurements. As we expected, for the last optimization step, the difference between DDR4 and MCDRAM vanishes. However, for less optimized code, DDR4 consistently performs a few percent faster. This came as a surprise because we expected that the streaming capabilities of MCDRAM should improve the application performance. Our interpretation of the observed result is that the performance is limited by memory latency, rather than bandwidth, which is comparable in DDR4 and MCDRAM (possibly, slightly better in DDR4—see Chapter 6).

We demonstrate a clever way to use MCDRAM while in flat mode: testing if an application is memory-bound vs CPU-bound.

	Step 0: Initial	Step 1: Multi-threaded	Step 2: Scalar tuning	Step 3: Vectorized with SoA	Step 4: Tiled, unrolled
Application in DDR4	2.1±0.0	316.1±0.1	442.5±0.1	1612±7	2875±3
Application in MCDRAM	2.1±0.0	309.4±0.0	424.7±0.0	1596±4	2874±2
Effect of MCDRAM	1.00	0.98	0.96	0.99	1.00

FIG. 23.8

Effect of MCDRAM usage on *N*-body simulation performance. See caption to Fig. 23.1 for explanation of units of measurements.

SUMMARY

We have demonstrated that through consistent application of optimization techniques to a high-level language code, step-by-step, it is possible to tap most of the available performance of highly parallel Intel® Architecture processors.

Furthermore, optimizing an application to take advantage of today's architectures is proven to be the best way to "future-proof" it and prepare for the upcoming architectures. Indeed, every step of the way the optimized code delivered greater performance on the Knights Landing than on the Knights Corner coprocessor, as well as on recent Intel Xeon processor platforms.

- Performance improvement due to multithreading is proportional to the number of cores. With up to 72 cores, Knights Landing gains more from multithreading than the previous platforms.
- Performance improvement due to scalar tuning translates to roughly the same speedup on all platforms.
- Conversion to unit-stride access helps on all platforms. Even though at this point, the code is memory-limited, it performs almost 2 × better than on an Intel Xeon processor with its large LLC. This is most likely entirely due to the improved cache architecture in Knights Landing.
- Tiling strategy proven to be helpful on the Knights Corner coprocessor also worked well for the newer Knights Landing processor.

The optimizations applied to the *N*-body code in this chapter were all identified by the optimization report or simple tests (e.g., thread, vector, and memory scalability tests), with only one exception — the strength reduction to exploit the reciprocal square root instruction. This took some architectural knowledge to exploit (see Chapters 2, 4, and 6) but still was straightforward to implement and test.

FOR MORE INFORMATION

(1) Recording of the *N*-body tutorial: http://lotsofcores.com/nbody01.

(2) Video illustration: http://lotsofcores.com/nbody02.

(3) Slides on the *N*-body tutorial with MPI (additional links and information inside): http://lotsofcores.com/nbody03.

(4) Updated information on *N*-body optimizations on Knights Landing and new presentation material after the publishing date of the book can be found at http://lotsofcores.com/nbody04.

(5) Another discussion of the direct *N*-body simulation with an alternative approach to tiling and details on performance estimates: Alejandro Duran and Larry Meadows, *Chapter 9: A Many-Core Implementation of the Direct N-Body Problem*, published in *High-Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, edited by James Reinders, Jim Jeffers, 2015, Morgan Kaufman, ISBN 978-0-12-802118-7.

CHAPTER

24 Machine learning

What is new with Knights Landing in this chapter?
AVX-512 and MCDRAM boost performance

Machine learning spans a broad set of algorithms that are used to extract useful models from raw data. These models in turn are used in a variety of mining tasks (where we extract one or more instances of a specific model from massive amounts of environmental data, such as finding a person's face occurring in a large number of picture frames at various resolutions, lightings, and so on) as well as synthesis tasks (where we construct new instances of the models, allowing predictive what-if scenarios or projecting the model in new environments).

Although machine learning as a field is not new, the vast amount of raw data that is now available (much of this due to mobile devices), advances in computational power of modern processors, and algorithmic advances to take advantage of this computational power have all combined to revolutionize the reach and impact of machine learning.

In this chapter, we focus on mapping and executing machine learning algorithms using an Intel product on the cutting edge of Moore's law: namely, the Knights Landing processor. Since the space of machine learning kernels is vast, we focus here on two specific algorithms that exercise different computational and memory access patterns and showcase the wide range of algorithms that can run on Knights Landing. First, we take a detailed look at Convolutional neural networks (CNNs) that form a subclass of networks used in "deep learning" neural network algorithms, which has proven to be quite accurate in many domains that deal with visual understanding problems such as object detection and classification, and scene recognition. State-of-the-art algorithms for CNN training and classification generally spend a lot of computational time in dense linear algebra operations, such as matrix multiply and convolutions.

Second, we focus on *k*-nearest neighbors (KNNs), a fundamental classification and regression method for machine learning used in text classification, prediction of economic events, medical diagnosis, and so on (see PANDA paper in the *For More Information* section). KNN works by finding the *k*-nearest points to a given query point in the feature space. There has been a lot of work done in KNNs to reduce the algorithmic complexity. An important contribution has been the use of acceleration data structures such as kd-trees (which hierarchically partitions points in a

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00024-7

k-dimensional space) which can help reduce the order complexity for nearest-neighbor searches from linear to logarithmic in the number of points. Such data structures work well in many scientific applications where the dimensionality of the points is not very high. In this chapter, we focus on kd-tree-based KNN algorithms. Such algorithms tend to have irregular computation and random memory accesses, in sharp contrast with the regularity of deep learning algorithms.

We also intend to map other machine learning algorithms to Knights Landing processor. We will provide updates at http://lotsofcores.com/knl_ML for these and other machine learning algorithms.

CONVOLUTIONAL NEURAL NETWORKS

CNNs have emerged as the *de facto* algorithm for most visual understanding problems spanning from object detection to sematic segmentation. Enabled by vast amounts of labeled data, collected off the internet and labeled by mechanical turks, and driven by availability of compute, CNNs have beaten state-of-the-art techniques in visual understanding by 5–10 × in terms of solution quality. Indeed few algorithms have benefited from Moore's law scaling more than CNNs—producing record-breaking performance by consuming all the compute that Moore's law could provide. In deep learning, the rule of thumb to get better solution quality is always to throw a bigger and deeper network at the problem, and CNNs are no different.

In this chapter, we discuss how to map and execute CNNs on Knights Landing processor. We deep dive into the cache blocking, threading, and register blocking strategies used for optimizing the three main kernels of CNN training: namely, the forward-propagation, the back-propagation, and weight-gradient update. We focus on convolutional layers since these comprise 90–100% of compute requirements for a CNN. We demonstrate how to best utilize the AVX-512, divide work to load balance between threads, and block dataset in caches to eliminate any memory bottlenecks. We use two threads per core to minimize the need for prefetches.

NEURAL NETWORKS

A neural network consists of layers of multiple neurons connected by weights. Each neuron computes the weighted sum of the input neurons of the previous layer to which it is connected. The value assigned to a neuron is often called activation. The activation of a neuron j in a layer L is computed as:

$$\text{Act}[L][j] = \sigma\left(\sum \text{Act}[L-1][i] \times W[L][i][j] + \text{Bias}[L][j]\right)$$

$W[L][i][j]$ is the weight connecting the ith activation in layer $L-1$ ($\text{Act}[L-1][i]$) to the jth activation in layer L ($\text{Act}[L][j]$), and $\text{Bias}[L][i]$ is the bias added to the above sum of products. Prior to assigning the activation for neuron j, a nonlinear

function $\sigma()$ is applied to it. Often a rectified linear unit is used, which essentially maps negative activations to zero, while keeping nonzero elements as is.

Fig. 24.1 illustrates a simple three-layer network with three neurons each in the first two layers and two neurons in the output layer. The bias is treated as a weight connected to a neuron with activation fixed to 1.

A neural network like the one in Fig. 24.1 has all neurons in layer L connected to all neurons in the previous/next layer. Such a neural network is called a fully connected neural network. Of course various forms of sparsity patterns can be possible in the connectivity of neurons. Moreover, weights may be shared across multiple neuron connections. One form of neural network with sparse connectivity and shared weights is CNNs.

In a CNN, the neurons in a given layer can be viewed as organized into a set of k-D matrices (2D for the most popular image processing CNNs) called *feature maps*. This structure is derived from the 2D nature of the input, and the fact that each feature map encodes a representation of the input in a 2D-space as well. The activation of an output neuron is computed as the weighted sum of patches across multiple input feature maps, and corresponding weight kernels. While CNNs have use across multiple domains, they have been most successful in the visual understanding space. In this chapter, we constrain ourselves to the image classification context.

In the simple CNN of Fig. 24.2, we have a convolutional layer with IFM = 2 input feature maps, an output layer with OFM = 4 output feature maps of size OFH*OFW each, and IFM*OFM kernels of size KH*KW corresponding to 8 input and output feature map pairs. It is notable that if KH = KW = 1 and OFH = OFW = 1, then such a convolutional layer is actually a fully connected layer.

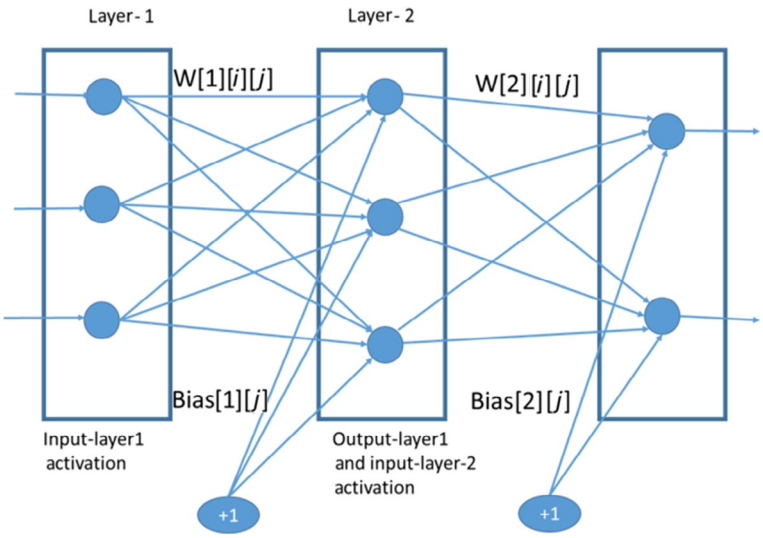


FIG. 24.1

A simple three-layer neural network, with one hidden layer with three neurons.

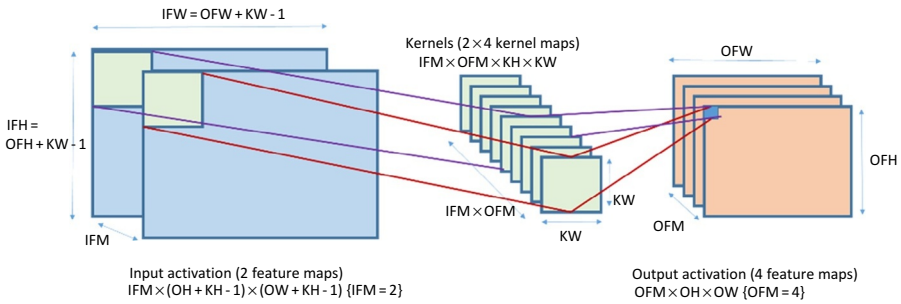


FIG. 24.2

A convolutional layer illustrating weight kernels, feature maps, and different variables used throughout the chapter.

Throughout the chapter we use the following notation:

- OFM/IFM: Number of Output/Input feature maps
- OFH/OFW: Height/Width of an output feature map
- IFH/IFW: Height/Width of an input feature map
- KH/KW: Height/Width of a weight kernel
- STRIDE: Stride of the convolution operation along both width and height directions.
- OB/IB: Block size along output/input feature map dimension.

A CNN is trained using the back-propagation algorithm, which takes a set of labeled input images as inputs and produces a set of learned kernel weights as output. This is essentially an optimization problem of trying to fit weights such that we minimize the error between classified outputs, and the actual ground truth labels. The minibatch stochastic gradient method is used, wherein the gradient of weights with respect to the output error is computed, and the weights are updated using a suitable update rule. In order to compute the gradient of the weights, three computationally identical steps, namely, forward-propagation, back-propagation, and weight-gradient computation are performed.

The loop for forward-propagation of a convolutional layer is presented in Fig. 24.3. Here the output activation ***output[ofm][ofh][ofw]*** is computed as a weighted sum of a patch of the input activation across *IFM* input feature maps. The operation in line 7 is a fused-multiply-add, accumulating into the output activations.

Similarly back-propagation has the exact loop structure and array indices, but the accumulation happens into the gradient of inputs (Fig. 24.4), while weight-gradient computation (Fig. 24.5) accumulates weight-gradient (***del_weight***) by convolving the gradient of output over the input activations.

The training of a CNN spends almost all of the time in these loops. Hence in the rest of the chapter, we concentrate on how to optimize the computation of these loops

```
1. for ofm = 0 … OFM-1
2.  for ifm= 0 … IFM-1
3.   for ofh = 0 … OFH-1
4.    for ofw = 0 … OFW-1
5.    | for kh = 0 … KH-1
6.    | | for kw = 0 … KW-1
7.    | | | output[ofm][ofh][ofw] +=
              input[ifm][STRIDE*ofh + kh][STRIDE*ofw + kw]*
                  weight[ofm][ifm][kh][kw];
8.    output[ofm][ofh][ofw] += bias[ofm];
```

FIG. 24.3

A naïve 6-nested loop for forward-propagation convolutional layers.

```
1. for ofm = 0 … OFM-1
2.  for ifm= 0… IFM-1
3.   for ofh = 0 … OFH-1
4.    for ofw = 0 … OFW-1
5.    | for kh = 0 … KH-1
6.    | | for kw = 0 … KW-1
7.    | | | del_input[ifm][STRIDE*ofh + kh][STRIDE*ofw + kw] +=
            del_output[ofm][ofh][ofw] * weight[ofm][ifm][kh][kw];
```

FIG. 24.4

A naïve 6-nested loop for back-propagation convolutional layers.

```
1. for ofm = 0 … OFM-1
2.  for ifm= 0… IFM-1
3.   for ofh = 0 … OFH-1
4.    for ofw = 0 … OFW-1
5.    | for kh = 0 … KH-1
6.    | | for kw = 0 … KW-1
7.    | | | del_weight[ofm][ifm][kh][kw] += del_output[ofm][ofh][ofw] *
              input[ifm][STRIDE*ofh + kh][STRIDE*ofw + kw];
8.    | del_bias[ofm] += del_output[ofm][ofh][ofw];
```

FIG. 24.5

A naïve 6-nested loop for weight-gradient update of convolutional layers.

on Knights Landing. We first look at cache blocking focusing on mainly the L2-cache. Thereafter we discuss the layout of data to enable contiguous access and vectorization. Finally, we discuss threading and register blocking strategies suitable for Knights Landing.

CACHE-BLOCKING STRATEGY FOR CONVOLUTIONAL LAYERS

The convolutional layer (forward-propagation) operation consists of a 6-nested loop as shown in Fig. 24.3. When written in the naïve fashion as in Fig. 24.6, the convolutional operation is bandwidth bound for many instances. It is simple to see that unless the activations (*input[] and output[]*) and weights completely fit in cache (which is often not the case), the third loop of neural network convolution operation (line 3)

```
1. for ofm = 0 … OFM-1
2.  for ifm= 0… IFM-1
3.   for ofh = 0 … OFH-1
4.    for ofw = 0 … OFW-1
5.    | for kh = 0 … KH-1
6.    | | for kw = 0 … KW-1
7.    | | | output[ofm][ofh][ofw] +=
              input[ifm][STRIDE*ofh + kh][STRIDE*ofw + kw]*
                   weight[ofm][ifm][kh][kw];
8.    output[ofm][ofh][ofw] += bias[ofm];
```

FIG. 24.6

A naïve 6-nested loop for forward-propagation of convolutional layers.

pulls in OFH*OFW output activations, (OFH*STRIDE+KH−1)*(OFW*STRIDE+KW−1) input activations (denoted as IFH*IFW), and KH*KW weights, while it performs KH*KW*OFW*OFH multiply-and-accumulate operations. The bytes to flops ratio can be easily computed to be:

```
B/F = data_size*(OFW*OFH + IFW*IFH + KW*KH)/
                (2*KW*KH*OFH*OFW).
```

For a typical CNN layer with IFM=OFM=1024, OFH=OFW=12, KH=KW=3, STRIDE=1 (as in layer C5 of the OverFeat-FAST CNN), we obtain a Bytes-Flops (B/F) ratio of 0.54. The single-precision bytes-to-flops (B/F) ratio for Knights Landing is computed as follows: 490 GB/s/(68*1.4*16*2*2) GFlops/s=0.08, assuming 490 GB/s of bandwidth. Clearly the kernel written like in Fig. 24.6 will be heavily bandwidth bound. The naïve loop therefore is theoretically limited to 15% efficiency (0.08/0.54). Since the operation is bandwidth bound, and the algorithmic B/F ratio is 0.54, the achievable performance is (490 GB/s)/(0.54 GB/s-per-GFlops) or 907 GFlops out of a peak of 6092 GFlops. This is also the ratio of algorithmic B/F and machine B/F. Now, even if we assume that after loop 2 the content can be stored in on-die caches, the B/F ratio improves only to 0.24 and the efficiency is limited to about 30% (0.08/0.24).

Clearly there is a need and opportunity to block loops 1 and 2 in on-die caches over input and output feature maps. We first consider blocking on loop 1 over output feature maps (block-size=OB) and produce the loop structure in Fig. 24.7. If OB output features, each of size OFH*OFW can be stored in the cache and then the B/F ratio becomes:

```
B/F = data_size*
      (OB*OFH*OFW + OB*IFM*KH*KW + IFM*IFH*IFW)/
      (OB*2*OFW*OFH*KH*KW*IFM)
```

In Fig. 24.7, we stream through the input features, and we reuse each input feature to compute OB output features. For the C5 layer of OverFeat-FAST, the values for variables are IFM=OFM=1024, OH=OW=12, and KH=KW=3. For this, we compute the B/F ratio for OB=16 to be: 0.033. This is below the 0.08 machine

```
 1. for ofm = 0 … OFM/OB-1
 2.   for ifm= 0… IFM-1
 3.     for ofh = 0 … OFH-1
 4.       for ofw = 0 … OFW-1
 5.       | for kh = 0 … KH-1
 6.       | | for kw = 0 … KW-1
 7.       | | | for ob = 0 … OB-1
 8.       | | | | output[ofm*OB+ob][ofh][ofw]+=
                    input[ifm][STRIDE*ofh + kh][stride*ofw + kw] *
                    weight[ofm][ifm][kh][kw];
 9.       for ob = 0 … OB-1
10.         output[ofm*OB + ob][ofh][ofw] += bias[ofm*OB + ob];
```

FIG. 24.7

A blocked 7-nested loop for forward-propagation of convolutional layers, with the loop over output feature maps blocked (loop-1 and loop-7).

B/F ratio and makes the loop compute bound. Many convolutional layers become compute bound when OB is set to the SIMD-Width (sixteen 32b-precision for Knights Landing) of the processor. We can further improve the B/F ratio by additionally blocking on loop 2 (Fig. 24.7) over IFM, wherein the B/F ratio further improves to:

```
B/F = data_size*
      (OB*OFH*OFW + OB*IB*KH*KW + IB*IFH*IFW)/
      (IB*OB*2*OFW*OFH*KH*KW)
```

For the example C5 layer can achieve a B/F ratio of 0.02. Note, however, that a B/F ratio below the machine B/F ratio is sufficient, and blocking the loop over input feature maps with a simple SIMD_WIDTH sized block keeps the implementation simple, and B/F ratio within requisite limits. Moreover keeping the innermost loop over OB (line 7, Fig. 24.7) enables vectorization of fused-multiply-and-add operation.

The backward pass also has a similar loop blocking, wherein loops 1 and 2 are swapped, and the blocking is performed over the input-feature maps dimension. The cache-blocking strategy for weight updates is the same as that for forward-propagation.

DATA LAYOUT

Following up on the blocked loop structure in Fig. 24.7, we immediately see that data is accessed in a strided manner in the innermost loop 7. Clearly there is benefit in laying out data so that the access in the innermost loop is continuous. This aids both a better utilization of cache lines and, therefore, bandwidth, while improving prefetcher performance.

In this work, we lay out all data, including activations (input[…], and output[…] arrays) and weights, with the innermost dimension over groups of SIMD_WIDTH output feature maps as described next. Here *fm* indicates a feature-map, fm_height/width indicate the height/width of a feature map, while output/input_fm indicate the output/input feature maps for a convolutional operation.

Activations and gradient of activations:
Act[fm/SIMD_WIDTH][fm_height][fm_width][SIMD_WIDTH]
Weights and gradients of weights:
Wt[fm/SIMD_WIDTH][output_fm/SIMD_WIDTH][kernel_heigh][height]
[kernel_width][SIMD_WIDTH][SIMD_WIDTH]
Transpose-weights:
TransWt[output_fm/SIMD_WIDTH][input_fm/SIMD_WIDTH]
[kernel_height][kernel_width][SIMD_WIDTH][SIMD_WIDTH]

VECTORIZATION AND REGISTER BLOCKING

Register blocking is used to improve reuse of data in register files, decrease L1 cache traffic, and most importantly hide the latency of the fused-multiply and add operation. The three main loops for forward-propagation, back-propagation, and weight-gradient update are written as shown in Figs. 24.8–24.10.

We first describe the forward-propagation loop (Fig. 24.8). This loop is derived from the cache-blocked loop described in Fig. 24.7. We further block the loop over output feature map width (line 5 in Fig. 24.8) into blocks of size RB_SIZE (Register-Block Size) and push the loop over RB_SIZE to the innermost loop (aside from the vector operation). This loop is to provide register blocking, wherein registers `vout[...]` store intermediate results for multiply-accumulates involving all the weights. This way we can generate an instruction sequence for the innermost loop that contains fused-broadcast-and-FMA instructions provided by Knights Landing. The number of vector FMA instructions in the innermost loop per vector load equal to the RB_size. While register blocking is illustrated within one row-vector of the output feature map, we can also perform 2D blocking, especially in cases where

```
          activations in L1-cache.
 1. vector vwt, vout[RB_SIZE];
 2. for ofm = 0 … OFM/SIMD-1
 3.   for ifm= 0… IFM/SIMD_WIDTH-1
 4.   |  for ofh = 0 … OFH-1
 5.   |  |  for ofw = 0 … OFW/RB_SIZE-1
 6.   |  |  |  for rb=0 … RB_SIZE-1
 7.   |  |  |  |  vout[rb] = SETZERO()
 8.   |  |  |  for kh = 0 … KH-1
 9.   |  |  |  |  for kw = 0 … KW-1
10.   |  |  |  |  |  for ib = 0…SIMD_WIDTH-1
11.   |  |  |  |  |  |  vwt = LOAD(weights[ifm][ofm][kh][kw][0])
12.   |  |  |  |  |  |  for rb = 0 … RB_SIZE-1
13.   |  |  |  |  |  |  |  VFMADD(EXTLOAD(input[ib]
                               [STRIDE*ofh + kh][STRIDE*ofw+kw][0]),
                               vwt, vout[rb])
14.   |  |  for rb=0 … RB-1
15.   |  |  |  STORE(vout,
                        output[ofm][ofmh][ofmw*RB_SIZE + rb][0])
```

FIG. 24.8

Vectorized pseudocode for forward-propagation (bias additions are not included).

```
 1. vector vtranswt, vdelinp[RB_SIZE];
 2. for ifm = 0 … IFM/SIMD-1
 3.   for ofm= 0… OFM/SIMD_WIDTH-1
 4.    |  for ofh = 0 … OFH-1
 5.    |  |  for ofw = 0 … OFW/RB_SIZE-1
 6.    |  |  |  for kw = 0 …  KW
 .   |  |  |  |  for rb=0 … RB_SIZE-1
 8.    |  |  |  |    vout[rb] =
                        LOAD(delinput[ifm][ofmh][ofmw*RB_SIZE+rb][0])
 9.    |  |  |  |  for kh = kh_start … kh_end
10.    |  |  |  |  |  for ob=0…SIMD_WIDTH-1
11.    |  |  |  |  |  |  vtranswt =
                        LOAD(trans_weights[ifm][ofm][kh][kw][0])
12.    |  |  |  |  |  |  for rb = 0 … RB_SIZE-1
13.    |  |  |  |  |  |  |  VFMADD(EXTLOAD(
                                 deloutput[ofm*SIMD_WIDTH + ob]
                                 [ofh][ofw*RB_SIZE+ rb][0]),
                                 vtranswt, vdelinp[rb])
14.    |  |  |  |  for rb=0 … RB-1
15.    |  |  |  |  |  STORE(delinp,
                           delinput[ifm][ofmh][ofmw*RB_SIZE + rb][0])
```

FIG. 24.9

Vectorized pseudocode for backward propagation.

```
 1. vector vdelout, vdelwt[RB_SIZE];
 2. for ofm = 0 … OFM/SIMD-1
 3.   for ifm= 0… IFM/SIMD_WIDTH-1
 4.   |  for mb=0… MINIBATCH-1
 5.   |  |  for ib = 0…SIMD_WIDTH-1 in steps of INPUT_RB_SIZE
 6.   |  |  |  for ob = 0 … SIMD_WIDTH-1
 7.   |  |  |  for kh=0 … KH
 8.   |  |  |  |  for kw=0…KW
 9.   |  |  |  |  |  for(inp_rbs=0 … INP_RB_SIZE)
10.   |  |  |  |  |  |  STORE(vdelwt[kh*KW+kw], delweights[ifm][ofm][kh][kw][0])
11.   |  |  |  |  for ofh = 0 … OFH-1
12.   |  |  |  |  |  for ofw = 0 … OFW/RB_SIZE-1
13.   |  |  |  |  |  |  for rb=0 … RB_SIZE-1
14.   |  |  |  |  |  |  |  vdelout = LOAD(deloutput[mb][ofm]
                                        [ofh][ofw*RB_SIZE+rb][0])
15.   |  |  |  |  |  |  |  for kh = 0 … KH-1
16.   |  |  |  |  |  |  |  |  for kw = 0 … KW-1
17.   |  |  |  |  |  |  |  |  |   for (inp_rbs = 0 … INPUT_RB_SIZE)
18.   |  |  |  |  |  |  |  |  |  |  vdelwt[inp_rbs][kh][kw] =
                                    VFMADD(EXTLOAD(input[mb][ib+inp_rbs]
                                      [STRIDE*ofh + kh]
                                      [STRIDE*(ofw*RB_SIZE+rb)+kw][0]),
                                      vdelout, vdelwt[kh*KH+kw])
19.   |  |  |  for kh=0 … KH
20.   |  |  |  |  for kw=0…KW
21.   |  |  |  |  |  for(inp_rbs=0… INPUT_RB_SIZE)
22.   |  |  |  |  |  |  STORE(vdelwt[inp_rbs][kh][kw],
                              delweights[ifm][ofm][kh][kw][0])
```

FIG. 24.10

Weight-gradient update operation for convolutional layers.

the size of the row is less than that of the number of registers which can be used. The second dimension can either be the next row or an adjacent set of SIMD_WIDTH output feature maps.

In addition to the blocking along the width of the feature map, we also create blocks of SIMD_WIDTH over the input feature maps, whenever possible.

Moreover we push the loop over a SIMD_WIDTH sized input feature map to the second innermost loop (line 10). This affords higher utilization of input activations in L1-cache.

The backward propagation code is presented in Fig. 24.9. This is similar to forward-propagation in terms of vectorization and cache blocking but differs in terms of how the register blocking is used. Here, the register block slides across the input row, while getting updated by corresponding transpose of weights and "gradient of outputs" (*deloutput[…]*). Similar to forward propagation, the loop over a block of "gradient of output" (of size SIMD_WIDTH) is the second innermost loop (line 10, Fig. 24.9).

For weight-gradient computation, we store the "weight gradients" (*delweights*) corresponding to kernels associated with one or multiple input feature maps and SIMD_WIDTH output feature maps in the register file.

The register blocking strategy varies for different types of kernels:

(1) 11×11 kernel: Typically 11×11 kernels are used in the first layer which has 1 or 3 feature maps. Hence the recommended register blocking strategy is to create a block for 2 or 1 row of the kernel.

(2) 7×7 kernel: Like 11×11 kernels, 7×7 kernels are typically used in the first convolutional layer, so we again recommend 3 or 2 rows of the kernel to be assigned to one register block.

(3) 5×5 kernel: The complete 5×5 kernel can be stored in the register.

(4) 3×3 kernel: Two 3×3 kernels associated with two consecutive input feature maps and the same SIMD_WIDTH number of output feature maps is assigned to one register block (INPUT_RB_SIZE = 2).

(5) 1×1 kernel: Kernels associated with SIMD_WIDTH input feature maps and sharing the same SIMD_WIDTH output feature maps are assigned to the register block (INPUT_RB_SIZE = 16). This is akin to a small block-GEMM operation.

The weight-gradient computation unlike forward or backward pass requires a reduce operation, wherein the weight-gradient contributions of each image are accumulated. For this, we structure the loop to create an inner loop over SIMD_WIDTH input features, and SIMD_WIDTH output features (lines 5 and 6 in Fig. 24.10) over the minibatch size (line 4). We stride along the input feature map dimension in steps of INPUT_RB_SIZE feature maps.

THREADING AND WORK PARTITIONING

For a many-core processor like Knights Landing, exposing sufficient parallelism and partitioning work into threads becomes a crucial design task. There are two considerations for threading: (a) The work partitioning should be such that the output produced by any two threads is ideally nonoverlapping. (b) In cases where the job cannot

be partitioned into threads with nonoverlapping outputs, we privatize outputs and follow up the convolution layer operation with synchronization and reduce operation.

We first examine the forward-propagation operation. Here the *output* activation consists of OFM*OFH*OFW neurons per image and a total of MINIBATCH* OFM*OFH*OFW neurons, each of which can be computed independently. Hence we can potentially have *MINIBATCH*OFM*OFH*OFW* (147456 work items for MINIBATCH = 1) work items, which can be easily partitioned into hundreds of threads. However, we recall that vectorization and register blocking require us to at least have each work item to have SIMD_WIDTH output feature maps, and RB_SIZE elements along the width of the feature map. Hence the number of work items is reduced to *(MINIBATCH*OFM*OFH*OFW)/(RB_SIZE*SIMD_WIDTH)* work items. We can simplify this further by partitioning along work items of size: SIMD_WIDTH*OFW instead of RB_SIZE*SIMD_WIDTH. For the forward-propagation, we continue to still have a large number of work items (768 for OverFeat C5 layer for MINIBATCH = 1), which can for example be divided into 132 threads with a load balance of 97% ((768/132)/ceiling (768/132)). While the Knights Landing preproduction part we use has 68 cores, we use only 66 of them and leave two cores for the operating system, communication management, and I/O operations. Hence the compute is performed by 132 threads running on 66 cores.

A similar work partitioning happens for the back-propagation and the threading strategy is again identical. The "gradient of inputs" (*del_input*) is partitioned into (IFM*IFH*IFW)/(IFW*SIMD_WIDTH) work items which are the distributed among threads.

For the weight-gradient updates, the number of work items can be considerably small when the number of input and output feature maps is low. For the OverFeat C5 layer, the number of work items (recall KH = KW = 3) with due consideration to the register blocking strategy leads to creation of: (OFM*IFM*KH*KW)/(KH*KW*SIMD_WIDTH*2) or 32768 work items. This yields to good load balance. However, for the C1 layer with OFM = 96, IFM = 3, KH = KW = 11, we have (OFM*IFM*KH*KW)/(SIMD_WIDTH*KW*IFM) = 66. This can fortunately be divided into 66 threads equally to attain good load balance, but a slight change in topology could lead to up-to 30% load imbalance for this layer. In such cases, we need to privatize the "gradient of weights" array and follow it up with a synchronization and reduce operation.

THE EXPERIMENTAL SETUP

The experimental setup for each of the systems used in our benchmarks is described in Fig. 24.11.

Average performance over four benchmark runs is reported. We found no significant performance differences between applications compiled with v15 or v16 versions of the Intel C/C++ Compiler.

Code name	Knights Landing
Version	Preproduction B0
Cores	68 cores @ 1.4 GHz
Memory	6x16GB DDR4 2400MHz
Default MCDRAM configuration	Size: 16 GB
Other details	Memory mode: Flat Cluster mode: Quadrant MCDRAM bandwidth: 460GB/s

FIG. 24.11

Experimental setup on preproduction Knights Landing.

The Knights Landing environment was setup via the following environment variables:

- KMP_AFFINITY = compact, granularity = fine.
- KMP_PLACE_THREADS was used to set the number of threads per core used.

We utilized used "`numactl -m 1 -- <app cmdline>`" (see Chapter 3) to allocate in MCDRAM when testing the flat mode.

OverFeat-FAST RESULTS

In this section, we present performance results for the OverFeat-FAST topology. The throughput for training is measured in terms of images-per-second processed. We attain a peak performance of 220 img/s on the Knights Landing system described above with 66 threads, with 1 thread executing per core. This translates into a training time of about 114 h (less than 5 days) assuming that training takes 70 epochs. All experiments are conducted for a minibatch size of 256. The working set of the CNN training code fits into MCDRAM and is accessed at a high-bandwidth.

K-NEAREST NEIGHBORS

The KNN algorithm is used in a variety of classification and regression tasks in machine learning. The key idea behind its machine learning applications is that points tend to share the properties of nearby points (the distance function from one point to another often depends on the context—some common ones include Euclidean distance between particles in space, Hamming distance between words, etc.). In a classification setting, a majority vote of the labels of the KNN is often used to determine the label of a point. In a *regression* setting (where regression is a machine learning technique commonly used to obtain continuous outputs as opposed to discrete outputs in classification), an average (or maximum or minimum) of the KNN is typically used to determine the value of the variable being regressed. Due to its wide applicability and simplicity, KNN is used in a large number of applications such as text or image classifications, medical diagnosis, and prediction of economic events.

ALGORITHMS

There are many approaches to finding the KNN of a given point, usually known as a query point. Assuming a suitable distance function has been defined, one approach is to simply iterate through all points and compute distances from the query point. The nearest k distances can then be selected. This "brute force" approach has the advantages of being very simple and in fact highly parallelizable. However, each query in this approach takes linear time in the size of the dataset. For large datasets with millions to billions of points, such as those commonly found in scientific datasets from astronomy or particle physics, this can be prohibitive in runtime.

KD-TREES

kd-trees are an efficient data structure to find nearest neighbors of points. Instead of computing distances from the query point to all other points in order to find the nearest neighbors, we prune away many regions of space during kd-tree traversal and compute only a few distances along the way (Fig. 24.12).

The key idea behind kd-trees is a spatial partitioning of the data points. It is likely that nearby points belong to the same partition; hence all points in partitions that are far away from the query can be completely ignored. A simple example of using a kd-tree for KNN is shown in Fig. 24.12. In this figure, the dataset is divided recursively based on axis-aligned splits of data. For example, the first split in Fig. 24.12 lies along the central vertical line with two partitions to the left and right of the central vertical line. This recursive partitioning can also be shown by a tree (the kd-tree) to the right. The root node depicts the bounding box (it is sufficient to store only the split dimension and split value) corresponding to the entire dataset and the two children are two partitions created by the central vertical line. Each node in the tree hence corresponds to a bounding box in the partitioned space. Each nonleaf node in the tree

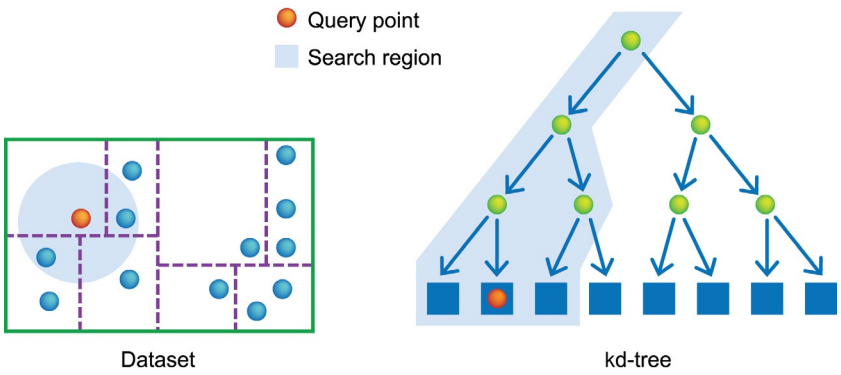


FIG. 24.12

Computing the k-nearest neighbors of a query point for $k=3$. *Left*: Shows the k-nearest neighbors. *Right*: Shows the search regions.

defines a split of the data. The leaves of the tree hold the data—each leaf node holds one or more data points in them.

In order to execute a query, we perform a depth-first traversal of the kd-tree. At any point of time, we keep an upper bound to the distance from the query to the *k*th farthest away point—this starts at infinity. As we move down the tree, we find the distance between the query and the bounding box corresponding to each child. If any bounding box is farther away than the upper bound distance to the *k*th neighbor, then we are guaranteed that no points in that subtree can be among the KNN of the query and we do not have to traverse the subtree and perform distance computations—this is at the heart of the computational savings. As we reach leaf nodes, we perform distance computations to each of the points in the leaf from the query and update the neighbor list and upper bound distance if any closer neighbor is found.

With the right spatial decomposition, kd-trees have been shown to reduce the number of distance computations from linear to logarithmic in the number of data points. This speedup during the query comes at the expense of the construction of the kd-tree; it also results in data redistribution and control divergence. Hence it is imperative to perform careful optimizations for both kd-tree construction and querying in order to reap the benefits of the improved order complexity. Further, the number of leaf nodes traversed during query depends on the quality of the tree (choice of splits). There is often a tradeoff between kd-tree construction times versus tree quality. We delve into these topics next.

OPTIMIZED KD-TREE CONSTRUCTION

We adopt the kd-tree construction algorithm described in PANDA (see *For More Information*). At each level of the kd-tree, the data points need to be split into two parts. For best efficiency during query, it is important that we try to split the data points approximately equally into two subsets. We perform this split by choosing a dimension and a split point along that dimension. The algorithms used for these choices are presented later in detail. Once the split point is chosen, the points have to be rearranged such that all points in one split are placed adjacent to each other in memory. For efficiency, we move only indices to the data and not the points themselves. At the end of the kd-tree construction, we shuffle the data according to the rearranged indices.

In order to reduce the runtime of construction, we develop a multithreaded algorithm to run on the highly parallel Knights Landing processor. The overall algorithm consists of three parallel phases (Fig. 24.13).

- *Data parallel*: For the first few levels of the tree (called upper part), there are not enough branches to exploit thread level parallelism where we can assign threads to independent branches that need to be constructed. Since we deal with large datasets, we can instead use all threads to cooperatively calculate the split and shuffle the points and proceed in breadth-first fashion (one level at a time). Once

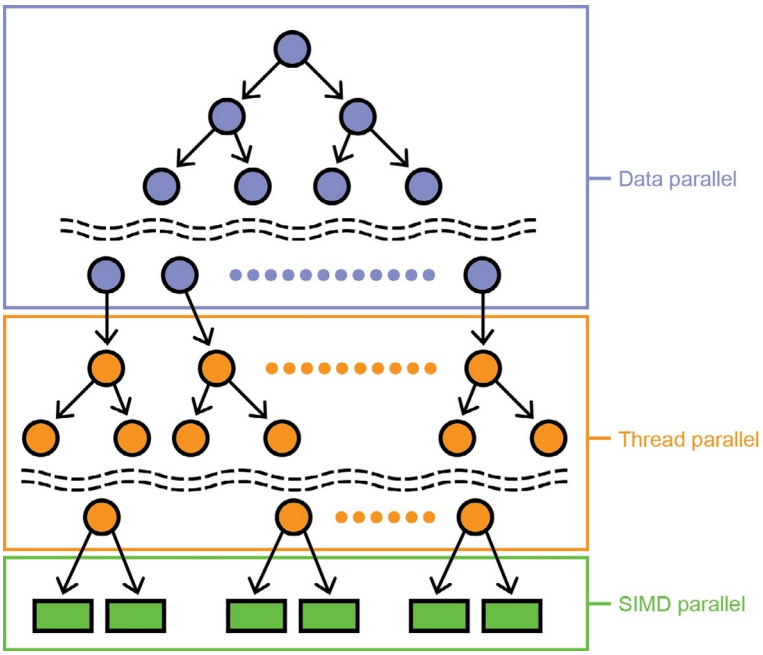


FIG. 24.13

Construction of Kd-tree exploiting data, thread, and SIMD parallelization.

there are sufficient branches, we move to the next stage of exploiting thread level parallelism.

- *Thread parallel*: In this stage, each thread proceeds to create the kd-tree from a distinct, nonoverlapping set of points. To ensure cache locality, this tree construction proceeds in a depth-first manner. We declare a node to be a leaf node when the number of points in it reaches a threshold (maximum bucket size, typically 32).
- *SIMD parallel*: Once the points in each bucket are fixed, we shuffle the dataset such that points within a bucket are localized in memory. This improves the query performance as we need to perform exhaustive distance computation with all points in a bucket (if selected) at query time.

ALGORITHMIC CHOICES

We discuss some of the key algorithmic choices in this section:

- *Choice of split dimension*: As mentioned earlier, at every level of the kd-tree, the data needs to be split approximately into two subsets. We perform this split by choosing a split dimension and a split point along that dimension. The former choice may be based on maximum range (e.g., ANN - see *For More Information*)

or a more nuanced metric allowing for better partitioning. We use the dimension with the maximum variance as the best dimension to split in the kd-tree. As this computation could be expensive, we take a random subset of points to compute variances. This is similar to the strategy used in FLANN (see *For More Information*).

- *Choice of split point*: We then have to choose the split point along that dimension. The ideal point is the median along that dimension that promises to divide the dataset into two equal subsets. Calculating the median is expensive; hence we use heuristics to approximate this. We utilize a sampling heuristic to estimate the data distribution along a given dimension and choose a point close to the median as the split. Our heuristic is similar to that of used in the PANDA work listed in *For More Information*. Each thread samples a small set of points ($m = 1024$ points each for our implementation). Given all samples, the threads then sort these samples and cooperatively build a histogram along the chosen dimension using these as the (nonuniform) interval points. This histogram is then traversed by all threads, and the interval point closest to 50% is used as the approximate median. We further optimize the operation of finding the right histogram bin to increment per data point. Rather than doing a binary search on the sorted interval points (which suffers from branch misprediction), we pull in every 32nd interval point into a separate subinterval array to reduce the search space. This is then scanned using AVX-512F instructions. Once we identify the two subinterval points between which the data point lies, we scan that specific range of 32-elements in the full interval point array (again using AVX-512F) and locate the histogram bin to increment. We get overall performance gains of up to 42% (e.g., cosmology dataset) during local kd-tree construction over binary search.
- *Choice of bucket size (or number of levels)*: We create new levels until the number of points in a leaf node is a predefined bucket size. Once this is reached, we stop creating new levels and pack the points into a bucket. Larger buckets improve construction time, but make querying more expensive, because querying with a bucket requires an exhaustive search of all points in the bucket. Empirically, we found that a bucket size of 32 gave the best performance.

OPTIMIZED KD-TREE QUERY

The algorithm for finding the k-nearest neighbors from a local kd-tree, T is given as pseudocode in Fig. 24.14. The algorithm relies on maintaining bounds on the distance to kth neighbor (denoted as r' in Fig. 24.14) and progressively refining it while using it to prune regions of the tree to limit exploration. Before traversing a node down in the tree, we always keep track of the minimum distance of the query point to all the points in the node (denoted by d and d' in Fig. 24.14) and use this for pruning. Once we reach a leaf node, we find the distance from query to all the points in the bucket. This computation is very SIMD friendly as the required points are localized in memory. In nonleaf nodes, we push the closer child into the stack later than the other child. This is an optimization done so that we get to the closer neighbors of the

```
Input: kd-tree T, Query q, and k
Output: A set, R of k nearest neighbors.

procedure FINDKNN(T, q, k)
 1: r' ← ∞; push (root,0) into S
 2: while S is not empty do
 3: |  (node,d) ← pop from S
 4: |  if node is leaf then
 5: |  |  for each particle x in node do
 6: |  |  |  compute distance, d[x] of x from q
 7: |  |  |  if d[x] < r' then
 8: |  |  |  |  if |H| < k then
 9: |  |  |  |  |  add x into H
10: |  |  |  |  if |H| = k then
12: |  |  |  |  |  r' ← H.maximum distance
13: |  |  |  |  else if d[x] < max distance in H then
14: |  |  |  |  |  replace the topmost point of H by x
15: |  |  |  |  |  r' ← d[x]
16: |  else
17: |  |  if d < r' then
18: |  |  |  d' ← q[node.dim] − node.median
19: |  |  |  d' ← √{d * d + d' * d'}
20: |  |  |  C1 ← closer child of node from q
21: |  |  |  C2 ← other child of node
22: |  |  |  if d' < r' then
23: |  |  |  |  push (C2,d') into S
24: |  |  |  push (C1,d) into S
25: R ← H
```

FIG. 24.14

Finding *k*-nearest neighbors from the local kd-tree.

query earlier. This essentially helps pruning (Line 17 and 22) the search space for the later processed nodes. We use a heap (H) to keep track of *k*-nearest neighbors found so far.

KNN RESULTS

We now present results for KNN kd-tree construction and query. We use the same experimental setup as in.

To evaluate the performance of KNN query, we used datasets collected from the Sloan Digital Sky Survey (SDSS) database (see *For More Information*) that gathers data for hundreds of millions of astronomical objects. From these objects, one usually extracts a small set of features using different extraction schemes such as point spread-function (*psf*), the model approach (*model*), and the petrosian (*pet*) approaches. Using these schemes, we collected three datasets, psf_mag, psf_model_mag, and all_mag, which have also been used in a similar kd-tree-based KNN study

	Construction		Query	
Dataset name	Points	Dims	Points	Dims
psf_mag	2 M	5	10 M	5
psf_mod_mag	2 M	10	10 M	10
all_mag	2 M	15	10 M	15
M50	50 M	3	-	-
M250	250 M	3	-	-

FIG. 24.15

Datasets used in KNN kd-tree construction and query. The first three datasets are used to evaluate queries and the last two datasets to evaluate the kd-tree construction. M denotes datasets with millions of points.

on GPUs (see *For More Information* for more details). The first three rows of the table in Fig. 24.15 show the properties of the construction and query datasets used in these experiments. The query datasets are much larger than the construction datasets, a precise setting faced in astronomy and similar to the use cases in previous work on GPUs. Since the construction dataset contains only 2 M points and it takes less than a second of runtime to construct the kd-tree, this is not a bottleneck for these datasets. We therefore used a separate set of larger construction datasets (M50 and M250) to analyze the performance of KNN training. These datasets have been taken from cosmological *N*-body simulations using Gadget code, and we use spatial locations in this analysis similar to our previous work on CPU clusters. These two datasets are shown in the last two rows of Fig. 24.15.

Scalability

We first show scalability of kd-tree construction and query in Fig. 24.16 for the datasets shown in Fig. 24.15. Fig. 24.16A shows the scaling of the kd-tree construction phase. As we scale tree construction up to 68 threads, we obtain near-linear

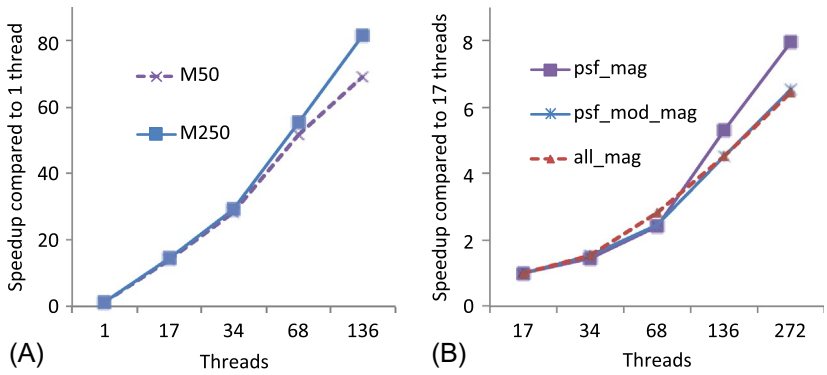


FIG. 24.16

Thread scaling for (A) kd-tree construction and (B) kd-tree query.

scaling—we obtain a speedup of about 52–56× for 68-thread runs as opposed to single-threaded runs. This near-linear scaling shows that we have successfully distributed the construction with varying levels of parallelism to threads with minimal load imbalance. We also examine the impact of hyperthreading (HT). As we increase the number of threads from 68 to 136 by running 2 threads/core, we obtain performance improvement of 1.3–1.5×. This impact is due to the irregular memory accesses in KNN kd-tree construction, which results in high-latency memory misses. HT can help hide this latency by issuing and executing instructions from the other thread if a thread stalls waiting for data to come back from memory.

Fig. 24.16B shows the thread scalability for kd-tree query for 10 M queries. Since the runtime even for 17-threaded runs are quite high (∼10 min for the all_mag dataset), we did not run experiments with fewer than 17 threads (we arrive at this odd-looking number "17" when evenly dividing 68 cores to create lower core count experiments). The kd-tree query code is significantly limited by memory accesses, since there is very little work done at each intermediate node of the kd-tree traversal. The only real computation is the distance computation at the leaf nodes, and this work increases with dimensionality of the dataset. We see that the all_mag dataset, with 15 dimensions, scales better (2.8× scaling from 17 to 68 (4×) threads) than the psf_mag and psf_mod_mag datasets with fewer dimensions (2.4–2.5× from 17 to 68 (4×) threads). We also see significant impact of using more than 1 thread/core. We see 1.6–2.2× performance improvement when running with 2 threads/core rather than 1 thread/core, and we get a further performance improvement of about 1.4–1.5× when running with 4 threads/core. This results in a total performance speedup of 2.3–3.3× using HT. As expected, the higher-dimensional all_mag dataset is less sensitive to HT (2.3× scaling) than the lower-dimensional psf_mag dataset since there is more computation to perform at the leaf nodes.

Sensitivity analysis

We now examine the sensitivity of KNN kd-tree construction and query to algorithmic choices—the choice of bucket size and the level where we switch from data to thread parallel approaches during tree construction. These choices were described in the algorithms section.

CHOICE OF BUCKET SIZE

The choice of bucket size mainly influences the runtime of kd-tree querying. We found little impact on kd-tree construction runtimes. The query times are influenced due to two factors: (1) as the bucket size increases, the height of the tree constructed reduces. This reduces the number of levels that need to be traversed during query, which leads to reduced query times. (2) At the same time, the number of distance computations once the traversal reaches a leaf node increases with bucket size. This tends to increase query runtimes. In practice, there is an ideal middle ground where these two factors get balanced. Fig. 24.17 shows how query performance changes as

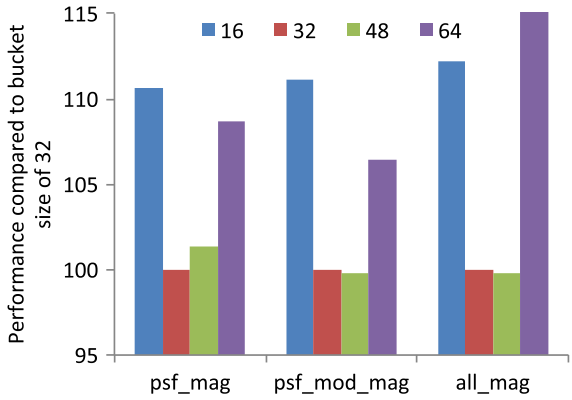


FIG. 24.17

Sensitivity of tree query performance on changing bucket size from 16 to 64. Performance for a bucket size of 32 is normalized to a value of 100.

we vary bucket size from 16 to 64 (normalized to 100 for a bucket size of 32). As we can see, for all three datasets, the best bucket size is 32.

CHOICE OF LEVEL AT WHICH TO SWITCH FROM DATA-PARALLEL TREE CONSTRUCTION TO A THREAD-PARALLEL APPROACH

This determines the level where threads stop cooperatively creating kd-tree splits and rearrangement of points. After this point, we can assign the construction of independent subtrees to threads. This will have little impact on query times, since the tree constructed should be the same no matter what the parallelization strategy for tree construction is. Fig. 24.18 shows the impact of varying the number of leaf nodes for the data-parallel section (upper part) from 64 to 512 (corresponding to 6–9 tree levels). As we increase this parameter, a bigger portion of the tree is created in a cooperative manner, which requires tight synchronization between threads and is inherently less efficient than letting threads perform independent computations. However, at the same time, we cannot drop this parameter too far as this determines the number of independent subtrees that will be available at the thread parallel construction phase. We need at least as many independent subtrees as the number of threads. In practice, our performance, at 128 and 256 leaf nodes at the data-parallel level (128–256 subtrees for 136 threads), tended to perform the best.

ABSOLUTE PERFORMANCE RESULTS

We report absolute performance results for the astronomy data sets listed in Fig. 24.15. We believe these are the best performance results that have been reported for both KNN kd-tree construction and query.

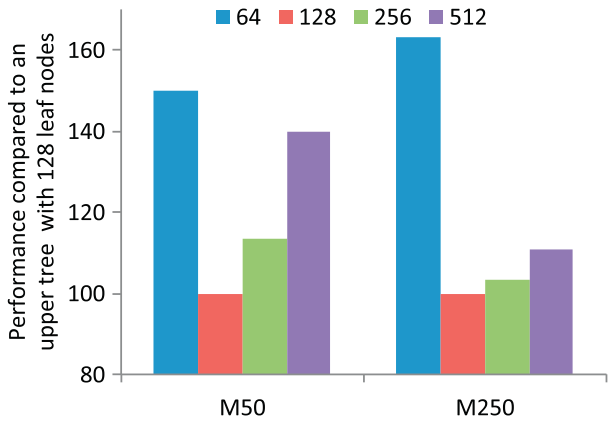


FIG. 24.18

Sensitivity of tree construction performance on the level where we change parallelization strategies from data parallel to thread parallel. We vary the number of leaf nodes in the data-parallel part of the tree from 64 to 512. The performance for 128 leaf nodes is normalized to 100.

	Results on NVIDIA Titan Z GPU [2] (s)	Knights Landing (preproduction) (s)
psf_mag	12	3.77
psf_mod_mag	36	19.96
all_mag	310	91.11

FIG. 24.19

Comparison of runtimes for KNN query using the implementation in [2] for the NVIDIA Titan Z and our implementation for Knights Landing. Runtimes are shown in seconds, so lower is better.

We compare our results (see Fig. 24.19) to state-of-the-art implementations for KNN for highly parallel systems. Due to the difficulty of implementing and optimizing kd-trees, much of the previous work in the area especially on GPUs has focused on brute-force KNN approaches. There are also hybrid approaches that perform kd-tree training on the CPU, transfer the resulting kd-tree to the GPU, and perform queries on the GPU. We compare only our KNN query results with those in state-of-the-art GPU implementations since tree constructions times were not reported. Compared to the results reported on an NVIDIA Titan Z GPU (Fig. 24.19), our results on the Knights Landing processor are 1.8–3.4× faster. We note that the buffered kd-tree approach in Fig. 24.19 requires a large query set to be streamed from the CPU to GPU over PCIe. Since we have access to the entire DDR memory in a Knights Landing processor, we do not need any such streaming of queries and we avoid this potential bottleneck.

FOR MORE INFORMATION

- SDSS dataset: Ahn, C.P. et al. The tenth data release of the Sloan Digital Sky Survey: First spectroscopic data from the SDSS III Apache Point Observatory galactic evolution experiment. eprint arXiv:1307.7735, 2013.
- KNN using Buffered kd-trees on multiple GPUs: Gieseke, Fabian, Cosmin Eugen Oancea, Ashish Mahabal, Christian Igel, and Tom Heskes. "Bigger Buffer kd-Trees on Multi-Many-Core Systems." arXiv preprint arXiv:1512.02831 (2015).
- KNN using buffered kd-tree on a single GPU: Gieseke, Fabian, Justin Heinermann, Cosmin Oancea, and Christian Igel. "Buffer kd trees: processing massive nearest-neighbor queries on GPUs." In Proceedings of the 31st International Conference on Machine Learning, pp. 172–180. 2014.
- KNN on large scale CPU clusters: *PANDA: Extreme Scale Parallel K-Nearest Neighbor on Distributed Architectures*; Md. Mostofa Ali Patwary, Nadathur Satish, Narayanan Sundaram, Jialin Liu, Peter Sadowski, Evan Racahc, Surendra Byna, Wahid Bhimji, Craig Tull, Prabhat, and Pradeep Dubey, accepted to IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016.
- V. Springel, "The cosmological simulation code GADGET-2," Monthly Notices of the Royal Astronomical Society, vol. 364, pp. 1105–1134, Dec. 2005.
- ANN library: D. M. Mount and S. Arya, "ANN: A library for approximate nearest-neighbor searching," https://www.cs.umd.edu/~mount/ANN/, 2010, version 1.1.2.
- FLANN library: M. Muja and D. G. Lowe, "FLANN—fast library for approximate nearest neighbors," http://www.cs.ubc.ca/research/flann/, 2013, [Version 1.8.4].
- We will provide updates at http://lotsofcores.com/knl_ML for machine learning algorithms.

CHAPTER

Trinity workloads

25

What is new with Knights Landing in this chapter?
Use of MCDRAM, Hyperthreading and AVX-512.

In this chapter, we explore a set of High Performance Computing workloads, the Trinity workloads, on Knights Landing. To get the best performance from such workloads on Knights Landing, you will need to consider multiple cluster modes, memory modes, problem sizes of your workload, and hardware threads utilized. We will discuss the "out-of-the-box" performance (no source code changes) and compare it to an Intel® Xeon® Processor. As we present the performance results, we will provide suggestions on when to try the different cluster modes and memory modes supported by Knights Landing. We will discuss problem sizing and why it is critical and demonstrate how hyperthreading performance varies depending on the cluster or memory mode being used. We will end with a case study of MiniGhost performance analysis and optimizations. Please see the *For More Information* section at the end of this chapter to download Trinity workloads and to get more details about each of these workloads.

OUT OF THE BOX PERFORMANCE

Two of the earliest Knights Landing supercomputers: NNSA/Trinity and NERSC/Cori both specified the same eight Trinity workloads for performance evaluation. The procurement asked for performance results with no source code changes and for MPI-only performance. As described in Section I, Knights Landing is an out-of-order many-core processor with multiple hardware threads per core, multiple cluster configurations, multiple memory modes, sharing binary compatibility with Intel Xeon processors with AVX, AVX2, AVX-512F&CD instruction sets—these features are what enable Knights Landing to have superior "Out of the Box" performance. In this section, we present the performance of Trinity workloads as-is (no source code changes) with MPI-only on single node. The eight Trinity workloads are listed in Fig. 25.1.

These workloads are intended to be run on a cluster but can also be executed on a single node. This chapter focuses on the single-node runs with two restrictions on the problem size:

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00025-9

	Weak/ Strong	Can Fit in MCDRAM?
AMG	Weak	Y
GTC	Weak	**N**
MILC	Weak	Y
MiniDFT	**Strong**	**N**
MiniFE	Weak	Y
MiniGhost	Weak	Y
SNAP	Weak	Y
UMT	Weak	Y

FIG. 25.1

The eight Trinity workloads.

(1) The minimum problem size per node should be such that the full problem can still fit on to a cluster of 9K nodes as the supercomputers mentioned previously will have at most 9K Knights Landing processor nodes.

(2) Problem size selected needs to be large enough where data is coming from memory and not from core caches so that we can study the effects of the memory subsystem.

Two single-node systems were used:

- Intel® Xeon E5-2697 v4 processor (codenamed Broadwell)
  - Preproduction B0 stepping
  - 18 cores per socket, 36 total cores
  - 2.3 GHz
  - hyperthreading performance was not tested
  - Red Hat 6.5
  - Turbo Enabled
  - 145 W per socket, 290 W for both sockets
  - Memory: 128 GB RAM (DDR4 2400 8 × 16 GB DIMMS)
- Knights Landing
  - Preproduction Knights Landing stepping B0 @ 1.40 GHz
  - 34 Tiles, 68 total cores, 272 total hardware threads
  - Default configuration with power states (P0: 1600 MHz, P1: 1400 MHz, PN: 1000 MHz); Turbo enabled
  - Default uncore frequency: 1700 MHz
  - SUSE Linux 12
  - MCDRAM size: 16 GB
  - Memory: 96 GB (6 × 16 GB 2400 MHz DDR DIMMS)
  - 215 W per socket which includes power for 16 GB of MCDRAM

BUILDING THE WORKLOADS

The source code was obtained from NERSC website. The makefiles of the Trinity workloads were modified to enable Intel tools.

Four Intel tools were used to build and run these workloads:

- Intel® C\C++ Compiler and Fortran Compiler: 16.0.0 20150815 and Mainline 20150119
- Intel® MPI Library: 5.1.1 Build 20150715
- Intel® MKL: 20150730, 11.3.0

The following build options were used:

- **AMG Knights Landing:**

```
-O3
-xMIC-AVX512
-fp-model precise
-openmp
-ip
-no-prec-div
-no-prec-sqrt
-DTIMER_USE_MPI
-DHYPRE_USING_OPENMP
-DHYPRE_LONG_LONG
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **AMG Broadwell:**

```
-O3
-xCORE-AVX2
-fp-model precise
-openmp
-ip
-no-prec-div
-no-prec-sqrt
-DTIMER_USE_MPI
-DHYPRE_USING_OPENMP
-DHYPRE_LONG_LONG
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **UMT Knights Landing:**

```
-w
-fPIC
-openmp
```

```
-O3
-xMIC-AVX512
-openmp
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **UMT Broadwell:**

```
-fPIC
-openmp
-O3
-xCORE-AVX2
-fp-model fast=1
-ftz
-fno-alias
-openmp
-ip
-no-prec-div
-no-prec-sqrt
```

- **GTC Knights Landing:**

```
-O3
-xMIC-AVX512
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-openmp
-align array64byte
-ip
-no-prec-div
-no-prec-sqrt
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **GTC Broadwell:**

```
-O3
-xCORE-AVX2
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-openmp
```

```
-align array64byte
-ip
-no-prec-div
-no-prec-sqrt
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **MILC Knights Landing:**

```
-O3
-xMIC-AVX512
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-openmp
-ip
-no-prec-div
-no-prec-sqrt
-std=c99
-DMPI
-DCGTIME
-DFFTIME
-DLLTIME
-DGFTIME
-DREMAP
-DDBLSTORE_FN
-D_LARGEFILE64_SOURCE
-DFN
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **MILC Broadwell:**

```
-O2g
-xCORE-AVX2 g
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-openmp
-ip
-no-prec-div
```

```
-no-prec-sqrt
-std=c99
-DMPI
-DCGTIME
-DFFTIME
-DLLTIME
-DGFTIME
-DREMAP
-DDBLSTORE_FN
-DD_FN_GATHER13
-DFEWSUMS
-DC_GLOBAL_INLINE
-DPRECISION=1
-D_FILE_OFFSET_BITS=64
-D_LARGEFILE64_SOURC
-DFN
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **MiniDFT Knights Landing:**

```
-O3
-xMIC-AVX512
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-openmp
-ip
-no-prec-div
-no-prec-sqrt
-mkl=parallel
-lmkl_cdft_core
-D__FFTW3
-D__MPI
-D__SCALAPACK
-D__IPM
-lmkl_scalapack_lp64
-lmkl_blacs_intelmpi_lp64
-fpp
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **MiniDFT Broadwell:**

```
-O3
-xCORE-AVX2
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-openmp
-ip
-no-prec-div
-no-prec-sqrt
-mkl=parallel
-lmkl_cdft_core
-D__FFTW3
-D__MPI
-D__SCALAPACK
-D__IPM
-lmkl_scalapack_lp64
-lmkl_blacs_intelmpi_lp64
-fpp
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **MiniFE Knights Landing:**

```
-O3
-xMIC-AVX512
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-ip
-no-prec-div
-no-prec-sqrt
-DMINIFE_SCALAR=double
-DMINIFE_LOCAL_ORDINAL=int
-DMINIFE_GLOBAL_ORDINAL=long long int
-DMINIFE_CSR_MATRIX
-DHAVE_MPI
-DMPICH_IGNORE_CXX_SEEK
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **MiniFE Broadwell:**

```
-O3
-xCORE-AVX2
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-openmp
-ip
-no-prec-div
-no-prec-sqrt
-DMINIFE_SCALAR=double
-DMINIFE_LOCAL_ORDINAL=int
-DMINIFE_GLOBAL_ORDINAL=long long int
-DMINIFE_CSR_MATRIX
-DHAVE_MPI
-DMPICH_IGNORE_CXX_SEEK
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **MiniGhost Knights Landing:**

```
-O3
-xMIC-AVX512
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-openmp
-ip
-no-prec-div
-no-prec-sqrt
-D_MG_MPI
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **MiniGhost Broadwell:**

```
-O3
-xCORE-AVX2
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
```

```
-fimf-domain-exclusion=15
-openmp
-ip
-no-prec-div
-no-prec-sqrt
-D_MG_MPI
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **SNAP Knights Landing:**

```
-O3
-xMIC-AVX512
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-openmp
-fno-alias
-align array64byte
-ip
-no-prec-div
-no-prec-sqrt
-g
-debug inline-debug-info
-parallel-source-info=2
```

- **SNAP Broadwell:**

```
-O3
-xCORE-AVX2g
-fp-model fast=1
-ftz
-fno-alias
-fimf-precision=low
-fimf-domain-exclusion=15
-openmp
-fno-alias
-align array64byte
-ip
-no-prec-div
-no-prec-sqrt
-g
-debug inline-debug-info
-parallel-source-info=2
```

ITEMS TO CONSIDER BEFORE RUNNING ON KNIGHTS LANDING

Two questions should always be answered when describing any workload running on Knights Landing: What memory and cluster modes are used? What is the problem size per node? Before we can discuss the performance results of Knights Landing, we need to briefly touch on the different memory modes and cluster modes one can use and how problem size impacts performance.

Memory modes and cluster modes

Knights Landing has new memory modes and cluster modes that can be used. These modes were designed to provide different performance experiences depending on the needs of the workloads. The modes and their programming are discussed in detail in Section I of this book.

Strong scaling vs weak scaling vs problem sizing

Problem sizing is critical on Knights Landing for many workloads due to the limited capacity of the high-bandwidth MCDRAM memory. Strong scaling is when the overall problem size is fixed. Therefore, as we add more processors, the work done per processor decreases. Weak scaling is when the work done per processor is fixed. The result then is that as we add more processors, the overall problem size grows proportionally. We use the term "problem sizing" to describe the process of optimizing the work done per processor before applying weak scaling (see Fig. 25.2). Problem sizing is not a new concept, but on Knights Landing, using problem sizing to find the sweet spot per node is one of the first optimizations a user should consider.

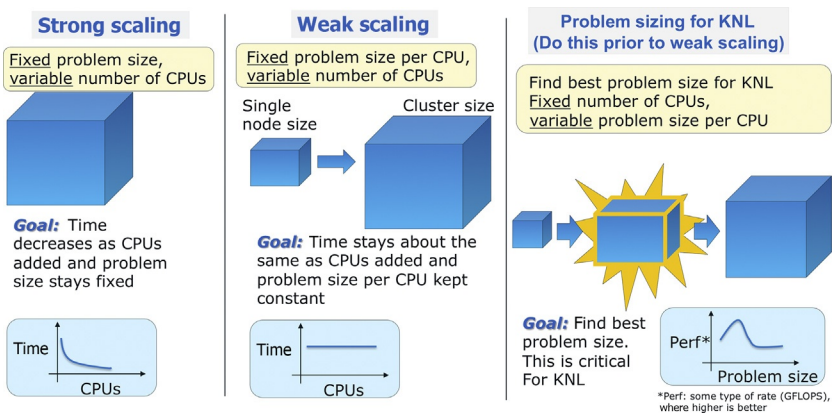


FIG. 25.2

Problem sizing is critical for Knights Landing. Problem sizing needs to be done before weak scaling to get optimal performance.

This sweet spot behavior is due to significantly higher bandwidth of MCDRAM and limited capacity of MCDRAM. MCDRAM on our test system is 16 GB and has bandwidth of ~490 GB/s. DDR memory is 96 GB in size and has ~90 GB/s bandwidth.

RUNNING ON KNIGHTS LANDING: QUADRANT-CACHE MODE

Let us first consider quadrant cluster mode and MCDRAM as cache memory mode (*quadrant-cache* for short). In quadrant cluster mode, when a memory access causes a cache miss, the cache homing agent (CHA) can be located anywhere on the chip, but the CHA is affinitized to the memory controller of that quadrant. In cache mode, the MCDRAM is a memory-side cache. All memory accesses go through the MCDRAM cache to access DDR memory (see Fig. 25.3).

Since all of the Trinity workloads are memory bandwidth sensitive, performance will be better if most of the data is coming from the MCDRAM cache instead of DDR memory. As discussed in the previous section, problem size will be critical for some of the workloads to ensure the data is coming from the MCDRAM cache. For the Trinity workloads, we see two behaviors:

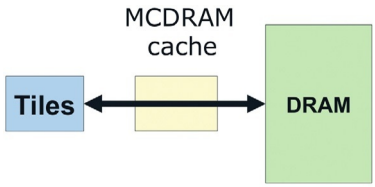


FIG. 25.3

In cache mode, memory accesses go through the MCDRAM cache.

(a) *Cache unfriendly*: Maximum performance is attained when the memory footprint is near or below the MCDRAM capacity and decreases dramatically when the problem size is larger.

(b) *Cache friendly*: Performance does not decrease dramatically when the MCDRAM capacity is exceeded and levels off only as MCDRAM-bandwidth limit is reached. These workloads are able to use MCDRAM effectively even at larger problem sizes.

Fig. 25.4 shows the performance of five of the eight workloads when executed with MPI-only and using 68 ranks (using one hardware thread per core (1 TPC)) as the problem size varies. The other three workloads are a bit different and cannot be drawn in this graph: MiniDFT is a strong-scaling workload with a distinct problem size; GTC's problem size starts at 32 GB and the next valid problem size is 66 GB; MILC's problem size is smaller than the rest of the workloads with most of the problem sizes fitting in MCDRAM. Notice that MiniFE and MiniGhost exhibit the *cache unfriendly or sweet spot* behavior, and the other three workloads exhibit the *cache friendly or saturation* behavior. Fig. 25.5 summarizes the best performance so far for

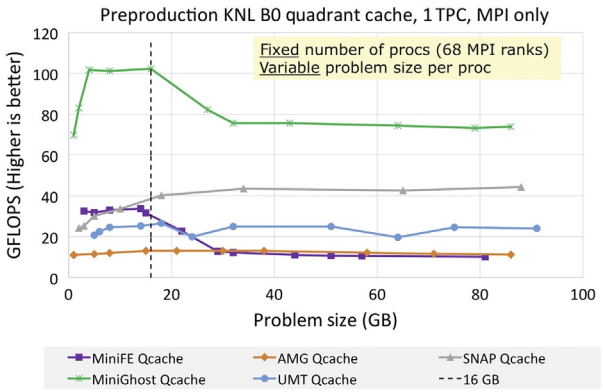


FIG. 25.4

Performance of five Trinity workloads as problem size changes on Knights Landing quadrant-cache mode.

Workload	Mode	Memory Footprint (GB)	Problem Size	Ranks	OMP Threads	HW Threads Per Core (TPC)	Performance (GFLOPS)
MiniGhost	Quad cache	16	336×336×340	68	1	1	102
MiniFE	Quad cache	14	300×300×300	68	1	1	34
AMG	Quad cache	38	119×119×119	68	1	1	13
SNAP	Quad cache	88	64×64×102	68	1	1	44
UMT	Quad cache	18	7×7×7	68	1	1	27
MILC	Quad cache	16	16×32×32×34	68	1	1	83
GTC	Quad cache	32	micell=100, npartdom=1	64	1	1	47
MiniDFT	Quad cache	64	Single node workload	68	1	1	506

FIG. 25.5

Trinity workloads in quadrant-cache mode with problem sizes selected to maximize performance.

all eight of the Trinity workloads. In this figure, problem sizes for one workload cannot be compared with problem sizes for other workloads using only the workload parameters. That is, UMT's $7 \times 7 \times 7$ problem size is different and cannot be compared to MiniGhost's $336 \times 336 \times 340$ problem size. To do the comparison, we need to convert it to memory footprint. The memory footprint in GB is a measured value, not a theoretical size based on workload parameters.

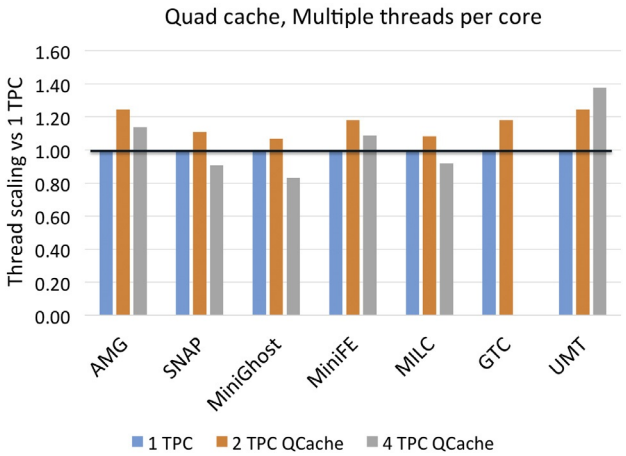


FIG. 25.6

Thread scaling in quadrant-cache mode. GTC was only be executed with 1 TPC and 2 TPC; 4 TPC requires more than 96 GB.

Now that we varied the workload's problem size in quadrant-cache mode, the next thing to consider is the number of hardware threads per core. Knights Landing supports up to four threads per core. Hyperthreading is useful to maximize utilization of the execution units and/or memory operations at a given time interval. If the workload executing at one thread per core is already maximizing the execution units needed by the workload or has saturated memory resources at a given time interval, hyperthreading will not provide added benefit. For Trinity workloads, MiniGhost, MiniFE, MILC, GTC, SNAP, AMG, and UMT, performance improves with two threads per core on optimal problem sizes. UMT also improves with four threads per core. MiniDFT without source code changes is set up to run ZGEMM best with one thread per core; 2 TPC and 4 TPC were not executed. ZGEMM is a key kernel inside MiniDFT. Fig. 25.6 plots the thread scaling of 7 of the 8 Trinity workloads (i.e., without MiniDFT). Fig. 25.7 summarizes the current best performance including the hyperthreading speedup of the Trinity workloads in quadrant mode with MCDRAM as cache on optimal problem sizes.

RUNNING TRINITY ON KNIGHTS LANDING: QUADRANT-FLAT VS QUADRANT-CACHE

Let us now consider flat memory mode with quadrant cluster mode (*quadrant-flat* for short), and how that compares to *quadrant-cache*. To change to this mode, we have to reboot with modified BIOS options. The cluster mode remains the same, but now MCDRAM is treated as addressable memory and can be accessed via a separate NUMA node. If we run the command `numactl --hardware`, two NUMA nodes will

Workload	Mode	Memory Footprint (GB)	Problem Size	Ranks	OMP Threads	TPC	Performance (GFLOPS)	HT Speedup (TPC Speedup vs. 1 TPC)
MiniGhost	Quad cache	16	336×336×340	136	1	2	109	1.07
MiniFE	Quad cache	8	244×244×244	136	1	2	40	1.18
AMG	Quad cache	38	119×119×119	136	1	2	16	1.24
SNAP	Quad cache	34	32×64×68	136	1	2	49	1.11
UMT	Quad cache	18	7×7×7	272	1	4	38	1.38
MILC	Quad cache	16	16×32×32×34	136	1	2	90	1.08
GTC	Quad cache	66	micell=200, npartdom=2	128	1	2	55	1.18
MiniDFT	Quad cache	64	Single node workload	68	1	1	506	1.00

FIG. 25.7

Trinity workloads in quadrant-cache mode when problem sizes and hardware threads per core selected to maximize performance.

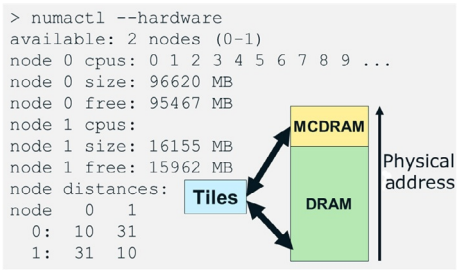


FIG. 25.8

Flat mode showing two separate NUMA nodes when executing `numactl -hardware`.

be shown (see Fig. 25.8). The CPUs and DDR memory are in node 0, and MCDRAM is in node 1. If we run the workload without using numactl, for example, `mpirun -np 68 ./myKNLwkld`, the workload uses memory from the nearest NUMA node first (node 0 in this case, i.e., DDR memory). Defining the NUMA nodes such that the NUMA distance between the MCDRAM and the CPUs is greater than the distance between the DDR and the CPUs is done so that we have to explicitly specify when MCDRAM is used. There a few different ways to use MCDRAM in flat mode.

- Use `numactl --membind 1` to bind the workload to NUMA node 1
  - For example: `numactl --membind 1 ./myScriptToRunOnKNL.sh`

  By binding the workload, the workload is forced to use only the memory designated by that NUMA node (in this case, MCDRAM). If you need more than the MCDRAM memory size, the workload will crash. For example, GTC requires 32 GB to run on a single node. Thus, GTC cannot be run only out of MCDRAM.

- Use `numactl --preferred 1` to *prefer* NUMA node 1
  Now the workload will attempt to use MCDRAM first but will not crash if the memory required is greater than MCDRAM size; this provides a safety net in case memory footprint becomes slightly larger than MCDRAM capacity.
- memKind library (hbwmalloc)—this library enables users to specify which arrays or other blocks of memory should be allocated in MCDRAM.

Tools can help with flat mode:

- Intel® VTune™ Amplifier Memory Object Analysis—determines dynamic and static memory objects and their performance impact to identify candidate data structures for MCDRAM allocation.
- autohbw—Interposer library that comes with memkind. Finds memory allocation calls and automatically allocates memory in MCDRAM if allocation is greater than a given threshold.

Figs. 25.9 and 25.10 show performance of the same five workloads previously plotted in Fig. 25.4; dashed lines indicate quadrant-cache results, and solid lines indicate quadrant-flat results using `numactl --preferred 1`.

MiniGhost performance is very interesting; at lower problem sizes (<16 GB) flat mode is better, but at larger problem sizes (>16 GB), cache mode is better. This performance drop is due to limitations of `numactl --preferred 1`, using other methods to allocate specific memory blocks into MCDRAM such as memkind library may improve performance. MiniGhost bandwidth measurements show the larger problem sizes in quad flat have minimal MCDRAM use when run with `numactl --preferred 1`.

Memory bandwidth was measured to show how bandwidth changes for MiniGhost and MiniFE at different problem sizes. At the 16 GB problem size, MiniGhost in quadrant-cache has MCDRAM bandwidth of 312 GB/s and DDR bandwidth of

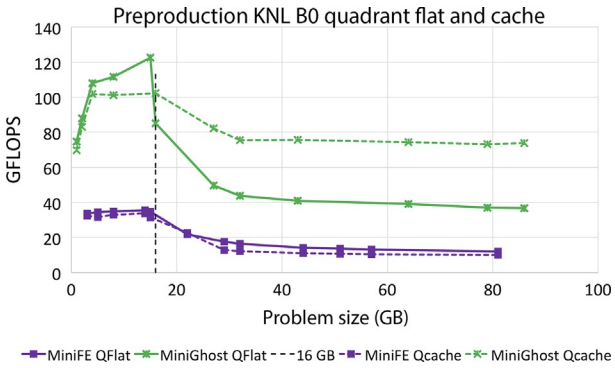


FIG. 25.9

MiniFE and MiniGhost performance in quadrant-flat and quadrant-cache.

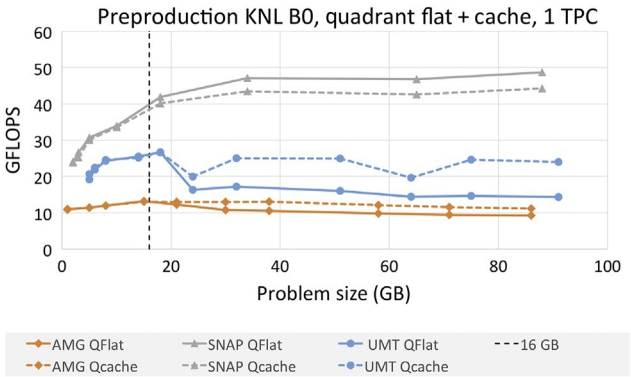


FIG. 25.10

AMG, UMT, and SNAP performance in quadrant-flat vs quadrant-cache.

15 GB/s. At the 64 GB problem size, MiniGhost in quadrant-cache has MCDRAM bandwidth of 214 GB/s and DDR bandwidth of 65 GB/s. In contrast, MiniGhost in quadrant-flat at 64 GB problem size has MCDRAM bandwidth of 0 GB/s and DDR bandwidth of 88 GB/s. Although, we called MiniGhost cache unfriendly due to the "sweet spot" behavior, the workload is still benefiting from MCDRAM cache at large problem sizes in quadrant-cache. MiniFE at large problem sizes does not benefit from MCDRAM as cache. At 16 GB problem size, MiniFE has 267 GB/s MCDRAM bandwidth and 31 GB/s DDR bandwidth. However, at 57 GB problem size, MiniFE in quadrant-cache has 16 GB/s MCDRAM bandwidth and 78 GB/s DDR bandwidth. MiniFE performance at large problem sizes is limited by DDR memory bandwidth.

UMT and AMG are very cache-friendly workloads at small and large problem sizes. However, performance drops at larger problem sizes when using flat mode as the workload's allocated data no longer fits in MCDRAM. SNAP performs better with flat mode at larger problem sizes. SNAP suffers from MCDRAM cache aliasing on large problem sizes similar to MiniGhost at small problem sizes. However in flat mode, there is no aliasing, enabling SNAP to perform better at large problem sizes and MiniGhost to perform better at small problem sizes compared to cache mode.

Consider using flat mode under the following conditions:

- Problem size does not fit in 16-GB MCDRAM and workload is latency bound instead of being bandwidth limited: DDR latency in flat mode will be lower than MCDRAM latency in cache mode if there are excessive MCDRAM cache misses. (see Chapters 4 and 6)
- Workload uses full cache-line streaming stores or partial cache-line (e.g., SSE/AVX ISA) streaming stores

- Streaming stores bypass core cpu-caches and update memory directly, but MCDRAM is a memory-side cache (whenever it is operated as a cache) and cannot be bypassed. MCDRAM cache incurs additional overhead to handle streaming stores: it requires one extra memory read to determine whether a line is already present in MCDRAM (partial cache-line write may also require an extra memory write to fill the line from memory in case of a miss). Hence, if a workload is likely to have a large number of streaming stores, then it may be better to configure MCDRAM as flat, since MCDRAM as flat memory does not have such overheads because it is directly addressable memory same as DDR memory.

- Workload is significantly affected by cache aliasing in cache mode (example: SNAP and MiniGhost). MCDRAM is a direct-mapped cache which means memory accesses that map to same MCDRAM cache-line will conflict causing evictions of data being used and increased conflict misses which will impact performance (see Fig. 25.11)
  - With a direct-mapped cache, capacity misses will occur when the working set size is greater than MCDRAM capacity (as in SNAP case). However, conflict misses can also occur with small problem sizes (less than MCDRAM capacity) depending on where data is allocated in physical memory (as we observed in the MiniGhost case).

Performance comparison between cache mode and flat mode with respect to thread scaling is shown in Fig. 25.12. AMG and UMT are two of our cache-friendly

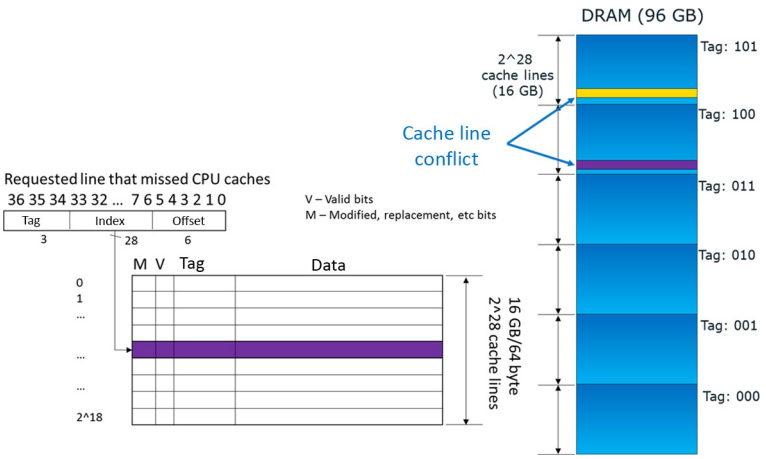


FIG. 25.11

Cache line conflict occurs on Knights Landing with a 16 GB MCDRAM cache if bits 6 to 33 of physical address match. Increased MCDRAM cache misses due to conflicts will impact performance.

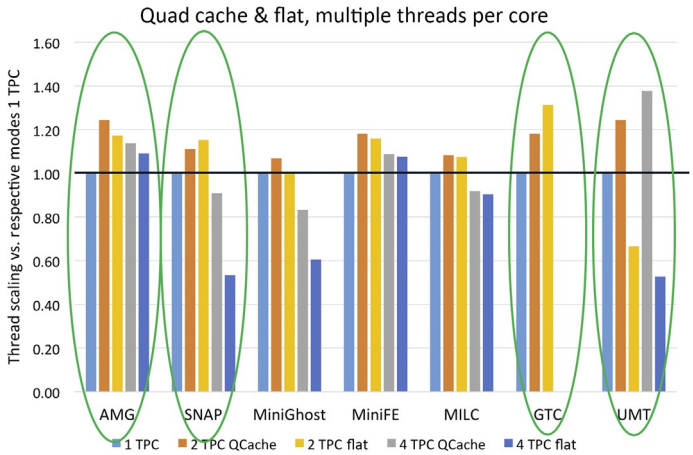


FIG. 25.12

Multiple threads per core scaling in quadrant-flat vs quadrant-cache; *Y*-axis is speedup over respective mode's single-thread performance.

Workload	Mode	Memory Footprint (GB)	Problem Size	Ranks	OMP Threads	HW Threads Per Core	Performance (GFLOPS)	Speedup vs Quad Cache 1TPC
MiniGhost	Quad flat	15	308×308×323	136	1	2	123	1.20
MiniFE	Quad flat	16	307×307×307	136	1	2	42	1.26
AMG	Quad cache	38	119×119×119	136	1	2	16	1.24
SNAP	Quad flat	34	32×64×68	136	1	2	54	1.23
UMT	Quad cache	18	7×7×7	272	1	4	38	1.38
MILC	Quad flat	16	16×32×32×34	136	1	2	91	1.09
GTC	Quad flat	66	micell=200, npartdom=2	128	1	2	62	1.32
MiniDFT	Quad flat	64	Single node workload	68	1	1	521	1.03

FIG. 25.13

Best performance of all 8 Trinity workloads when considering quadrant-cache and quadrant-flat.

workloads; as such, these workloads have better scaling in cache mode than in flat mode. SNAP and GTC in contrast have better scaling in flat mode.

Fig. 25.13 includes the best performance of all 8 Trinity workloads, where best performance was selected between quadrant-cache and quadrant-flat, varying problem size, and varying hardware threads per core. Some workloads perform better in quadrant-cache and others in quadrant-flat.

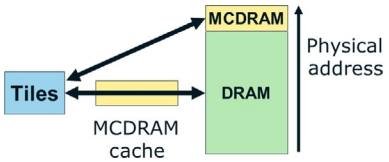


FIG. 25.14

Hybrid mode, part of MCDRAM, is addressable memory and part is memory-side cache.

Hybrid mode

Knights Landing's hybrid mode combines cache mode with flat mode (see Fig. 25.14). This enables advanced optimizations where users can specify which data should be allocated in MCDRAM in flat mode via the memkind library (hbwmalloc calls, see Chapter 3), and it also enables a smaller memory-side cache for remaining data. The baseline versions of the Trinity workloads are not optimized for this mode as no source code changes were done. However, keep in mind that this is an option to consider when optimizing workloads for Knights Landing.

RUNNING ON KNIGHTS LANDING: SNC-4-CACHE VS QUADRANT-CACHE

We started our discussion with quadrant-cache and looked at how changing the memory mode to quadrant-flat impacts performance. Now let us consider the performance impact of changing the cluster mode to SNC-4. SNC-4 creates separate NUMA nodes by dividing up the tiles into four sub-NUMA clusters. In this mode, the tiles, the CHA, and the memory controllers are affinitized within the NUMA node. In cache mode with SNC-4 cluster mode (*SNC-4-cache* for short), the DDR memory and the MCDRAM cache are also divided into the same four sub-NUMA clusters. Fig. 25.15 shows the output of `numactl --hardware`.

Our preproduction Knights Landing has 68 cores, so the number of tiles is not divisible by 4. Thus, two of the sub-NUMA nodes have 9 tiles (18 cores) and two have 8 tiles (16 cores). In this mode, we may consider running with 16 MPI ranks or OpenMP threads to minimize load balancing effects (i.e., use only 64 of the 68 cores). For the following results, we used all 68 cores (18 cores in nodes 0 and 1, 16 cores in nodes 2 and 3).

SNC-4-cache (one thread per core) shows similar performance as quadrant-cache mode; see Fig. 25.16. UMT performance seems to be more stable (less dips) than in quadrant-cache mode. MiniGhost shows higher performance when the problem size is less than or equal to MCDRAM capacity and is similar to or a bit worse when the problem size is greater than MCDRAM capacity.

But, the real story about SNC-4-cache is how the performance improved with multiple threads per core for the cache-friendly workloads. Fig. 25.17 describes

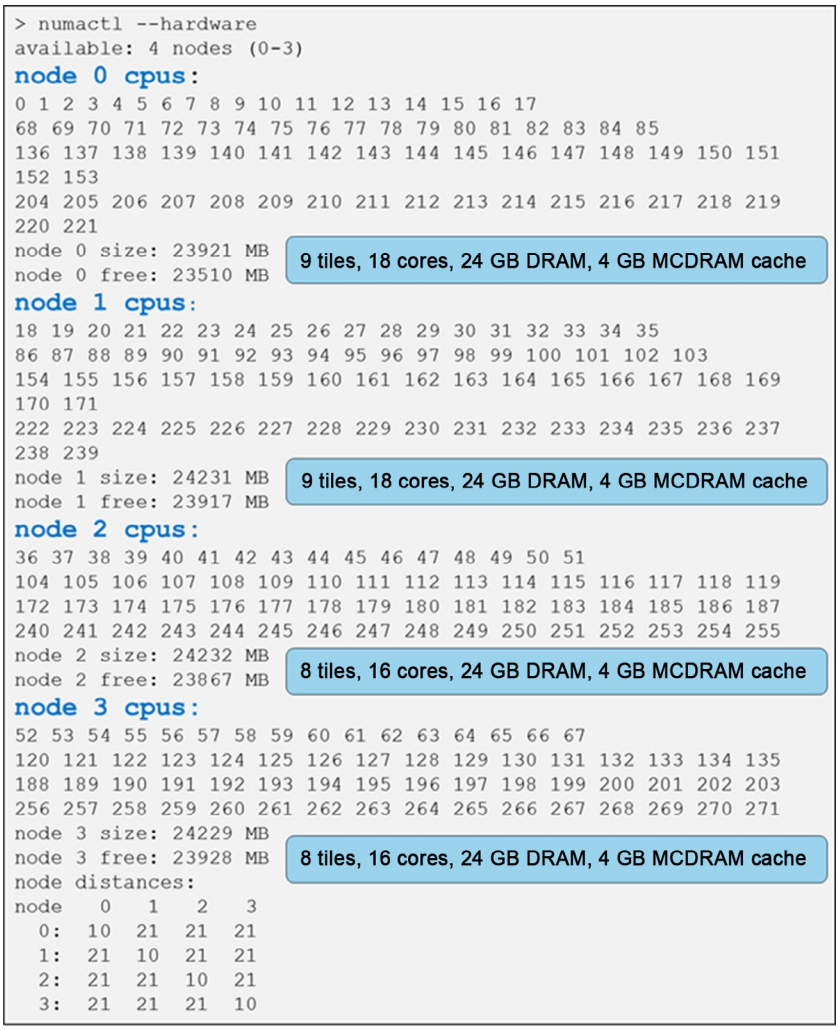


FIG. 25.15

SNC-4-cache mode showing four separate NUMA nodes when executing `numactl --hardware`.

the behavior. The improved thread scaling on the cache-friendly workloads is due to SNC-4 mode decreasing latency and improving distribution of the memory transactions. Fig. 25.18 summarizes the best performance at this point comparing quadrant-cache, quadrant-flat, and SNC-4-cache at problem sizes and threads per core selected to maximize bandwidth and performance.

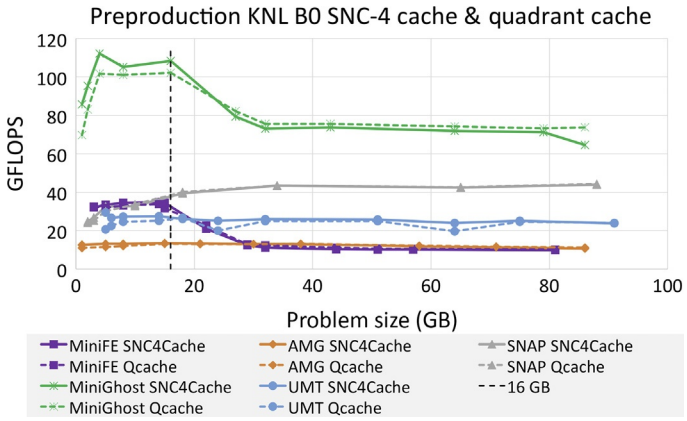


FIG. 25.16

Five of the Trinity workloads comparing performance of SNC-4-cache and quadrant-cache. All workloads executed with one thread per core.

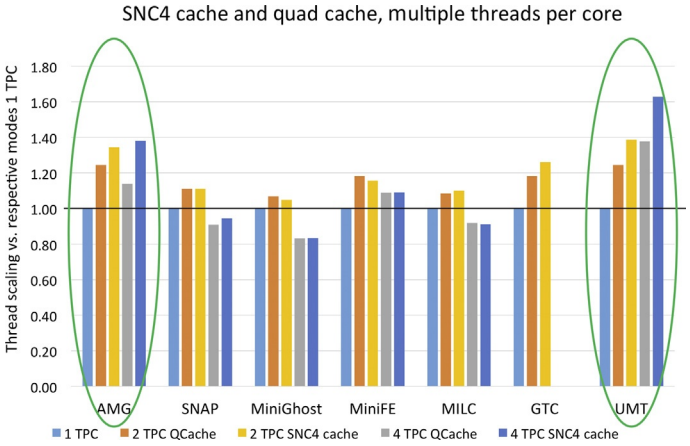


FIG. 25.17

Thread scaling of Trinity workloads, SNC-4-cache vs quadrant-cache.

COMPARING KNIGHTS LANDING PERFORMANCE VS INTEL XEON E5-2697 V4 PROCESSOR

Trinity workloads were executed on multiple cluster modes and memory modes. Some of the data for other modes are still in progress and will be added to http://lotsofcores.com/trinity when data is complete. Fig. 25.19 shows the best performance of all of the modes tested including SNC4-Flat.

Workload	Mode	Memory Footprint (GB)	Problem Size	Ranks	OMP Threads	HW Threads Per Core	Performance (GFLOPS)	Speedup vs Quad Cache 1TPC
MiniGhost	Quad flat	15	308×308×323	136	1	2	123	1.20
MiniFE	Quad flat	16	307×307×307	136	1	2	42	1.26
AMG	SNC4 cache	21	60×60×60	272	1	4	18	1.42
SNAP	Quad flat	34	32×64×68	136	1	2	54	1.23
UMT	SNC4 cache	18	7×7×7	272	1	4	42	1.53
MILC	SNC4 cache	16	16×32×32×34	136	1	2	100	1.21
GTC	Quad flat	66	micell=200, npartdom=2	128	1	2	62	1.32
MiniDFT	Quad flat	64	Single node workload	68	1	1	521	1.03

FIG. 25.18

Best performance of all 8 Trinity workloads when considering quadrant-cache, quadrant-flat, and SNC-4-cache.

Workload	Mode	Memory Footprint (GB)	Problem Size	Ranks	OMP Threads	HW Threads Per Core	Performance (GFLOPS)	Speedup vs Quad Cache 1TPC
MiniGhost	SNC4 flat	8	268×268×272	136	1	2	128	1.25
MiniFE	Quad flat	16	307×307×307	136	1	2	42	1.26
AMG	SNC4 cache	21	60×60×60	272	1	4	18	1.42
SNAP	SNC4 flat	34	32×64×68	136	1	2	55	1.24
UMT	SNC4 cache	18	7×7×7	272	1	4	42	1.53
MILC	SNC4 flat	16	16×32×32×34	136	1	2	108	1.30
GTC	Quad flat	66	micell=200, npartdom=2	128	1	2	62	1.32
MiniDFT	Quad flat	64	Single node workload	68	1	1	521	1.03

FIG. 25.19

Best performance of all different cluster and memory modes tried with optimal problem sizes and optimal threads per core.

Trinity mini-applications with different problem sizes were measured on an Intel Xeon E5-2697 v4 processor to get best performance numbers (albeit performance of different problem sizes had much less impact on the Intel Xeon E5-2697 v4 processor as expected). Knights Landing performance compares well to a two-socket Intel Xeon E5-2697 v4 processor, see Fig. 25.20.

SUMMARY OF OUT OF BOX SECTION

In this section, we discussed some of the different Knights Landing memory modes and cluster modes and provided insights that users should consider in choosing the appropriate modes for their applications. We discussed the importance of problem

Workload	KNL B0 GFLOPS	BDW-EP GFLOPS	KNL Speedup
UMT	42	27	1.57
GTC	62	56	1.11
MILC	108	71	1.51
MiniGhost	128	54	2.35
AMG	18	13	1.39
SNAP	55	57	0.97
MiniFE	42	16	2.67
MiniDFT (small problem)	521	435	1.20
Geomean			1.51

FIG. 25.20

Knights Landing 68 core performance vs a two-socket Intel Xeon E5-2697 v4 processor.

sizing the workload and considering multiple threads per core. Most importantly, we demonstrated the out-of-box performance of Knights Landing. With no source code changes, Knights Landing has 1.5 × geomean performance speedup vs a two-socket Intel Xeon E5-2697 v4 processor both at single node.

OPTIMIZING MiniGhost OpenMP PERFORMANCE

The MiniGhost benchmark was developed to provide a performance proxy for a common high-performance computing physical-simulation paradigm: the application of a difference-stencil computation through multiple time steps across a large multidimensional grid spread across multiple nodes in a cluster. The applied stencil typically inputs a number of points from the grid neighboring a central point in one time step, calculates a value as a function of those points, and outputs the result into the grid for the next time step. The MiniGhost benchmark contains four sample stencils: 5-point 2D, 9-point 2D, 7-point 3D, and 27-point 3D.

Because each node in the cluster works on a subset of the grid, data must be exchanged between the nodes along the boundaries between the grid subsets after each time step to provide the updated neighbor values. The boundary values to be exchanged are also known as the "halo" or "ghost" values, hence the name of the benchmark.

MiniGhost was developed primarily to exercise and profile the MPI communication between nodes. However, as the core count of nodes continues to grow, the importance of intercore and interthread communication becomes more significant. Due to the high number of cores now available in Knights Landing, this chapter focuses on tuning MiniGhost performance using OpenMP on a single node. The results can then be extended to a cluster environment using the existing MPI communication mechanism.

ALGORITHM OVERVIEW

The top-level flow of the MiniGhost benchmark algorithm is shown in Fig. 25.21. After allocating and initializing the data, the main loop iterates over the number of time steps specified. Within each time step, the halo data from the previous step

```
For each timestep t {
  Exchange boundary data between MPI ranks;
  For each 3D variable grid v {
    For each k in 1 to NZ {
      For each j in 1 to NY {
        For each i in 1 to NX {
          v(i, j, k, t+1) =
            stencil_function(v(i, j, k, t));
        }
      }
    }
  }
}
```

FIG. 25.21

Simplified pseudocode for high-level algorithm.

is exchanged between MPI ranks. Then, the selected stencil is applied to each point in each of the 3D "variable" grids. In this chapter, we consider the performance of a typical MiniGhost configuration: the 27-point 3D stencil applied to 40 variable grids, a simulation of heat dissipation.

ORIGINAL IMPLEMENTATION

The main data structure in MiniGhost is the set of N 3D grids ($N = 40$ in our case) and one "work" grid. Each of these $N+1$ grids is the same size, which is defined by the overall problem size divided among all the MPI ranks. In addition, a one-point halo is added around the entire surface of the 3D space. Thus, for a rank assigned to a NX×NY×NZ-size subset of the overall problem, the FORTRAN dimensions of each of the $N+1$ grids are defined as follows:

```
DIMENSION(0:NX+1, 0:NY+1, 0:NZ+1).
```

These grids are either allocated as $N+1$ separate 3D grids in older versions of MiniGhost or as one 4D grid with a fourth dimension of N plus another 3D work grid in newer versions.

After the halo exchanges within each time step of the high-level algorithm showed earlier, the required computations are performed as follows for each of the N variable grids:

- The selected stencil function is calculated at each 3D point in the variable grid. The result is written to the corresponding point in the work grid. An OpenMP pragma is provided on the outer (Z) loop for this calculation.
- When all the new values have been calculated, the work grid is copied back to the variable grid to save the results. The work grid can then be used for the next variable. An OpenMP pragma is provided on the outer (Z) loop for this copy.

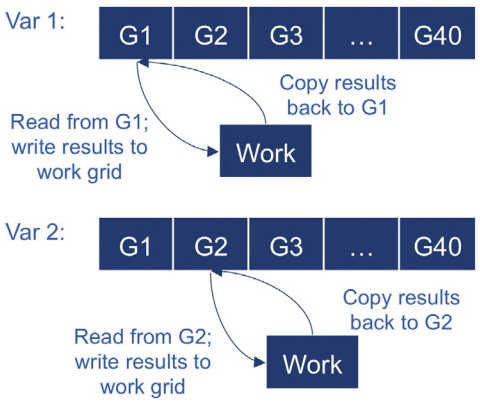


FIG. 25.22

The original code writes results to a temporary work grid and then copies them back into the variable grid Gn.

- A separate function is called to accumulate the "flux" values, which is a measure of the heat dissipation out of the grid. Separate conditionals and loops are used for each face, edge, and corner of the 3D space (older versions did not include edges and corners). There are no OpenMP pragmas in this function.
- Another function is called to accumulate the values remaining inside the grid into a global sum. An OpenMP pragma is provided on the outer (Z) loop for this calculation.

After all N grids have been updated, the flux values and global sum are used to compare to a checksum. The benchmark exits with an error if the relative error is above a maximum threshold.

The storing of temporary results into the work grid and the copying back to the variable grid is illustrated in Fig. 25.22 for the first two variables.

INITIAL CONFIGURATION SIZING, TESTING, AND TUNING

Before any code modification was attempted, we wished to establish a baseline performance value.

Initial sizing and baseline performance

Based on the characterization work reported earlier in this chapter, we found that MiniGhost performs well with a problem size less than the MCDRAM memory size.

The MiniGhost result from Fig. 25.13 is repeated in Fig. 25.23, along with results from running one and four ranks per core at the previously-discussed well-performing problem size of $308 \times 308 \times 323$.

Number of MPI Ranks	GFLOPS
68	122.5
136	122.9 (best)
272	95.4

FIG. 25.23

Baseline MPI-only performance.

Number of OpenMP Threads	GFLOPS
68	55.4 (best)
136	52.3
272	35.8

FIG. 25.24

Baseline OpenMP-only performance.

The next step was to enable the existing OpenMP in MiniGhost. By running the benchmark on the same configuration with OpenMP only (no MPI), we obtained the results in Fig. 25.24. The best OpenMP-only result is significantly worse than the best MPI-only result for this problem size.

Problem-size selection

Perhaps another set of dimensions with a size that still fits into MCDRAM will perform better for OpenMP.

By inspecting the code, we see that all three OpenMP code regions are parallelized along the *Z* axis, so if we set NX to 272 on 68-core processor, the work will be split evenly across OpenMP threads, regardless of how many hardware threads we use per core. We tried several combinations of settings for NX and NY and found the best performance at NX = 640 and NY = 270. The footprint of this size is approximately 14.4 GB, still within the 16 GB MCDRAM size. This size gave the performance shown in Fig. 25.25. The best performance using this problem size is 2.1 × better than the previous best OpenMP-only result but is slightly lower than (0.93 ×) the best MPI-only result.

Number of OpenMP Threads	GFLOPS
68	114.5 (best)
136	111.6
272	84.1

FIG. 25.25

OpenMP-only performance from 40 variable grids of 640 × 270 × 272 units.

Number of MPI Ranks	GFLOPS
68	62.1 (best)
136	49.9
272	51.4

FIG. 25.26

MPI-only performance from 40 variable grids of $640 \times 270 \times 272$ units.

For completeness, we used the new problem size from Fig. 25.25 to run the benchmark using only MPI and obtained the results in Fig. 25.26. The best result using MPI-only with this problem size is significantly worse than the best result from the problem size used in Fig. 25.23.

In summary, we found that enabling the existing OpenMP directives in the code and retuning the problem size produced OpenMP-only performance almost (but not quite) as good as the MPI-only results we obtained earlier. Next, we will delve into the benchmark code to look for OpenMP tuning opportunities.

CODE OPTIMIZATIONS

In this section, we describe three changes made to the original code to increase OpenMP performance: copy elimination, more parallelization, and data-access reduction.

Copy elimination

Recall from Fig. 25.22 that each variable grid is read during computation, the results are written to the work grid, and then the work grid is copied back to the variable grid. The copy does no computational work but consumes a significant amount of time because each variable grid is larger than a shared L2 cache, causing cache misses as the results are read back from MCDRAM.

To eliminate the copies, we created 41 pointers to 3D arrays. Forty are initialized to point to the 40 variable grids, and one points to the work grid. The location of the variable grid data will be defined by the pointers and no longer by the original location in the allocated array. Now, instead of copying the data from the work grid to the variable grid after a computation, the pointers are simply swapped. What was once the work grid is now the result grid for that variable, and what was once the input grid becomes the work grid for the next variable, since the old result is no longer needed. This "shallow-copy" scheme is illustrated in Fig. 25.27. This optimization produced a $1.3\times$ speedup over the best previous result in Fig. 25.25.

Parallelization of serial code

An analysis of the code using Intel® VTune™ Amplifier showed a significant amount of serial time in the OpenMP runs spent in the `FLUX_ACCUMULATE` subroutine. This routine consists of a number of small loops, some of which iterate over the faces

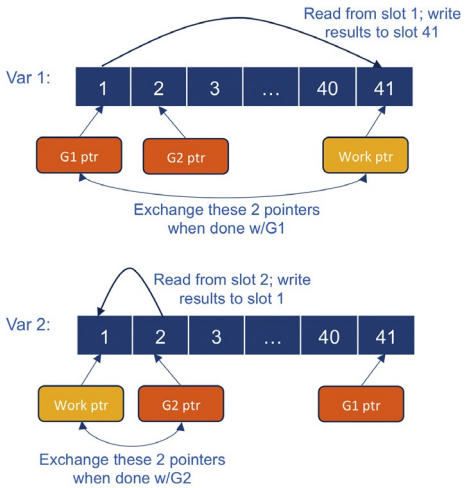


FIG. 25.27

The "shallow-copy" scheme, illustrated here for the first two variables, eliminates the need to copy the results from the work grid to the variable grid. Compare to Fig. 25.22.

of each 3D grid, some over edges, and some corners. Each loop is predicated by a conditional expression that tests whether the current MPI task contains the face, edge, or corner. We can parallelize each loop separately, but this produces a large number of synchronization points with little work in each. Rather, we refactored the flux-accumulate code by creating just one outer loop for each dimension and moving the conditionals and inner loops within them. The outer loops were parallelized with OpenMP, and all the outer loops were placed within one OpenMP parallel region. An example of two refactored loops is shown in Fig. 25.28. In addition, the Intel® compiler generates any required vector masking and gather instructions to enable vectorization within the inner loops. This parallelization resulted in a $1.8\times$ speedup in addition to the performance due to the shallow-copy scheme described previously.

Data-access reduction

The last OpenMP region in the original top-level loop creates a sum of all the values within a given grid. This sum, along with the result from the flux accumulation, is used as a minimal check for accuracy of the result after each time step.

Even though the summation is very simple, it still requires traversing the entire grid (not including the halo region). As with the deep copy that we eliminated previously, this causes many cache misses and contributes a significant amount of time.

To eliminate this traversal, we simply move the summation within the computation step as shown in Fig. 25.29.

This modification produced the final performance shown in Fig. 25.30. The performance is $1.4\times$ over the result from the flux accumulation improvements. Note that the best performance now uses all four hardware threads on each core.

Original code:

```
IF ( MYPZ == 0 )  THEN
   DO J = 1, NY
      DO I = 1, NX
         FLUX_OUT(IVAR) = FLUX_OUT(IVAR) + &
           ( ( GRID(I,J,1)) / DIVISOR )
      END DO
   END DO
END IF
IF ( MYPZ == ( NPZ - 1 ) ) THEN
   DO J = 1, NY
      DO I = 1, NX
         FLUX_OUT(IVAR) = FLUX_OUT(IVAR) + &
           ( ( GRID(I,J,NZ)) / DIVISOR )
      END DO
   END DO
END IF
```

Refactored code:

```
RESULT = 0
!OMP PARALLEL
…
!$OMP DO REDUCTION (+: RESULT)
DO J = 1, NY
   DO I = 1, NX
      IF ( MYPZ == 0 )  THEN
         RESULT = RESULT + ( ( GRID(I,J,1)) / DIVISOR )
      END IF
      IF ( MYPZ == ( NPZ - 1 ) ) THEN
         RESULT = RESULT + ( ( GRID(I,J,NZ)) / DIVISOR )
      END IF
   END DO
END DO
…
!$OMP END PARALLEL
FLUX_OUT(IVAR) = FLUX_OUT(IVAR) + RESULT
```

FIG. 25.28

Original and refactored code for adding OpenMP to the flux accumulation.

```
      LSUM = 0.0
!$OMP PARALLEL DO &
  PRIVATE(SLICE_BACK, SLICE_MINE, SLICE_FRONT, RES) &
  REDUCTION(+:LSUM) !COLLAPSE(2)
     DO K = 1, NZ
        DO J = 1, NY
           DO I = 1, NX
              SLICE_BACK =  GRID(I-1,J-1,K-1) + …
              SLICE_MINE =  GRID(I-1,J-1,K)   + …
              SLICE_FRONT = GRID(I-1,J-1,K+1) + …
              RES = (SLICE_BACK + SLICE_MINE + SLICE_FRONT) &
                / 27.0
              WORK(I,J,K) = RES
              LSUM = LSUM + RES
           END DO
        END DO
     END DO
```

FIG. 25.29

Summation added to computation loop. The new variable LSUM is the local sum for this grid and is later added to the global sum.

Number of OpenMP Threads	GFLOP/S
68	256.8
136	323.2
272	380.5 (best)

FIG. 25.30

Final performance of the modified MiniGhost code on the $640 \times 270 \times 272$ problem size.

The overall speedup from the three code modifications is $3.3\times$.

Effect of OpenMP code changes on an Intel Xeon E5-2697 v4 processor

We ran the newly tuned code with same problem size on a two-socket Intel Xeon E5-2697 v4 processor for comparison. Since the two sockets are on different NUMA domains, the best configuration we found was a "hybrid" MPI+OpenMP run using two MPI ranks with 18 OpenMP threads each (one for each core on each Xeon socket). This setting achieved 114 GFLOPS, $2.1\times$ better than the MPI-only result of 54 GFLOPS shown previously in Fig. 25.20, showing that the OpenMP tuning improves Xeon performance as well. The best Knights Landing result from Fig. 25.30 (380.5 GFLOPS) is now $3.3\times$ higher than the best Xeon result. (Compare this to the $2.4\times$ Knights Landing MPI-only speedup over Xeon shown previously.)

MiniGhost OPTIMIZATION SUMMARY

We discussed two sets of improvements over the baseline MPI-only execution of the MiniGhost benchmark: configuration changes and code changes. The configuration changes consisted of enabling OpenMP and selecting a problem size that was tuned for the MCDRAM capacity and number of hardware threads on the node. These changes enabled an OpenMP-only run that performed almost as well as the best MPI-only run. Code changes were required to further improve the OpenMP performance. These code changes consisted of copy elimination, more parallelization, and data-access reduction. These changes gave a $3.3\times$ speedup over the unmodified OpenMP-only run. Compared to the best MPI-only run, the changes provided a $3.1\times$ increase in GFLOPS throughput on single-node execution. Executing code with the same changes also produced a speedup on a two-socket Intel Xeon E5-2697 v4 processor, but only $2.1\times$, showing that the OpenMP tuning had a greater effect on Knights Landing performance.

SUMMARY

In this chapter, we discussed out-of-box performance of Trinity workloads, no source code changes with MPI-only. We presented results on four different cluster modes and memory modes with hyperthreading. We provided insights on when to use the different modes and discussed how problem sizing is critical for Knights Landing workloads. Knights Landing out-of-the-box performance with MPI-only was

compared to Intel Xeon E5-2697 v4 processor (codenamed Broadwell) out-of-the-box performance with MPI-only showing a geomean performance speedup of 1.5 ×. We then went a bit deeper on one of the Trinity workloads, MiniGhost, to show how OpenMP optimizations can further boost performance, resulting in a 3.1 × speedup over the best MPI-only results on Knights Landing and a 3.3 × speedup over a hybrid MPI+OpenMP configuration on a two-socket Intel Xeon E5-2697 v4 processor. This work was done on preproduction Knights Landing processors. We will share additional results at http://lotsofcores.com/trinity.

FOR MORE INFORMATION

Here are some additional reading materials we recommend related to this chapter.

- Additional information related to this chapter (ad further work on Trinity applications are done), http://lotsofcores.com/trinity.
- Trinity Supercomputer, http://www.lanl.gov/projects/trinity/.
- National Energy Research Scientific Computing Center (NERSC) Cori, http://www.nersc.gov/users/computational-systems/cori/.
- Trinity workloads, http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/.
- MCDRAM tools https://software.intel.com/articles/mcdram-high-bandwidth-memory-on-knights-landing-analysis-methods-tools.
- Stencil Code, http://wikipedia.org/wiki/Stencil_code.
- Stencil computation optimization and autotuning on state-of-the-art multicore architectures, http://dl.acm.org/citation.cfm?id=1413375.

CHAPTER

Quantum chromodynamics

26

What is new with Knights Landing in this chapter?

Knights Landing proves easier to get performance than on Knight Corner. MCDRAM boosts performance in both cache and flat memory modes, and software prefetching is no longer required.

Lattice quantum chromodynamics (LQCD) is an important application for calculations in Nuclear and High Energy particle physics, and LQCD codes are frequently among the first to be ported to new generations of supercomputers. In this chapter, we explore the performance of LQCD simulations on Knights Landing processing nodes booting in self-hosted mode. This chapter demonstrates the following important aspects of the Knights Landing architecture:

- The MCDRAM subsystem provides a nearly 3 × speed improvement compared to DDR4 memory.
- The wider AVX-512 instructions in combination with multiple threads per core on Knights Landing are able to reasonably utilize the higher MCDRAM bandwidth to substantially boost the performance.
- So far, utilizing multiple threads per core on Knights Landing seems to eliminate the need for software prefetching to hide memory access latencies.

LQCD

QCD (quantum chromodynamics) is the theory that describes the nuclear strong force that binds matter together. LQCD theories are mathematically well-defined reformulations of QCD, suitable for numerical solution by computers, and can be used for calculations in the so-called nonperturbative regime, where analytical calculations via other methods, such as via perturbation theory, break down. The numerical results are extremely important for the fields of high energy and nuclear physics, where they are used to try and constrain the parameters of, and look for physics beyond the Standard Model of particle interactions and to investigate the structure and allowed states of hadronic and nuclear matter.

The need for ever more precise and increasingly complex QCD calculations has kept LQCD simulations running in some form or other on most supercomputer

Intel Xeon Phi Processor High Performance Programming. http://dx.doi.org/10.1016/B978-0-12-809194-4.00026-0

architectures at HPC centers around the world, and will likely continue to keep LQCD simulations running on new hardware into the foreseeable future. As an example, in 2014, LQCD calculations were responsible for consuming 13% of the computer resources at the U.S. Department of Energy National Energy Research Scientific Computing Center (NERSC). Part of the motivation for our investigation described in this chapter is the observation that cost and performance are now favorable for the use of general-purpose hardware like Intel® Xeon Phi™ processors. As a large consumer of HPC resources, it is important to ensure that LQCD software runs as efficiently as possible to maximize performance and minimize energy efficiency.

LQCD is an unusual HPC application in that it has well understood performance models that reflect the runtime behavior of some of its key kernels on various computer architectures. This knowledge along with platform-specific optimizations has been incorporated into the QPhiX library and QPhiX Codegen code generator.

THE QPhiX LIBRARY AND CODE GENERATOR

The QPhiX library implements the Wilson and Clover Dirac Operators along with basic solvers such as Conjugate Gradient, BiCGStab, and iterative refinement for mixed precision. It is a C++ library that basically remains unchanged across architectures. The QPhiX code generator was written to abstract away the vector intrinsics and allow the inclusion and flexible placement of software prefetch instructions.

The Dslash (D, $A^{-1}D$) operators are the core components that have driven the data structure, threading, and vectorization design decisions in the code generator and library. These design decisions have been distilled into the "dslash" micro benchmark kernel used in this chapter.

The allocation of responsibilities between the QPhiX library and code generator is shown in Fig. 26.1.

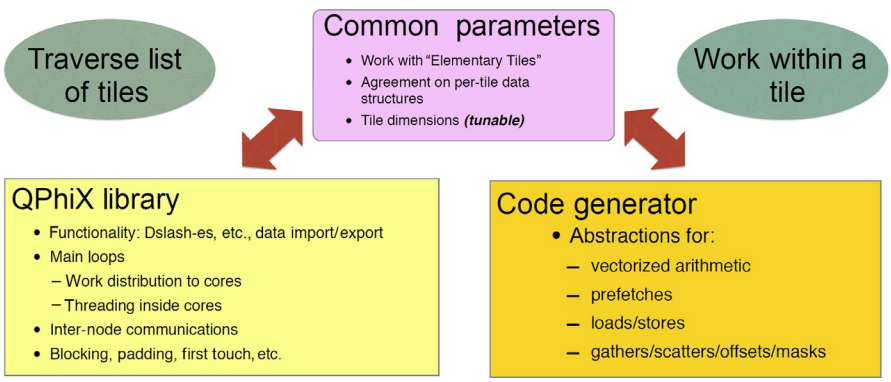


FIG. 26.1

Allocation of responsibilities between the QPhiX library and code generator.

In this chapter, we provide a brief introduction to the Dslash operator. Chapter 9 in volume 2 of *High-Performance Parallelism Pearls* entitled, "*Wilson-Dslash Kernel From Lattice QCD Optimization*" contains a more detailed description of the operator and the design decisions made to map to the Intel Xeon Phi architecture, and we recap only the parts of that discussion which are required to understand our results and performance estimates. See *For More Information* at the end of this chapter on where to download the code discussed in this chapter.

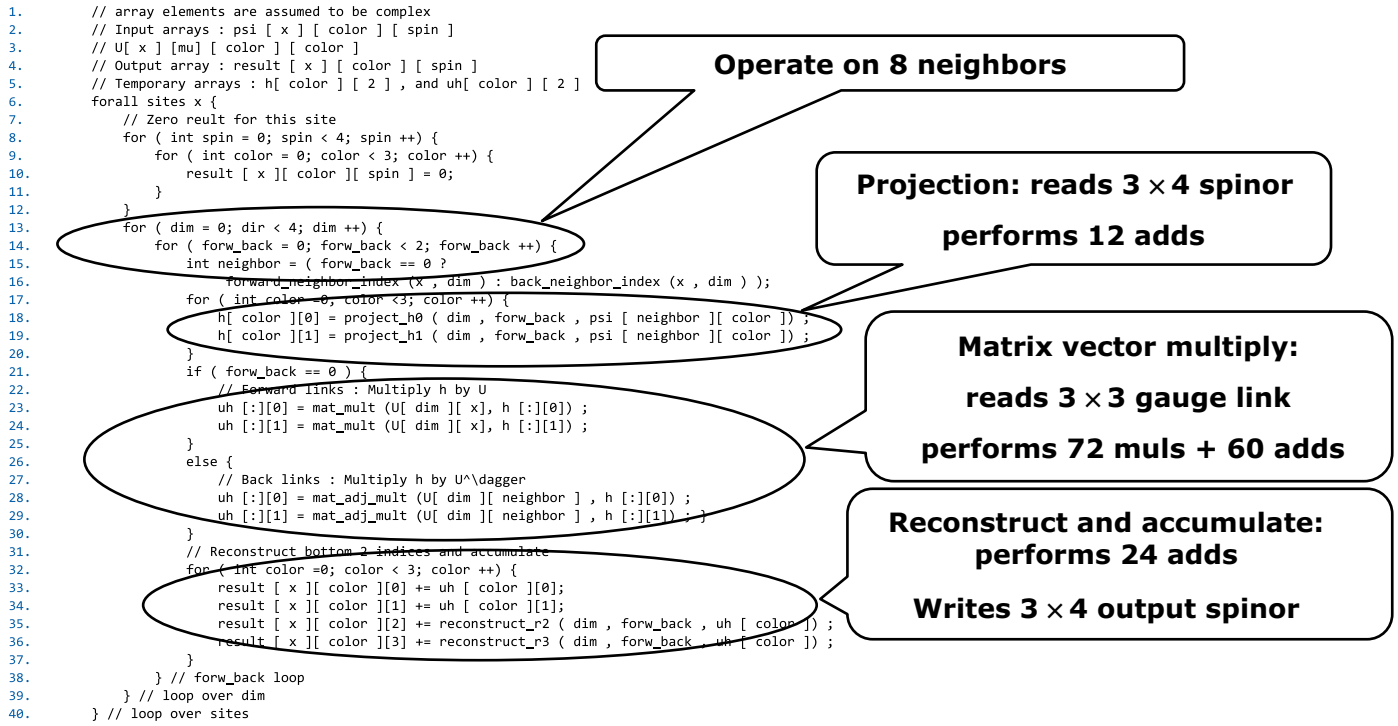
WILSON-DSLASH OPERATOR

The Wilson-Dslash operator, is a nearest neighbor stencil in four dimensions, with the added features that the neighboring sites contain small vectors called spinors which are made up of 4 sets of complex 3 vectors, and the links between the sites contain 3×3 unitary matrices. The small matrices are parallel transporters, which allow relating a spinor's living on sites at opposite ends of a link to each other. To evaluate the Wilson-Dslash at any one site, each of the 4 sets of 3 vectors in each neighboring spinor must be multiplied by this small matrix from the link between the central point and the neighboring site in question, before the components of the stencil can be accumulated. The spin structure of the Dirac operator is such that one can perform spin-projection operations on the neighboring spinors reducing them to just 2 sets of 3 vectors. One then needs to only multiply these two sets with the link matrix. A pseudocode for the algorithm from High-Performance Parallelism Pearls volume 2 is shown in Fig. 26.2.

Typically, the lattice sites are also checkerboarded, and one can evaluate Dslash on all the sites of a given checkerboard color in parallel. The defining characteristics which go into estimations of the arithmetic intensity of evaluating the Dslash operator on a given site are: (a) the number of neighboring spinors that are already in the cache (Spinor Reuse) and (b) the size of the matrices on the links (Gauge Compression). As shown in Fig. 26.3, by traversing the lattice in the right order, if a sufficiently large cache is available (or if cache-blocking is employed), one can in principle have all but one of the neighbors (appearing at the top in Fig. 26.3) in cache, needing to bring in only 1 spinor from DRAM. Due to the checkerboarding, there is no reuse of the link matrices, since the incoming link in each direction is the opposite color to the outgoing one.

Since the gauge fields are members of the SU(3) group, fundamentally they have only eight real parameters, allowing a large degree of compression from the 3×3 complex matrix representation. However, reconstructing from the 8 basic parameters involves the use of sine, cosine, and other functions, so in this chapter, we will stick to a 2-row compression where the third row can be easily recovered, using a vector cross product.

Evaluating Dslash on a site takes 1320 floating-point operations. The components come from spin-projection, matrix multiplication of 2 complex 3 vectors, and an accumulation of the results (specifically, Projection + matrix-vector multiply




```
1.        // array elements are assumed to be complex
2.        // Input arrays : psi [ x ] [ color ] [ spin ]
3.        // U[ x ] [mu] [ color ] [ color ]
4.        // Output array : result [ x ] [ color ] [ spin ]
5.        // Temporary arrays : h[ color ] [ 2 ] , and uh[ color ] [ 2 ]
6.        forall sites x {
7.            // Zero reult for this site
8.            for ( int spin = 0; spin < 4; spin ++) {
9.                for ( int color = 0; color < 3; color ++) {
10.                   result [ x ][ color ][ spin ] = 0;
11.               }
12.           }
13.           for ( dim = 0; dir < 4; dim ++) {
14.               for ( forw_back = 0; forw_back < 2; forw_back ++) {
15.                   int neighbor = ( forw_back == 0 ?
16.                       forward_neighbor_index (x , dim ) : back_neighbor_index (x , dim ) );
17.                   for ( int color =0; color <3; color ++) {
18.                       h[ color ][0] = project_h0 ( dim , forw_back , psi [ neighbor ][ color ]) ;
19.                       h[ color ][1] = project_h1 ( dim , forw_back , psi [ neighbor ][ color ]) ;
20.                   }
21.                   if ( forw_back == 0 ) {
22.                       // Forward links : Multiply h by U
23.                       uh [:][0] = mat_mult (U[ dim ][ x], h [:][0]) ;
24.                       uh [:][1] = mat_mult (U[ dim ][ x], h [:][1]) ;
25.                   }
26.                   else {
27.                       // Back links : Multiply h by U^\dagger
28.                       uh [:][0] = mat_adj_mult (U[ dim ][ neighbor ] , h [:][0]) ;
29.                       uh [:][1] = mat_adj_mult (U[ dim ][ neighbor ] , h [:][1]) ; }
30.                   }
31.                   // Reconstruct bottom 2 indices and accumulate
32.                   for ( int color =0; color < 3; color ++) {
33.                       result [ x ][ color ][0] += uh [ color ][0];
34.                       result [ x ][ color ][1] += uh [ color ][1];
35.                       result [ x ][ color ][2] += reconstruct_r2 ( dim , forw_back , uh [ color ]) ;
36.                       result [ x ][ color ][3] += reconstruct_r3 ( dim , forw_back , uh [ color ]) ;
37.                   }
38.               } // forw_back loop
39.           } // loop over dim
40.       } // loop over sites
```

FIG. 26.2

Annotated Dslash pseudocode.

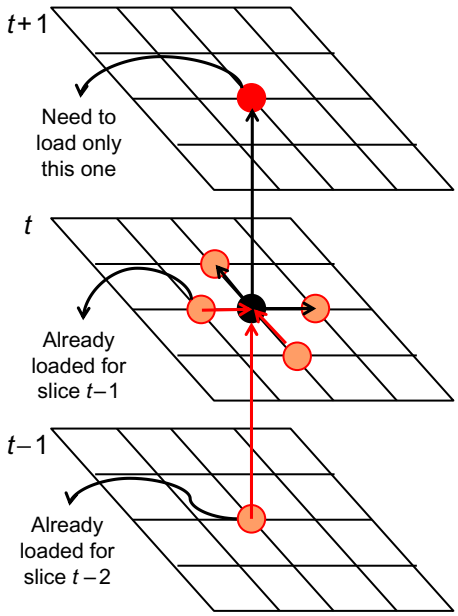


FIG. 26.3

Memory traffic for the Wilson-Dslash operator. Neighbors other than the one pictured at the top are expected to be already in the cache; only the $t+1$ neighbor (at the top here) needs to be brought in from memory (DDR). The gauge links also need to be read and the central point written.

$+\text{reconstruction} = 8 \times 12 + 8 \times 2 \times 66 + 7 \times 24 = 1320$). We count only the necessary FLOPs and ignore the extra operations needed to restore a 2-row compressed gauge field to its full 3×3 form.

The corresponding memory traffic involves reading in 8 gauge links, $8 - \text{R}$ spinors where R is the Spinor Reuse factor, indicating how many neighbors are already in the cache, and the writing of 1 spinor (which without streaming stores may need a read-for-ownership to the cache). Using a spinor size of 24 real numbers (4 sets of 3 complex) and a gauge size of 9 complex numbers (6 with 2-row compression), and assuming that we use 32-bit floating-point numbers, we arrive at the table of arithmetic intensities shown in Fig. 26.4. We can see that for these considerations, the arithmetic intensity can vary between 0.86 and 2.29 flop/byte. This intensity is significantly lower than machine flops/byte ratio on most commonly available systems making this a memory bandwidth limited kernel.

The efficient vectorization of Dslash operator requires data rearrangements to avoid gather/scatters from multiple cachelines. We chose an optimized AoSoA data layout, which allows vectorization over lattice sites. Hence, instead of having an outer array with an element for each site in our lattice, we instead have a number of blocks (outer_blocks) of spinor data structures which contain an inner array of length SOALEN sites. Ideally, we would like to have SOALEN a perfect multiple

Spinor reuse	Gauge compression*	Streaming stores	Arithmetic intensity (flops/byte)
0	No	No	0.86
0	No	Yes	0.92
7	No	No	1.53
7	No	Yes	1.72
7	Yes	No	1.96
7	Yes	Yes	2.29

FIG. 26.4

Wilson-Dslash arithmetic intensity (* Flop/s/byte does not include the flop/s required for gauge decompression).

```
// Ngy is the Y dimension of the XY block used
//  in vectorization
// Gauge field: 8 directions, 3x3 matrices,
//  2 complex components, Inner lattice
// pre-gather all the Y-strips
float Gauge[outer_blocks/ Ngy ][8][3][3][2][Ngy*SOALEN];
// Spinor: 4 sets of 3 vectors,
// each with 2 complex components,
// and inner  lattice
float Spinor[outer_blocks][4][3][2][SOALEN];
```

FIG. 26.5

The Gauge and Spinor data-types.

of vector width on the underlying architecture to eliminate the need for gathers and scatters. However, for efficient implementation, SOALEN needs to be a factor of the *X*-dimension of the lattice after even-odd checkerboarding. This requires SOALEN to be as small as possible to support more general *X*-dimensions for the lattice. As a balance, we use SOALEN equal to or a factor of vector width and use *XY* blocking to fill the full vector width. The tile has length SOALEN in the *X*-direction, and Ngy in the *Y* dimension. We require Ngy $\times$ SOALEN to equal the vector length. Thus, we support SOALEN $=$ 4, 8, or 16 for single precision and SOALEN $=$ 2, 4, or 8 for double precision. Additionally, our gauge field data structure is read-only, and correspondingly, we could essentially carry out the gathering of the *XY* blocks in advance, giving the key data structures shown in Fig. 26.5.

CONFIGURING THE QPhiX CODE GENERATOR

Our work on the previous generation Intel Xeon Phi coprocessors, code named Knights Corner, made the transition to the newer Knights Landing processors straight-forward. In this chapter, we configured the QPhiX code generator to utilize AVX-512 instructions and experimented with various software prefetching options implemented for Knights Corner.

Our work on the previous generation Intel Xeon Phi coprocessors, code named Knights Corner, made the transition to the newer Knights Landing processors straight-forward.

The QPhiX code generator was designed to generate a linear stream of compiler intrinsic-based C code. It assumes an infinite number of vector registers, internally referred to as FVec, which has worked well in practice and keeps the generated code simple. Fvec's are C++ classes that express the notational concept of a vector register. Succinctly, the code generator:

- Utilizes scope, via {}, to deal with different directions.
- Generates simple if(){} else {} constructs.
- Does not use loops or subfunction calls.
- Provides a mechanism for declaring symbol names and, when needed, function declarations.

The operation of the main Dslash kernel generation is conceptually straight-forward as it: (1) creates objects for registers and addresses, (2) creates instruction objects that are put into a C++ vector of pointers to these objects—referred to as the InstVector array, and (3) the generator iterates through the InstVector array calling each objects serialize() method. Periodic and antiperiodic boundary conditions are handled by setting spatial and temporal coefficients via a set of vectors in the QPhiX library (not discussed in this chapter).

The kernel generation process is illustrated in the code snippet shown in Fig. 26.6.

The dslash.cc file defines the body of the code. The pseudocode for the Wilson-Dslash operator can be seen in Fig. 26.2. Most of the functions are straightforward: project, multiply, and reconstruct. However, complications arise in:

- Deciding which neighbors to accumulate. (This is done via an accumulate[] array, which has eight elements (e.g., four forward and four backward dimensions).)
- Handling the *X* and *Y* directions plus masking.
- Handling masking in software for non-Intel Xeon Phi systems.

```
// codegen.cc
void dumpIVector(InstVector& ivector, string filename)
{
   ofstream outfile(filename.c_str());
   for(int i=0; i < ivector.size(); i++) {
   outfile << ivector[i]->serialize() << endl;
}
outfile.close();
```

FIG. 26.6

Example codegen snippet.

All these complications were already handled for Knights Corner and minimal changes were required to convert the dslash.cc source code to utilize the AVX-512 instructions as shown in the Fig. 26.7 code snippet. We could already use a vector length of 16 because of our work on the Knights Corner coprocessor.

Two files, inst_dp_vec8.cc and inst_sp_vec16.cc, were also modified to emit AVX-512 intrinsics when the C preprocessor macro AVX-512 is defined. Following is an example from the single-precision inst_sp_vec16.cc source file (Fig. 26.8).

```
#if VECLEN == 16
#ifdef AVX-512
std::string ARCH_NAME="AVX-512";
#else
std::string ARCH_NAME="mic";
#endif
…
std::string ARCH_NAME="scalar";
#endif
#elif PRECISION == 2
#if VECLEN == 8
#ifdef AVX-512
std::string ARCH_NAME="AVX-512";
…
```

FIG. 26.7

dslash.cc code changes.

```
if(streaming) {
   #ifdef AVX-512
        buf << "_mm512_stream_ps((void*)"
            << a->serialize() << ","
            << v.getName() <<  ");" <<endl;
   #else
        buf << "_mm512_storenrngo_ps((void*)"
            << a->serialize() << ","
            << v.getName() <<  ");" <<endl;
   #endif
}
else {
        buf << "_mm512_extstore_ps("
            << a->serialize() << "," << v.getName()
            <<  ", " << downConv << ", _MM_HINT_NONE);"
            <<endl;
}
```

FIG. 26.8

Example code modification from qphix-codegen/inst_sp_vec16.cc.

This code snippet emits the correct intrinsic depending on if streaming stores are enabled and the preprocessor flag definitions. Depending on the value of streaming and the AVX-512 macro, the intrinsic emitted will be:

- **_mm512_stream_ps**: Knights Landing-specific streaming store.
- **_mm512_storenrngo_ps**: Knights Corner-specific streaming store.
- **_mm512_extstore_ps**: If streaming stores are disabled or not desired, perform a regular store with optional down convert if required.

THE EXPERIMENTAL SETUP

The experimental setup for each of the systems used to run our kernel is described in Fig. 26.9.

In all our experiments, we utilized the Wilson-Dslash microbenchmark kernel for performance measurements. Unless otherwise mentioned, we used a problem size of $32 \times 32 \times 32 \times 96$ utilizing single-precision arithmetic. Average Performance over four microbenchmark runs are reported except for the Knights Corner results where the first, warm-up, run was skipped. Streaming stores were used. We found no significant performance differences between compiling our microbenchmark with v15 or v16 versions of the Intel C/C++ Compiler.

The Knights Landing environment was setup via the following environment variables:

- KMP_AFFINITY=compact, granularity=fine
- KMP_PLACE_THREADS was used to set the number of threads per core used.

We used "`numactl -m 1 - <app cmdline>`" (see Chapter 3) to allocate in MCDRAM when testing the MCDRAM in flat mode and cluster mode not set to an SNC mode.

Code name	Knights Landing	Knights Corner	Haswell EP
Full name	B0 preproduction	Xeon Phi 7120P	Xeon E5-2697v3
Cores	68 cores @ 1.4 GHz	61 cores @ 1.3GHz	2×14 cores @ 2.3 GHz
Memory	6×16GB DDR4 2400 MHz	16 GB GDDR	8×8GB DDR4 2133 MHz
Default MCDRAM configuration	Size: 16 GB Memory mode: Flat Cluster mode: Quadrant (QUAD)	–	–
Other details:		icache_snoops_off, timer = tsc	Dual Socket Xeon system

FIG. 26.9

Experimental setup.

RESULTS

The dslash microbenchmark was run on the target system to collect results. The results reported are based on the average time to perform 100 calls to the Dslash Operator. For this chapter, we report results averaged over four runs.

Our reported times include neither the time to initialize the problem-specific global state, allocate memory for the required data structures such as gauge fields and Spinors, nor the time to initialize the data structures with pseudorandom numbers. The Dslash results were validated with a simple checksum test for correctness.

OVERALL PERFORMANCE

Fig. 26.10 shows the performance across Intel® Xeon® processors code named Haswell, Knights Corner, and Knights Landing for both single- and double-precision microbenchmarks. We also show performance sensitivity to different values of SOALEN. In general, we would always use the smaller value for SOALEN if it performs better and switch to a higher value if the problem size allows and the higher value of SOALEN buys more performance. For instance, on a Haswell system, we would always use SOALEN = 2 for double precision and 4 for single precision. However, on Knights Landing, we would use SOALEN = 4 in single precision only when the checker boarded X-dimension of the lattice is not a multiple of 8.

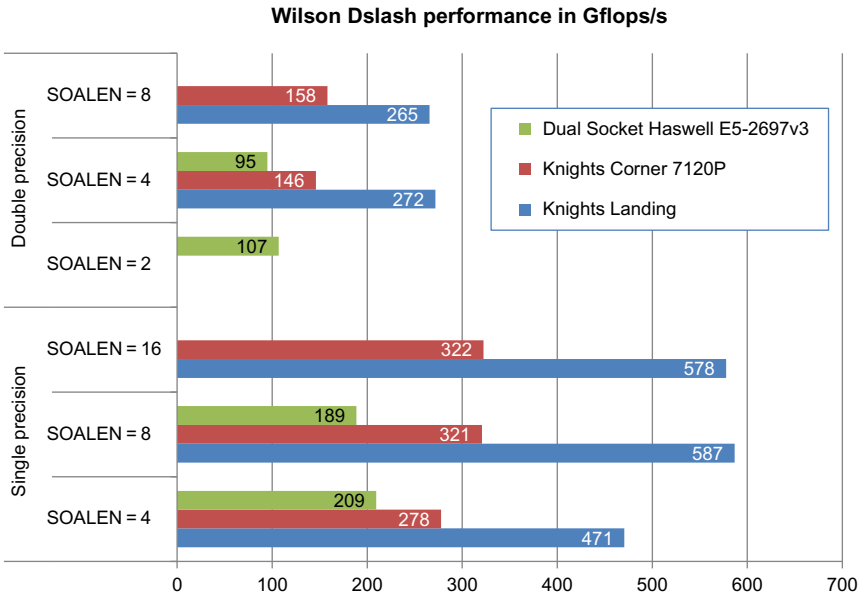


FIG. 26.10

Wison Dslash performance.

OVERALL PERFORMANCE ANALYSIS

The benefits of the Knights Landing processor node are clearly illustrated in Fig. 26.10 as it delivers nearly 3× the single-precision performance of the Dual Socket Haswell E5-2697v3 processor and roughly 1.8× that of a Knights Corner-based coprocessor. The minor difference in performance between the SOALEN 8 and 16 results can reasonably be attributed to runtime variations.

IMPACT OF MEMORY BANDWIDTH

Fig. 26.11 shows the scaling of our microbenchmark according to the number of cores when running out of either MCDRAM or DDR4 memory. Our analysis is simplified as our microbenchmark fits entirely in the 16 GB of MCDRAM on the Knights Landing node, which means we should be able to realize the full bandwidth of this memory subsystem.

As can be seen in Fig. 26.11, there is approximately a 3× performance difference at high core counts compared to the performance reported when running out of the DDR4 memory subsystem. The flat performance of the DDR4 memory starting at low core counts clearly shows that memory bandwidth limits the performance of the Knights Landing processor. Notice that we do not see the same flat line behavior with the MCDRAM memory subsystem. However, beyond 50 cores, scaling efficiency slowly drops due to increased L2 latencies. Overall, we get 27× speed up (79% efficiency) when scaling from 1 Tile (2 cores) to 34 Tiles (68 cores).

STREAMING BANDWIDTH ROOFLINE ESTIMATE

We then investigated the relative streaming memory performance of the MCDRAM and DDR4 memory system by running a specialize version to measure pure streaming read performance. We picked read bandwidth as opposed to standard Triad Bandwidth because Dslash operator is read heavy with 5:1 read-write ratio compared to 2:1 ratio for stream triad. Moreover, MCDRAM has separate channels for read and write making streaming writes virtually free. We measured the systems read bandwidth by timing a loop that simply sums all the elements of an array to a single value (Fig. 26.12).

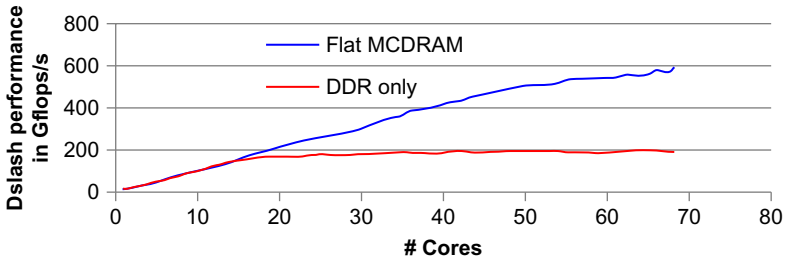


FIG. 26.11

Comparative Dslash performance between MCDRAM and DDR4.

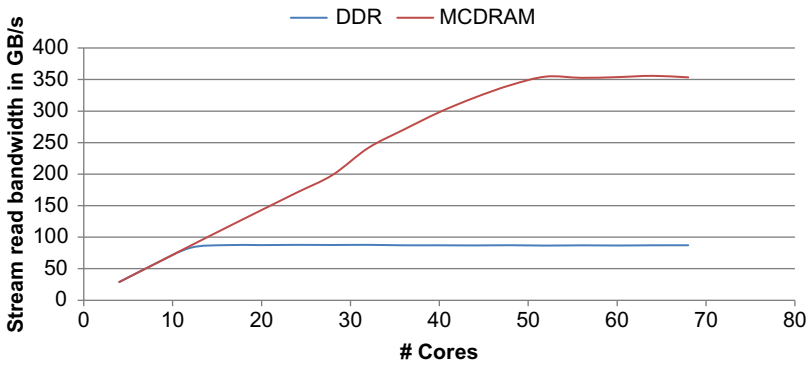


FIG. 26.12

DDR4 vs MCDRAM read bandwidth (roof line est. for our application).

PEAK ACHIEVABLE KNIGHTS LANDING PERFORMANCE

Finally, we investigated the peak achievable performance calculated based on achieved steam bandwidth and assuming ideal cache reuse. In other words, the maximum performance we would expect were we to exhaust the read-bandwidth of the system to a reasonable degree. For our purposes, this is our target when performing architecture-specific optimizations.

Fig. 26.13 compares current achieved best performance with expected MCDRAM bandwidth-bound performance and instruction-bound performance.

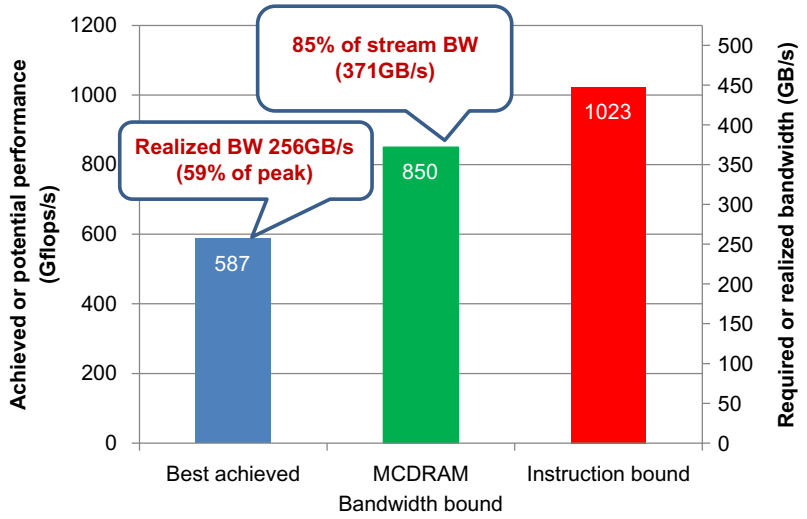


FIG. 26.13

Achieved or potential performance (so-called roof line estimates).

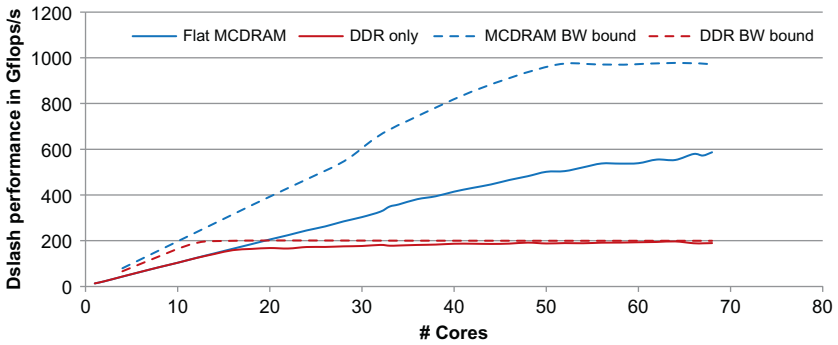


FIG. 26.14

Scaling by cores.

Instruction-bound performance was measured by running the innermost Dslash call 100 times with same data so that most of the data comes from L1 cache. This sets the upper bound on performance if we are not limited by memory bandwidth. The expected MCDRAM bandwidth-bound performance was calculated assuming that we can achieve 85% of STREAM bandwidth (based on our observation on Haswell and Knights Corner systems) and multiplying it by flops/byte ratio. For MCDRAM, we adjusted flops/byte ratio to account only for read traffic ($2.29 \times 6/5 = 2.75$) and then multiplied it by 85% of the STREAM read bandwidth.

Fig. 26.13 indicates that our microbenchmark is memory bandwidth limited rather than instruction bound.

In Fig. 26.14, we show scaling of the potential and achieved performance as a function of the number of cores used. The solid lines show performance achieved on the Knights Landing system when using the MCDRAM and DDR4 memory subsystems, whereas dotted lines show maximum achievable (potential) performance limited by stream bandwidth provided by underlying memory subsystem.

MEMORY BANDWIDTH ANALYSIS

We observe that the MCDRAM memory subsystem provides approximately $4\times$ the read bandwidth of the DDR4 memory subsystem. In addition to this, MCDRAM provides separate write bandwidth. Thus, compared to DDR we have close to $5\times$ more bandwidth when using MCDRAM. Therefore, we would expect $5\times$ more performance when running from MCDRAM as indicated by dotted lines in Fig. 26.14. However, with our Knights Landing optimized implementation of microbenchmark, we see only $3\times$ faster Dslash performance when using MCDRAM. Although this is a substantial gain realized due to higher MCDRAM bandwidth, there is still another 30–35% performance left that could be realized by Knights Landing-specific tunings and optimizations.

We then investigated how various modes of MCDRAM, thread per core, and software prefetching impact the performance of Dslash on the Knights Landing system.

MEMORY AND CLUSTER MODES

We measured the performance of Wilson-Dslash on Knights Landing system with three different memory modes (i.e., DDR only, using MCDRAM in Flat mode using numactl, and using MCDRAM as cache) and two different cluster modes (Quadrant and All-to-All). Fig. 26.15 shows the achieved performance in these two modes. Not surprisingly, the values between all-to-all and quadrant cluster modes are nearly identical—the differences are likely noise. The cache does quite well suggesting this code is generally cache friendly. It will be interesting to try additional configurations in the future.

ANALYSIS

We found that the best performance is achieved when using MCDRAM in flat mode, and the cluster mode is set to Quadrant (Quad). However, we do not see any significant impact on performance change when using all-to-all cluster mode. Similarly, performance remains within 2–3% of best achieved when using MCDRAM as cache. Finally, as discussed earlier, performance for DDR only configuration is severely limited by DDR bandwidth and has virtually no impact from cluster mode.

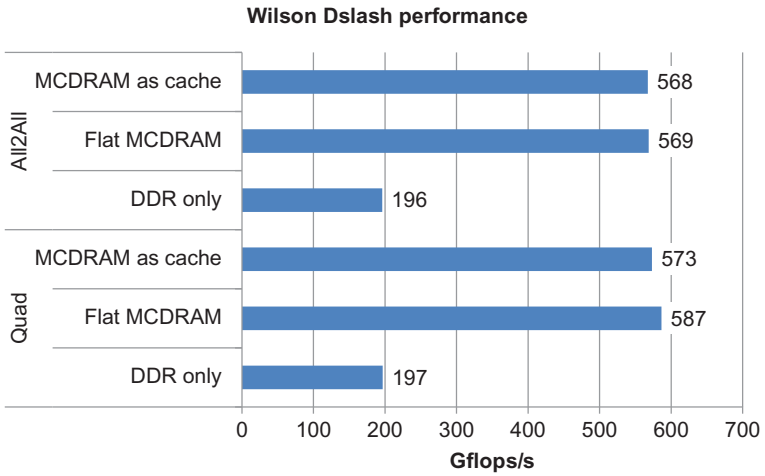


FIG. 26.15

Performance of Wilson-Dslash across memory (flat used numactl) and cluster modes.

THREADING

Fig. 26.16 demonstrates that Knights Landing can deliver single-thread per core performance within 80% of best achieved performance across a wide spectrum of core counts while two threads per core delivers almost same performance as four threads per core. In many cases, we observed that two threads per core delivers the best performance—even surpassing four threads per core.

Fig. 26.17 shows how performance varies according to the number of threads per core when using all available cores. The impact of the more powerful Knights Landing processing core compared to Knights Corner is readily apparent as single-threaded Knights Landing achieved nearly 80% of the highest observed Knights Landing performance. In contrast, single-threaded Knights Corner performance was only 50% of the best observed multithreaded Knights Corner performance.

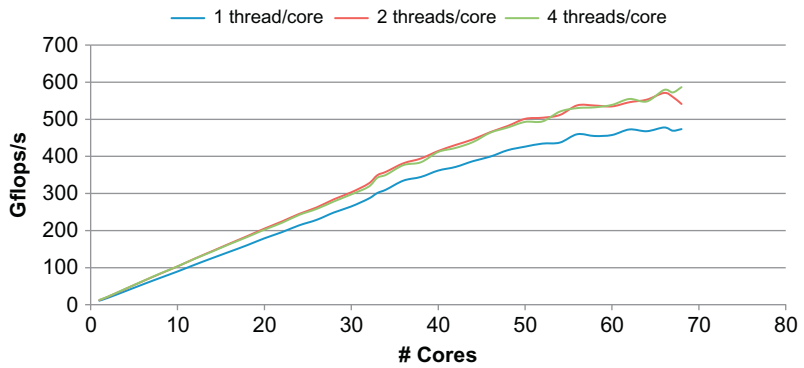


FIG. 26.16

Knights Landing performance as a function of number of cores.

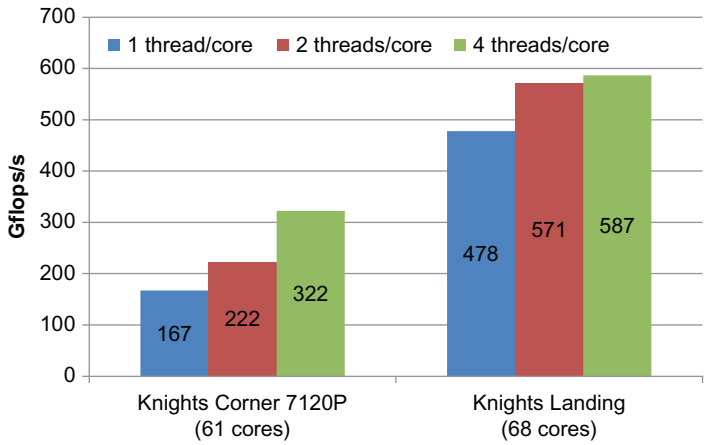


FIG. 26.17

Threads per core on Knights Corner and Knights Landing.

THREADING (OR THREADS PER CORE) ANALYSIS

While it is clear that utilizing multiple threads per core is important on both Knights Corner and Knights Landing devices, the observed 3% variation between two and four thread per core indicates that tuning the number of threads per core is not as important on Knights Landing as it was on the previous generation Knights Corner-based devices.

PREFETCHING

Fig. 26.18 shows the ratio of performance observed with software prefetching relative to performance observed without prefetching. A value greater than 1.0 indicates a performance gain from software prefetching, while a value less than 1.0 indicates that software prefetching is detrimental to performance.

Fig. 26.19 shows that software prefetching helped deliver as much as 50% extra performance on the Knights Corner-based coprocessor. In contrast, same strategy for software prefetching on Knights Landing did not buy any extra performance. Instead, we observe a performance loss, of up to 5%, presumably due to instruction overheads. However, Knights Landing-specific fine tuning may help improve performance as we observe that we are still about 30% off from expected MCDRAM bandwidth-bound performance.

SOFTWARE PREFETCH ANALYSIS

Our tests found that Knights Corner-specific software prefetching strategy buys about 5% extra performance on Knights Landing at lower core count but benefits effectively disappear when using all the cores. Overall, we found the best performance is achieved on Knights Landing when no software prefetches were used.

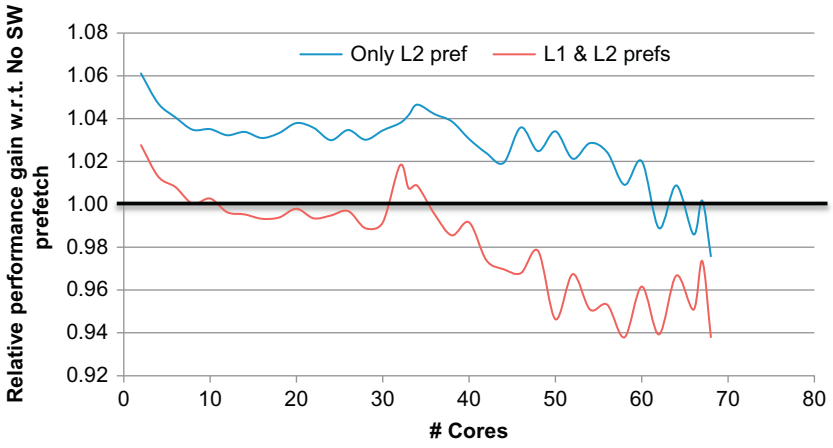


FIG. 26.18

Relative software prefetching performance on Knights Landing.

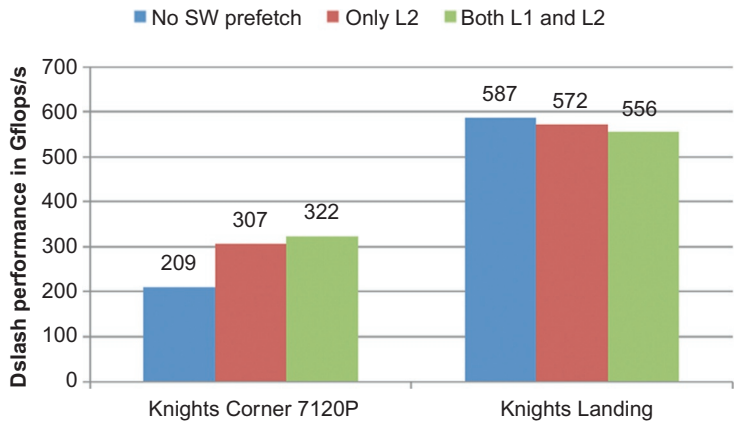


FIG. 26.19

Dslash performance on Knights Corner and Knights Landing as a function of prefetching.

CONCLUSION

We note our kernel runs approximately 3× faster when running in MCDRAM as opposed to the DDR4 memory subsystem on the Knights Landing node as shown in Fig. 26.11. We primarily attribute the increased performance to the higher MCDRAM memory bandwidth as our kernel is expected to be bandwidth bound due to its low flops to byte ratio.

Our expectation is borne out in practice, as changing the number of threads per core and other latency hiding operations does not significantly increase performance. However, the Knights Landing architecture does provide an advantage as performance is much less sensitive to the number of threads per core than on the earlier Knights Corner coprocessors. Further, we note that software prefetching is not very critical for realizing performance benefits on Knights Landing hardware.

Overall, we observe there is about 3× advantage to running on Knights Landing-based nodes over a dual socket Haswell E5-2697v3-based system.

FOR MORE INFORMATION

- More detail background on this application, with optimization work for Knights Corner discussed: Chapter 9—*Wilson-Dslash Kernel From Lattice QCD Optimization*, by Balint Joo, Mikhail Smelyanskiy, Dhiraj D. Kalamkar, and Karthikeyan Vaidyanathan, in *High-Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, 2015, ISBN 978-0-12-803819-2.

- The QPhiX library and code generator are available on GitHub at https://github.com/JeffersonLab/qphix and https://github.com/JeffersonLab/qphix-codegen, respectively.
- http://lotsofcores.com has code specific to our examples used in this book chapter.

Contributors

Jefferson Amstutz, Intel

Jeff is a visualization software engineer where he works on the open source OSPRay Project. He enjoys all things ray tracing, high performance computing, clearly implemented code, and the perfect combination of Git/CMake/modern C++. When he is able, he enjoys academic research in ray tracing and high performance computing, with a specific interest in multi-hit ray tracing algorithms and applications for both graphics 3D rendering and ray-based simulations.

James A. Ang, Sandia National Laboratories

Jim directly supports the U.S. Department of Energy's Exascale Computing Project (ECP). His primary responsibility is to define and execute the ECP research and development strategy for node and system architecture designs—including processor designs, advanced memory and storage subsystems, interconnection networks, system resilience, power management, and application/architecture performance analysis.

Ryo Asai, Colfax International

Ryo is a researcher developing optimization methods for scientific applications targeting emerging parallel computing platforms, computing accelerators, and interconnect technologies.

Igor Astakhov, Intel

Igor is a software architect for the Intel Performance Primitives Libraries (IPP). His areas of interests include signal and image processing, and computer vision, algorithms, and their optimization for the latest and future Intel architectures.

Michael Bader, Technische Universität München (TUM)

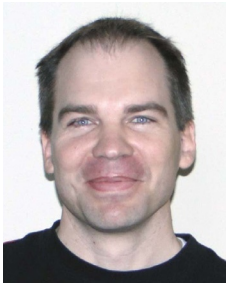
Michael is an associate professor at the Department of Informatics. He works on hardware aware algorithms and software for high performance computing. His research interests include parallel adaptive mesh refinement, optimizing PDE software for novel supercomputing architectures, and—*on the applications side*—tsunami and earthquake simulation.

Heinrich Bockhorst, Intel

Heinrich is a senior technical consultant for HPC in the Software and Services Group. He supports several Intel Parallel Computing Centers in their code modernization efforts. His primary interests include MPI/hybrid applications and related analyzing tools.

Alexander Breuer, San Diego Supercomputer Center

Alex is a postdoctoral researcher at the San Diego Supercomputer Center of the University of California, San Diego. His research interests cover high performance simulations of dynamic rupture and seismic wave propagation.

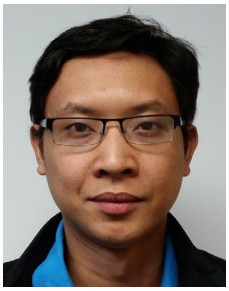


Michael Brown, Intel

Mike is a software architect/engineer focusing on HPC Life Sciences. His researches how to advance capabilities for studying important problems in the life sciences using HPC.

Ilya Burylov, Intel

Ilya is a senior software engineer in the Numerics team at Intel Corporation. His background is in computation optimizations for statistical, financial, and transcendental math function algorithms. He focuses on optimization of computationally intensive analytics algorithms and data manipulation steps for Big Data workflows within distributed systems.



Jan-Michael Y. Carrillo, Oak Ridge National Laboratory

Jan-Michael focuses on the use of computational methods to investigate polymers. He performs simulations that provide insights on the interpretation of experiments. His simulations inspire novel characterization experiments and influence the direction of syntheses of polymeric and colloidal materials.

Bruce Cherniak, Intel

Bruce is a graphics software engineer and currently develops the OpenSWR scalable graphics renderer. He has developed for a variety of Intel graphics solutions, including the OpenGL driver and rendering pipeline for the Larrabee Project. His broad graphics experience spans from gaming to professional CAD/DCC.

Sundaram Chinthamani, Intel

Sundaram is a senior staff performance architect. He leads work on Knights Landing performance tuning. His research interests are in high performance computing, uncore architecture and microarchitecture, memory controllers, analytical, and simulation based performance modeling. He has worked on developing detailed cycle-accurate performance simulators, performance projections, and pre-silicon performance validation of many Intel Xeon server processors. He has six patents in the field of computer architecture.

Dipankar Das, Intel

Dipankar is a research scientist with the Parallel Computing Lab where he works on the design of efficient parallel applications, workload driven architecture design, and performance productivity problems. His research focus is on design and analysis of parallel applications and systems.

Mark Debbage, Intel

Mark is a principal engineer in the Enterprise and HPC Platform Group. He is the lead architect of the Host Fabric Interface (HFI) for the 100 series of Omni-Path Architecture (OPA) products.

Bob De Kraker, Shell Global Solutions

Bob is a research engineer working on the structure-property relationships of hydrocarbons.

James Dinan, Intel

Jim is a software architect in Intel's Extreme Scale Software Pathfinding team. He also serves on the MPI Forum and OpenSHMEM specification committee. His research interests include parallel programming models, high-performance fabrics, scalable runtime systems, distributed algorithms, scientific computing, and computer architecture.

Pradeep Dubey, Intel

Pradeep is an Intel Fellow, the director of the Parallel Computing Lab, and an IEEE Fellow. He has made significant contributions to the design, architecture, and application performance of various microprocessors, including the IBM Power PC, the Intel386, Intel486, Intel Pentium, Intel Xeon processors, and the Intel Xeon Phi product line. He holds more than 30 patents and has published more than 50 peer-reviewed technical papers.

Rob Farber, Techenablement.Com

Rob is a consultant with an extensive background in HPC and a long history of working with national laboratories and corporations engaged in HPC optimization work. He has authored/edited several books on GPU programming and is the CEO/publisher of TechEnablement.com.

Nitin Gavhane, Shell India Markets Private Limited

Nitin is a computational software specialist. He has experience in performance engineering of HPC applications. His work also involves preparing Legacy applications for Exascale computing.

Indraneil M. Gokhale, Intel

Indraneil is a software architect at Intel with emphasis on pre-silicon, and post-silicon performance analysis, and optimization for Knights Landing. His work focuses on the optimization of scientific applications and machine learning algorithms.

Roger Gramunt, Intel

Roger is a senior computer architect and one of the primary architects of the Knights Landing core. His research interests include core architecture, graphics architecture, binary translation, and vector instruction sets.

Jonathan C. Hall, Intel

Jon is an architect on the Intel Xeon Phi products including Knights Landing where he specialized in the VPU, OOO, and Exec portions of Knights Landing. He is an optimization expert who advises compiler teams and application engineers on tuning techniques. Jon's prior projects include Nehalem and Pentium 4 processors.

Jeff R. Hammond, Intel

Jeff is a research scientist working on massively parallel scientific applications and programming models for high-performance computing. He contributes to the development of open standards for parallel programming (MPI, OpenSHMEM, and OpenMP) and numerous open-source software projects, especially NWChem. His interests include the efficacy of parallel programming models using the Parallel Research Kernels (https://github.com/ParRes/Kernels).

Alexander Heinecke, Intel

Alex is a research scientist in the Parallel Computing Lab. His research focus lays on the use of multi- and many-core architectures in advanced scientific computing applications and extreme scale computing.

James L. Jeffers, Intel

Jim is a principal engineer and engineering manager in the HPC Platform Group. He has over 25 years software design and technical leadership experience for high performance computing, visual computing, digital television, and data communications. His prior work includes development of the virtual image insertion television technology behind American football's "Electronic First Down Line." He has coauthored/edited three other books with James Reinders on Intel Xeon Phi programming. He currently leads the HPC Visualization team.

Ashish Jha, Intel

Ashish is a performance architect in the Software and Services Group. He worked on AVX2 and AVX-512 path finding to drive HPC requirements, and compiler support, for exploiting these instructions. His extensive expertise in performance analysis, pre-silicon and post-silicon, includes work on Atom, Xeon, and Xeon Phi processors driving improvements in their architectures. He has 20 patents filed.

Dhiraj D. Kalamkar, Intel

Dhiraj is a research engineer in Parallel Computing Lab. His research interests include parallel computer architecture, GPU architectures, hardware specific power and performance optimizations, and hardware enabled platform security. He is currently working on analyzing and optimizing various workloads for Intel CPU, Gen, and MIC (Intel Xeon Phi) architectures.

Robert J. Kyanko, Intel

Rob is a principal engineer specializing in software support for new memory systems. He has an extensive background in operating systems, device drivers, 3D graphics, CPU microarchitectural studies, embedded programming, network programming, simulation, and scientific programming.

Thomas D. Lovett, Intel

Tom is a senior principal engineer in the Enterprise and HPC Platform Group at Intel Corporation, and manages the Core Architecture Group within engineering in the Enterprise High Performance Computing Group.

Andrew Mallinson, Intel

Andy is an application engineer focused on enabling developers to fully utilize Knights Landing and other platforms. His research interests include Partitioned Global Address Space (PGAS) programming paradigms in particular their use within irregular and data intensive applications.

Zakhar A. Matveev, Intel

Zakhar is Intel Parallel Studio software architect. He is currently focused on product design for tools to help with efficient x86 vector SIMD parallelism, effective memory sub-systems utilization, and more broadly HPC codes modernization. His professional interests include HPC systems optimization, parallel programming, computer graphics, customer-oriented software design, and usability.

Lawrence F. Meadows, Intel

Larry is a principal engineer focused on optimization of scientific applications to fully utilize Knights Landing. He has worked on compilers, tools, and applications software for HPC since 1982.

John Michalakes, University Corporation for Atmospheric Research

John is a scientific programmer/analyst and a visiting scientist in the Naval Research Laboratory's Marine Meteorology Division. He was the lead software developer for the Weather Research and Forecast (WRF) model software at the National Center for Atmospheric Research.

Biswajit Mishra, Shell India Markets Private Limited

Biswajit works as a computational software specialist at Shell and his interest is in optimization and parallelization of scientific software on many-core HPC platforms.

Andrey Nikolaev, Intel

Andrey is a software architect in the Software and Services Group. He is the architect for the Intel® Data Analytics Acceleration Library (DAAL) and architect of the statistical/data processing capabilities in Intel® Math Kernel Library (MKL). Andrey helped enhance security capabilities of several generations of Intel platforms by designing relevant stochastic models and algorithms. He has contributed to the specifications of the financial industry benchmark STAC A2.

Mostofa Ali Patwary, Intel

Mostofa is a research scientist in the Parallel Computing Lab. His research interests span the areas of big data analytics, high performance computing, combinatorial scientific computing, and parallel algorithms. He is currently working on optimizing and parallelizing machine learning kernels for Intel Xeon and Intel Xeon Phi architectures.

Ramesh V. Peri, Intel

Ramesh is a senior principal engineer, specializing in performance optimization across all platforms. He is the architect for IoTDevkit and development tools for mobile platforms.

Vadim O. Pirogov, Intel

Vadim is a software engineering manager in the Software and Services Group at Intel Corporation. He leads several teams development teams for the Intel MKL library.

Steve Plimpton, Sandia National Laboratories

Steve is a staff member in the Multiscale Sciences Department at Sandia and the lead developer for the LAMMPS molecular dynamics package. He works with scientific simulations and algorithms designed for large parallel machines. See http://www.sandia.gov/~sjplimp for other open-source simulation projects he's involved with.

Karthik Raman, Intel

Karthik is a software architect at Intel focusing on performance analysis and optimization of HPC workloads for Knights Landing. He focuses on analyzing for optimal compiler code generation, vectorization, and assessing key architectural features for performance. He helps deliver transformative methods and tools to expose new opportunities and insights.

James R. Reinders, Intel

James helps advance parallel programming. His projects have included several world's firsts: the first Tflop/s supercomputer (ASCI Red), the first Tflop/s microprocessor (Knights Corner), the first 3 Tflop/s computer (ASCI Red upgrade), and the first 3 Tflop/s microprocessor (Knights Landing). This is his eighth programming book. He has coauthored/edited three other books with Jim Jeffers on Intel Xeon Phi programming. His work focuses on parallel programming models, especially issues surrounding performance portability.

Todd Rimmer, Intel

Todd is a principal engineer and the lead software architect for the Intel Omni-Path Fabric. He has over 15 years of experience with HPC, RDMA, and InfiniBand with an overall career focus on high performance IO and servers. Todd has 28 patents issued or pending.

Timothy O. Rowley, Intel

Tim is a graphics software engineer. He worked on the Larrabee graphics architecture in the areas of the OpenGL driver, rendering pipeline, and shader compiler. Since Larrabee, he contributed to a variety of Intel graphics solutions and currently develops the OpenSWR scalable software graphics rasterizer. Past employment includes PowerVR and project lead for Mozilla Scalable Vector Graphics (SVG).

Nadathur Rajagopalan Satish, Intel

Satish is a research scientist in the Parallel Computing Lab. His research interests generally relate to parallel applications and architectures. His recent work includes mapping emerging applications on parallel computer architectures, tools, and languages to aid parallel workload development, and next generation computer architectures.

Mikhail Smelyanskiy, Intel

Mikhail is a principal engineer in the Parallel Computing Lab. He focuses on design, implementation and analysis of parallel algorithms and workloads for current and future generation parallel processor systems. His research interests include medical imaging, computational finance and fundamental high performance compute kernels.

Avinash Sodani, Intel

Avinash is a senior principal engineer, and the chief architect of Knights Landing responsible for its overall architecture. He led a team of architects who took Knights Landing from initial definition, through implementation, and to final product. Previously, he was an architect on the Nehalem and Westmere processors, and did early work on the Haswell processor. Avinash specializes in high performance computing (HPC) with deep technical expertise in the core microarchitecture for multi-/many-core processors.

Foram Thakkar, Shell India Markets Private Limited

Foram is a researcher in Computational Center of Expertise at Shell. His work involves using various molecular simulation techniques for the systems and properties of interest.

Karthikeyan Vaidyanathan, Intel

Karthik is a research scientist in the Parallel Computing Labs. His research interests include high-performance computing, high-speed interconnects, storage, and performance optimizations in computer architecture. He is currently working on analyzing and optimizing communication performance of HPC and machine learning workloads for Intel Xeon and Intel Xeon Phi architectures.

Antonio C. Valles, Intel

Antonio is a senior software engineer focused on performance analysis and optimizations. Antonio has analyzed and optimized software at Intel spanning client, mobile, and HPC segments. He loves to code and has written multiple internal post-silicon and pre-silicon tools to help analyze and optimize applications.

Andrey Vladimirov, Colfax International

Andrey is the head of HPC Research at Colfax International. His research is focused on the application of modern computing technologies to computationally demanding scientific problems.

Ingo Wald, Intel

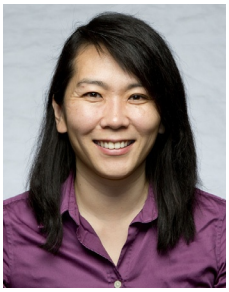
Ingo is a research scientist at Intel. His research interests revolve around all aspects of ray tracing and lighting simulation, real-time graphics, parallel computing, and general visual/high-performance computing.

Alex M. Wells, Intel

Alex is a senior application engineer who leads enabling data-parallelism in DreamWorks Animation tools, specifically character animation. He architected the Intel SIMD Data Layout Templates (SDLT) as well as the open source project, XForm Building Blocks, a C++ library to enable composition of 3D transforms.

Claude Wright, Intel

Claude is an HPC engineer with expertise on Xeon Phi power analysis. He was also involved with the number one Green 500 listing with the launch of Knights Corner, and is currently working on the CORAL Aurora Supercomputer bring-up and validation.



Masako Yamada, GE Global Research

Masako leads the Secure Architectures and Scalable Systems Lab at GE Global Research. She has been awarded four DOE supercomputing grants totaling 160 million CPU-hours and was selected a "2014 HPCWire People to Watch."

Chuck Yount, Intel

Chuck is a principal engineer in the Software and Services Group. His research interests are computer performance analysis and optimization (pre-silicon and post-silicon), object-oriented software design, machine learning, and computer architecture.

Robert C. Zak, Intel

Bob is a senior principal engineer at Intel. His current work focuses on the development of large scale high performance fabrics for distributed applications.

Glossary

µarch See microarchitecture.

µop See micro-op.

A0 stepping Stepping numbers are essentially version numbers. A0 was the first realization of the Knights Landing. The A0 stepping of Knights Landing was not available for sale. A0 was followed by a B0 stepping. Fig. 2.5 shares some A0 results, and all other performance data in this book uses preproduction B0 parts.

ACPI See Advanced Configuration and Power Interface Specification.

Advanced Configuration and Power Interface Specification (ACPI) Well-defined interfaces to power management and configuration interfaces.

Advanced Vector Extensions SIMD instructions originally in 256-bit versions (AVX, AVX2) and now in 512-bit versions with Knights Landing and future processors, see Chapters 9–12 and others (see Index).

Advisor An Intel analysis tool for determining potential benefits from various approaches to adding parallelism without having to write, debug, and test code in order to study the benefits, see Chapter 10.

Affinity The specification of methods to associate a particular software thread to a particular hardware thread usually with the objective of getting better or more predictable performance. Affinity specifications include the concept of being maximally spread apart to reduce contention (scatter), or to pack tightly (compact) to minimize distances for communication. OpenMP supports a rich set of affinity controls at various levels from abstract to full manual control. Fortran 2008 does not specify controls, but Intel reuses the OpenMP controls for "do concurrent." Intel Threading Building Blocks (TBB) provides an abstract loop-to-loop affinity biasing capability.

Aliasing When two distinct program identifiers or expressions refer to overlapping memory locations. For example, if two pointers p and q point to the same location, then $p[x]$ and $q[x]$ are said to alias each other. The potential for aliasing can severely restrict a compiler's ability to optimize a program, even when there is no actual aliasing.

All-to-all cluster mode One of the cluster modes. Typically used when DDR4 DIMMS are not all populated with identical capacity devices, see Chapters 3 and 4.

Allocation Unit (AU) A Knights Landing microarchitecture unit which is responsible for preparing µops for out-of-order execution, see Chapter 4.

Amdahl's law Speedup is limited by the nonparallelizable serial portion of the work. A program where two-thirds of the program can be run in parallel and one third of the original nonparallel program cannot be sped up by parallelism will find that speedup can only approach $3\times$ and never exceed it assuming the same work is done. If scaling the problem size places more demands on the parallel portions of the program, then Amdahl's law is not as bad as it may seem, see **Gustafson-Barsis' law**.

AMG One of the eight Trinity applications, see Chapter 25.

Amplifier See **VTune**.

AoS See Array of Structures.

AoSoA See Arrays of Structures of Arrays.

API See Application Programming Interface.

Application programming interface (API) An interface (set of function calls, operators, variables, and/or classes) used by an application developer to use a module. The implementation details of a module are ideally hidden from the application developer and the functionality is only defined through the API.

Array of Structures (AoS) A data organization which can hurt parallel application performance versus SoA but often feels more object oriented and intuitive to write, see Chapter 11 and others (see Index).

Arrays of Structures of Arrays (ASA, AoSoA) A variation of SoA data organization which can be more cache friendly and outperform AoS and SoA

ASA See Arrays of Structures of Arrays.

Atomic operation An operation that is guaranteed to appear as if it occurred indivisibly without interference from other threads. For example, a processor might provide a memory increment operation. This operation needs to read a value from memory, increment it, and write it back to memory. An atomic increment guarantees that the final memory value is the same as would have occurred if no other operations on that memory location were allowed to happen between the read and the write.

AU See Allocation Unit (AU).

autohbw An interposer library that redirects select memory allocations to use MCDRAM without regarding any changes to an application, part of **memkind** project, see Chapter 3.

Automatic offload (AO) A generic library feature that automatically redirects some computation to use a specialty device for data parallelism, such as a coprocessor (MIC). AO is supported by the Intel® Math Kernel Library (MKL). When desired, AO can also be controlled more finely with more complex parameters through additional options within MKL. A key concept is that offloading will occur when it will benefit the computation. The other key concept in AO

is that the computation will be performed by host processor(s) if offloading is not available. Put another way, if you ignore all AO-related extensions in the program, it will do the same computation but without use of the offload target, see Chapter 18.

AVX-512 See Advanced Vector Extensions.

AVX-512 intrinsics A way to program to use AVX-512, see Chapter 12.

AVX2 See Advanced Vector Extensions.

AVX See Advanced Vector Extensions.

B0 stepping Stepping numbers are essentially version numbers. A0 was the first realization of the Knights Landing. The A0 stepping of Knights Landing was not available for sale. A0 was followed by a B0 stepping. Aside from Fig. 2.5, all data in this book used preproduction B0 versions of Knights Landing.

Bandwidth The rate at which information is transferred, either from memory or over a communications channel. This term is used when the process being measured can be given a frequency-domain interpretation. When applied to computation, it can be seen as being equivalent to throughput.

Barrier When a computation is broken into phases, it is often necessary to ensure that all threads complete all the work in one phase before any thread moves onto another phase. A barrier is a form of synchronization that ensures this: threads arriving at a barrier wait there until the last thread arrives, then all threads continue. A barrier can be implemented using atomic operations. For example, all threads might try to increment a shared variable and then block if the value of that variable does not equal the number of threads that need to synchronize at the barrier. The last thread to arrive can then reset the barrier to zero and release all the blocked threads.

Basic Linear Algebra Subprograms See BLAS.

Bitwise copyable Characteristic of a data structure that allows a simple bit-by-bit copy (sometimes called a "shallow" copy) operation to work properly. This term is used in the C++ standard. A bitwise copyable data structure will not contain pointers and does not invoke constructors or destructors.

BIU See Bus Interface Unit.

BLAS The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for basic vector and matrix operations. The Level 1 BLAS perform scalar, vector, and vector-vector operations, the Level 2 BLAS perform matrix-vector operations, and the Level 3 BLAS perform matrix-matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high-quality linear algebra software, LAPACK for example. A sophisticated and generic implementation of BLAS has been maintained for decades at http://netlib.org/blas. Vendor-specific implementations of BLAS are common, including the Intel Math Kernel Library (MKL) that is a highly efficient version of BLAS and other standard routines for Intel architecture.

Block Used in two senses: (1) a state in which a thread is unable to proceed while it waits for some synchronization event and (2) a region of memory. The second meaning is also used in the sense of dividing a loop into a set of parallel tasks of a suitable granularity. To avoid confusion in this book, the term *tiling* is preferred over *blocking*.

Bus Interface Unit (BIU) A Knights Landing microarchitecture unit that controls the L2 cache and coherency between the two cores within the tile, see Chapter 4.

C++ Composer The Intel C/C++ compiler plus libraries.

C-state Core idle state, a power savings capability on processors and coprocessors with a tradeoff being lower power but higher latency to do the next real work when available. Deeper sleep states offer lower power but take longer to revive to full performance.

Cache A part of memory system that stores copies of data temporarily in a fast memory so that future uses of that data can be handled more quickly than if the request had to be fetched again from a more distant storage. Caches are generally automatic and are designed to enhance programs with temporal locality and/or spatial locality. Caching systems in modern computers are usually multileveled.

Cache friendly A characteristic of an application in which performance increases as problem size increases but then levels off as the bandwidth limit is reached. Such workloads are able to use MCDRAM effectively even at larger problem sizes.

Cache lines The units in which data are retrieved and held by a cache, which in order to exploit spatial locality are generally larger than a word. The cache line sizes are generally large enough to hold up to eight double-precision floating-point numbers, on any current design. Larger cache lines allow for more efficient bulk transfers from main memory but worsen certain issues including false sharing, which generally degrades performance.

Cache memory mode A memory mode in which all of MCDRAM acts as a cache, see Chapters 3 and 4.

Cache unfriendly A characteristic of an application in which the memory footprint of the workloads needs to be optimal, and caches are not useful or even detrimental. In this case, we see performance stays constant or increase as problem size reaches the optimal size (e.g., MCDRAM size, less than MCDRAM size or slightly larger than MCDRAM size) and then performance decreases as problem size increases.

Caching/Home Agent (CHA) A Knights Landing microarchitecture unit that is responsible for maintaining tag directories that keep the (distributed) L2 cache coherent. It also has hardware interface through which the tiles connect to the 2D Mesh on-die interconnect, see Chapter 4.

CHA See Caching/Home Agent (CHA).

Clusters A set of computers with distributed memory communicating over a high-speed interconnect. The individual computers are often called **nodes**.

Coarrays (Fortran) A global distributed array feature of Fortran 2008, see Chapter 16.

Code modernization The act of designing and optimizing applications to utilize parallelism, #CodeModern.

COI See Coprocessor Offload Infrastructure.

Communication Any exchange of data or **synchronization** between software tasks or threads. Understanding that communication costs are often a limiting factor in scaling is a critical concept for parallel programming.

Compiler-assisted offload OpenMP target directives, see Chapter 18.

Composability The ability to use two components in concert with each other without causing failure or unreasonable conflict (ideally no conflict). Limitations on composability, if they exist, are best when completely diagnosed at build time instead of requiring any testing. Composability problems that manifest only at runtime are the biggest problem with noncomposable systems. Can refer to system features, programming models, or software components.

Composer Intel compilers plus libraries.

Concurrent Things are logically happening simultaneously. Two tasks that are both logically active at some point in time are considered to be concurrent. Contrast with **parallel**.

Convolutional Neural Networks (CNN) A machine learn technique, see Chapter 24.

Coprocessor A separate processor, often on an add-in card (such as a PCIe card), usually with its own physical memory, which may or may not be in a separate address space from the host processor. Often also known as an accelerator (although it may only accelerate specific workloads). In the case of Intel Xeon Phi coprocessors, a coprocessor is a computing device that cannot be the only computing device in a system design. In other words, a computing device that also requires a processor in a system design.

Coprocessor Offload Infrastructure (COI) A middleware layer written by Intel with an API that supports the asynchronous delivery and management of code and data buffers between an Intel Xeon® host processor and an Intel Xeon Phi coprocessor(s). COI has been extended for Knights Landing to support offloading to Knights Landing processors using the fabric to transmit code and data.

Core A separate subprocessor on a multicore processor. A core should be able to support (at least one) separate and divergent flow of control from other cores on the same processor. Note: there is some inconsistency in the use of this term. For example, some graphic processor vendors use the term as well for SIMD lanes supporting fibers. However, the separate flows of control in fibers are simulated with masking on these devices, so there is a performance penalty for divergence. We will restrict the use of the term **core** to the case where control flow divergence can be done without penalty.

CPI Clocks-per-instruction, a common measure to gauge how well optimized an application is, see Chapters 14, 22, and others (see Index).

CUDA A parallel programming model created by NVidia for NVidia GPUs.

DAAL See Data Analytics Library, see Chapter 13.

Data Analytics Library (DAAL) An Intel Performance Library, see Chapter 13.

Data parallelism An approach to parallelism that attempts to be oriented around data rather than tasks. However, in reality, successful strategies in parallel algorithm development tend to focus on exploiting the parallelism in data, because data decomposition (generating tasks for different units of data) scales, but functional decomposition (generation of heterogeneous tasks for different functions) does not. See Amdahl's law and Gustafson-Barsis' law.

Dcache Cache for data (not instructions); unified caches handle instructions and data both.

DDR See Double Data Rate memory.

DDR4 Fourth-generation Double Data Rate memory, the type of main memory that may be used by Knights Landing processors.

Deadlock A programming error. Deadlock occurs when at least two tasks wait for each other and each will not resume until the other task proceeds. This happens easily when code requires the locking of multiple mutexes. For example, each task can be holding a mutex required by the other task.

Deterministic A deterministic algorithm is an algorithm that behaves predictably. Given a particular input, a deterministic algorithm will always produce the same output. The definition of what is the "same" may be important due to limited precision in mathematical operations and the likelihood that optimizations including parallelization will rearrange the order of operations. These are often referred to as "rounding" differences, which result when the order of mathematical operations to compute answer differ between the original program and the final concurrent program. Concurrency is not the only factor that can lead to **nondeterministic** algorithms but in practice it is often the cause. Use of programming models with sequential semantics and eliminating data races with proper access controls will generally eliminate the major effects of concurrency other than the "rounding" differences.

DIMM See Dual In-line Memory Module.

Directive A language construct (#pragma in C/C++, C$DIR in Fortran) that specifies how a compiler should process its input.

Distributed memory Memory that is physically located in separate computers. An indirect interface, such as message passing, is required to access memory on remote computers, while local memory can be accessed directly. Distributed memory is typically supported by clusters which, for purposes of this definition, we are considering to be a collection of computers. Since the memory on attached coprocessors also cannot typically be addressed directly from the host, it can be considered, for functional purposes, to be a form of distributed memory.

Double Data Rate memory (DDR) A type of synchronous dynamic random access memory (SDRAM), DDR4 can be used by Knights Landing.

DTLB A TLB (Translation Lookaside Buffer) for data (not instructions); unified TLBs handle instructions and data both.

Dual In-line Memory Module (DIMM) A standard form factor for memory. It is a module containing one or several random access memory (RAM) chips on a small circuit board with pins that connect it to the computer motherboard.

ECC Error Correction Code, a method to increase reliability by correcting transient errors on a device. Used extensively on Intel Xeon processors and Intel Xeon Phi processors and coprocessors to offer high degrees of reliability.

emacs The best text editor in the world (according to James), and it is open source. Superior to the vi editor. "emacs" was the first package James installed on the first Knights Landing system he used.

Embarrassing parallelism A description of an algorithm if it can be decomposed into a large number of independent tasks with little or no synchronization required between tasks.

EMON Event monitoring: counting of events such as cache misses on a processor or coprocessor, see Chapter 13.

ExaFLOP 10^{18} Floating-Point Operations.

Exascale A machine capable of 10^{18} floating-point operations per second.

False sharing Two separate tasks in two separate cores may write to separate locations in memory, but if those memory locations happened to be allocated in the same **cache line**, the cache coherence hardware will attempt to keep the cache lines coherent, resulting in extra interprocessor communication and reduced performance, even though the tasks are not actually sharing data.

Far memory A NUMA system, memory that has longer access times than the near memory. The view of which parts of memory are near versus far depends on the process from which code is running.

FASTMEM A feature of Intel Fortran to label certain memory allocations to be from MCDRAM, see Chapter 3.

Flat Memory Mode A memory mode in which all of MCDRAM is memory, and none of it acts as cache, see Chapters 3 and 4.

Floating-point number A format for numbers in computers characterized by trading a higher range for the numbers for a reduced precision by using the bits available for a number and a shift count (exponent) that places the point to the left or right of a fixed position. In contrast, fixed-point representations use all the bits available for the number such as an integer number is usually represented in a computer.

Floating-point operations An operation on a floating-point number, for example, add, multiply, substract.

FLOP/s Floating-Point Operations per second.

FLOPs Floating-Point Operations.

FMA (Fused Multiply and Add) A capability to request a multiply and an add operation in one instruction while retaining higher precision than the two operations separately. It also potentially doubles the computational throughput of a device.

Fortran A programming language used primarily for scientific and engineering problem solving. Originally spelled FORTRAN as an abbreviation for FORmula TRANSlation.

Fortran Composer Intel Fortran Compiler plus libraries.

Forward scaling The concept of having a program or algorithm scalable already in threads and/or vectors so as to be ready to take advantage of the growth of parallelism in future hardware with a simple recompile with a new compiler or relink to a new library. Using the right abstractions to express parallelism is normally a key to enabling forward scaling when writing a parallel program.

Front-End Unit (FEU) A Knights Landing microarchitecture unit, is responsible for fetching instructions, decoding them, predicting branches, breaking instructions into simpler micro-operations (μops) and delivering them to the Allocation Unit, see Chapter 4.

future-proofed A computer program written in a manner, so it will survive future computer architecture changes without requiring significant changes to the program itself. Generally, the more abstract a programming method is, the more **future-proof** that program is. Lower-level programming methods that in some way mirror computer architectural details will be less able to survive the future without change. Writing in an abstract, more **future-proof** fashion may involve tradeoffs in efficiency, however.

Gather Gather-scatter is a type of memory access pattern that often arises when addressing vectors in sparse linear algebra operations. It also can occur for programs with AoS data layouts. An AVX-512 gather can read 8 64b-wide or 16 32b-wide noncontiguous memory locations and pack them into a 512b vector register. An AVX-512 scatter can do the opposite, it writes the 8 64b-wide or 16 32b-wide values from 512b vector register and write them into 8 or 16 noncontiguous memory locations, respectively.

GFLOP/s (GigaFLOP/s) 10^9 Floating-Point Operations per second.

GFLOPs (GigaFLOPs) 10^9 Floating-Point Operations.

GPU Graphics Processing Unit.

GTC One of the eight Trinity Applications, see Chapter 25.

Gustafson-Barsis' law A different view on **Amdahl's law** that factors in the fact that as problem sizes grow, the serial portion of computations tend to shrink as a percentage of the total work to be done, see Chapter 1.

Hardware thread A hardware implementation of a task with a separate flow of control. Multiple hardware threads can be implemented using multiple cores or can run concurrently or simultaneously on one core in order to hide latency using methods such as hyperthreading of a processor core.

hbwalloc Specific memory allocation routine interfaces for MCDRAM, part of the **memkind** project, see Chapter 3.

High-Performance Computing (HPC) The highest performance computing available at a point in time, which today generally means at least a teraFLOP/s of computational capability. The term HPC is occasionally used as a synonym for supercomputing although supercomputing is probably more specific to even higher performance systems (today, at least 10 teraFLOP/s). While the use of HPC is spreading to more industries, it is generally associated with solution of the most challenging problems in science and engineering.

Host processor The main control processor in a system, as opposed to any graphics processors or coprocessors. The host processor is responsible for booting and running the operating system.

Hot Teams Intel OpenMP control to make nested OpenMP more efficient.

HPC See High-Performance Computing (HPC).

Hybrid memory modes Memory modes in which some of MCDRAM is memory (75% or 50%) and some is cache (25% or 50%), see Chapters 3 and 4.

Hyperthreading Multithreading on a single-processor core with the purpose of more fully utilizing the functional units in an out-of-order core by bringing together instructions from more than one software thread. With hyperthreading, multiple hardware threads may run on one core and share resources, but some benefit is still obtained from parallelism or concurrency. Typically each hyperthread has, at least, its own register file and program counter so that switching between hyperthreads is relatively lightweight. Knights Landing threads are hyperthreads.

IA A commonly used abbreviation for Intel architecture, also referred to as x86 in reference to Intel's original 8088 and 8086 processors that implemented Intel architecture.

Icache Cache for instructions (not data); unified caches handle instructions and data both.

IMCI (Intel® Initial Many Core Instructions) is the official name for the SIMD instructions used by the Knights Corner coprocessor. The Knights Corner coprocessor was the first device to offer 512-bit wide SIMD instructions. AVX-512 replaces IMCI in a fashion that is supported in processors, starting with Knights Landing. See Chapter 6 on differences between IMCI and AVX-512 and how to span the differences.

Inlining An optimization that replaces a call to a subroutine or function with the actual code from the subroutine or function. This can be done by a compiler or done by hand in the source code. This optimization has to be weighed against the disadvantages of increasing the size of the program. Some sophisticated compilers can do partial inlining to somewhat address this. A subroutine or function may be inlined by a compiler at the call site to improve performance by two methods: (1) remove the call overhead and (2) enable optimizations by bringing the code together instead of having code separated by a call.

Inspector An Intel analysis tool specializing in finding threading and memory related errors in a program. It can detect latent threading bugs (ones that are not causing program failure).

Integer Execution Unit (IEU) A Knights Landing microarchitecture unit, which executes integer μops, which are defined as those that operate on general-purpose registers, see Chapter 4.

Integrated Performance Primitives (IPP) An Intel Performance Library, see Chapter 13.

Intrinsics Functions in a language that are supported by the compiler directly. In the case of AVX-512 or other vector intrinsics, the intrinsic function may map directly to a small number, often one, of machine instructions, which the compiler inserts without the overhead of a real function call.

IPP Integrated Performance Primitives, see Chapter 13.

ISA Instruction Set Architecture.

ITLB A TLB (Translation Lookaside Buffer) for instructions (not data); unified TLBs handle instructions and data both.

Lane An element of a SIMD register file and associated functional unit, considered as a unit of hardware for performing parallel computation. SIMD instructions execute computations across multiple lanes.

Language Extensions for Offloading (LEO) A feature of the Intel compiler to specify offloading regions and data movement. Which predated standardization of target directives by OpenMP. New development should be done with the OpenMP "target" instead of LEO, and LEO directives can be converted to OpenMP target in relatively straight forward manner. See Chapter 18.

Latency The time it takes to complete a task; that is, the time between when the task begins and when it ends. Latency has units of time. The scale can be anywhere from nanoseconds to days. Lower latency is better in general.

Latency hiding Schedules computations on a processing element while other tasks using that core are waiting for long-latency operations to complete, such as memory or disk transfers. The latency is not actually avoided, since each task still takes the same time to complete, but more tasks can be completed in a given time since resources are shared more efficiently, so throughput is improved.

LEO See Language Extensions for Offloading.

Load balancing Moving tasks between available resources to more quickly finish tasks in the face of uneven task durations.

Locality Utilizing memory locations that are closer, rather than further, apart. This will maximize reuse of cache lines, memory pages, and TLB entries. Maintaining a high degree of locality of reference is a key to scaling.

Lock A mechanism for implementing mutual exclusion. Before entering a mutual exclusion region, a thread must first try to acquire a lock on that region. If the lock has already been acquired by another thread, the current thread must block, which it may do by either suspending operation or spinning. When the lock is released, then the current thread is free to acquire it. Locks can be implemented using atomic operations, which are themselves a form of mutual exclusion on basic operations, implemented in hardware.

Loop-carried dependence If the same data item (e.g., element [3] of an array) is written in one iteration of a loop and is read in a different iteration of a loop, there is said to be a loop-carried dependence. If there are no loop-carried dependencies, a loop can be vectorized or parallelized. If there is a loop-carried dependence, the direction (prior iteration versus future iteration, also known as backward or forward) and the distance (the number of iterations separating the read and write) must be considered.

Many-core Platform Software Package (MPSP) A collection of patches specific to Knights Landing for a variety of open source projects. MPSP patches are generally incorporated into open sources projects gradually over time.

Many-core Platform Software Stack (MPSS) A stack of software supplied by Intel in binaries and open source to support the Intel® Xeon Phi coprocessors. It includes drivers, Intel COI, SCIF, plus mods for the coprocessor's Linux OS, gcc, and gdb.

Many-core processor A **multicore** processor with so many cores that in practice we do not enumerate them; there are just "lots." The term has been generally used with processors with 57 or more cores, but there is no precise definition.

Math Kernel Library From Intel, includes numerous routines to provide a high level of performance from this hand-optimized library. Intel MKL includes highly vectorized and threaded linear algebra, fast Fourier transforms (FFTs), vector math, and statistics functions. Through a single C or Fortran API call, these functions automatically scale across previous, current, and future processor architectures by selecting the best code path for each, see Chapter 13.

Megahertz era A historical period of time during which processors doubled clock rates at a rate similar to the doubling of transistors in a design, roughly every 2 years. Such rapid rise in processor clock speeds ceased at just under 4 GHz (4000 MHz) in 2004. Designs shifted toward adding more cores marking the shift to the **multicore era**.

memkind An open source project for a library that is a user extensible heap manager which enables control of memory characteristics and a partitioning of the heap between kinds of memory. **autohbw** and **hbwalloc** are parts of this project. This is specifically interesting in this book for MCDRAM on Knights Landing, see Chapter 3.

Memory Execution Unit (MEU) A Knights Landing microarchitecture unit that executes memory μops and services fetch requests for Icache misses and ITLB misses, see Chapter 4.

Memory Mode One of three memory modes supported by Knights Landing, i.e., Cache, Flat, and Hybrid (which has two versions itself), see Chapters 3 and 4.

MESIF A cache protocol in which cache line can be in (M)odified (E)xclusive (S)hared (I)nvalid or (F)orward state.

Message Passing Interface (MPI) An industry-standard message-passing system designed to exchange data on a wide variety of parallel computers.

MIC (Many Integrated Core Architecture) An architecture from Intel designed for highly parallel workloads. The architecture emphasizes higher core counts on a single die, and simpler more efficient cores, than on a traditional CPU. See also Xeon Phi.

Micro-op (μop) Simple operations that are executed within the core of a processor. A single complex instruction can create a very large number of micro-ops, while a simple instruction may produce only one micro-op.

Microarchitecture (μarch) The specific design for a generation of a product. Generally, a specific microarchitecture differs from others in a number of design decisions made that set the overall tone for the product line.

MILC One of the eight Trinity Applications, see Chapter 25.

MiniDFT One of the eight Trinity Applications, see Chapter 25.
MiniFE One of the eight Trinity Applications, see Chapter 25.
MiniGhost One of the eight Trinity Applications, see Chapter 25.
MKL See Math Kernel Library.
Moore's law An observation that, over the history of semiconductors, the number of transistors in a dense integrated circuits has doubled approximately every 2 years.
MPI See Message Passing Interface.
MPSP See Many-core Platform Software Package.
MPSS See Many-core Platform Software Stack.
Multichannel DRAM (MCDRAM) A stacked memory technology used on Knights Landing.
Multicore A processor with multiple subprocessors, each subprocessor (known as a **core**) supporting at least one hardware thread.
Multicore era The time after which processor designs shifted away from rapidly rising clock rates and shifted toward adding more cores. This era began roughly in 2005.
Near memory in a NUMA system is memory that has shorter access times than the far memory. The view of which parts of memory are near versus far depends on the core from which code is running.
Node (in a cluster) A shared memory computer, often on a single board with multiple processors, that is connected with other nodes to form a **cluster** computer or supercomputer.
Nondeterministic Exhibiting a lack of deterministic behavior, so results can vary from run to run of an algorithm. Concurrency is not the only factor that can lead to nondeterministic algorithms but in practice it is often the cause. See more in the definition for **deterministic**.
Non-Uniform Memory Access (NUMA) Used to categorize memory design characteristics in a distributed memory architecture. NUMA = memory access latency is different for different memories. UMA = memory access latency is same for all memory. On Knights Landing, the DDR and MCDRAMs each can exhibit UMA or NUMA characteristics depending on the cluster mode selected to operate in. See Chapter 3.
OFA (OpenFabrics Alliance) Develops, tests, licenses, supports, and distributes OpenFabrics Enterprise Distribution (OFED) open source software to deliver high-efficiency computing, wire-speed messaging, ultra-low microsecond latencies, and fast I/O. The Alliance seeks to deliver a unified, cross-platform, transport-independent software stack for RDMA and kernel bypass so that users can utilize the same OpenFabrics RDMA and kernel bypass API and run their applications agnostically over various inconnects.
OFED OpenFabrics Enterprise Distribution, see OFA.
Offload Placing part of a computation on an attached device such as a GPU or coprocessor. See also LEO, OpenMP, and OpenACC. See Chapter 18.
Offload model A programming model in which each section of an application is presumed to run on the host except when code is specifically marked for offload. Related terms: LEO, OpenMP, and OpenACC, see Chapter 18.
Offload-over-Fabric (OoF) COI to support target devices, for the offload programming model, that are actually Knights Landing nodes reached over the node interconnect fabric instead of coprocessors reached via PCIe.
On-die Something manufactured on the same piece of silicon.
On-package Something inside the package, for example, Knights Landing, but is not necessarily on the same piece of silicon (die).
OoF See Offload over Fabric.
OpenACC A set of directives for offloading created by NVidia for NVidia GPUs. Precedes standardization of OpenMP target directives.
OpenMP An API that supports multiplatform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems. It is made up of a set of compiler directives, library routines, and environment variables that influence runtime behavior. OpenMP is managed by the nonprofit technology consortium OpenMP Architecture Review Board and is jointly defined by a group of major computer hardware and software vendors (http://openmp.org), see Chapter 8.
Page The granularity at which virtual to physical address mapping is done. Within a page, the mapping of virtual to physical memory addresses is continuous.
Parallel Actually happening simultaneously. Two tasks that are both actually doing work at some point in time are considered to be operating in parallel. When a distinction is made between concurrent and parallel, the key is whether work can ever be done simultaneously. Multiplexing of a single-processor core, by multitasking operating systems, has allowed concurrency for decades even when simultaneous execution was impossible because there was only one processing core. Contrast with **concurrent**.
Parallel Studio Suite of Intel tools for node level programming (no MPI support included). Consists of C/C++ and Fortran compilers, libraries, debugging, and analysis tools.

Parallelism Doing more than one thing at a time. Attempts to classify types of parallelism are numerous.

Parallelization The act of transforming code to enable simultaneous activities. The parallelization of a program allows at least parts of it to execute in parallel.

Partitioned Global Address Space (PGAS) A parallel programming model that assumes a global memory address space that is logically partitioned with a portion local to each node but all of it is accessible, see Chapter 16.

PCIe See Peripheral Component Interconnect Express

Peel loop A loop, usually compiler generated, created to go before a highly efficient (main) loop to set up conditions needed for the efficient loop. This is commonly needed with the efficient loop assumes *N* aligned elements per iterations, usually for vectorization, and the peel loop has to do any iterations that precede the required alignment. See also remainder loop.

Performance Application Programming Interface (PAPI) A portable library interface to hardware performance counters on microprocessors, see Chapter 14.

Performance Libraries A collection of three libraries from Intel, specifically MKL, IPP, and DAAL, see Chapter 13.

Peripheral Component Interconnect Express (PCIe or PCI-E) A serial expansion bus standard for connecting computers to peripheral devices.

PGAS See Partitioned Global Address Space.

Platform Memory Topology Tables (PMTT) ACPI-defined tables that contain both bandwidth and latency information that is critical in the reliable identification of MCDRAM.

PMON Performance monitoring, see Chapter 13.

PMTT See Platform Memory Topology Tables.

PMU (Performance Monitoring Unit) A programmable portion of Knights Landing, and other Intel processors, for monitoring performance counters, see Chapter 14.

Pragmas To give a hint to a compiler, but not change the semantics of a program. OpenMP makes extensive use of pragmas. Also called a *compiler directive*.

Prefetching The act of requesting memory ahead of the actual need use the memory, generally employed to hide latency so that the actual use of the data is not delayed.

Process An application-level unit of parallel work. A process has its own thread of control and is managed by the operating system. Usually, unless special provisions are made for shared memory, a process cannot access the memory of another process.

Processing element (PE) Thread or process that is executing its own stream of instructions; MPI programmers refer to PEs as ranks.

Quadrant cluster mode One of the cluster modes, generally considered the default, see Chapters 3 and 4.

Race conditions Nondeterministic behaviors in a parallel program that is generally a programming error. A race condition occurs when concurrent tasks perform operations on the same memory location without proper synchronization, and one of the memory operations is a write. Code with a race may operate correctly sometimes and fail other times.

Recursion The act of a function being reentered while an instance of the function is still active in the same thread of execution. In the simplest and most common case, a function directly calls itself, although recursion can also occur between multiple functions. Recursion is supported by storing the state for the continuations of partially completed functions in dynamically allocated memory, such as on a stack; although if higher-order functions are supported, a more complex memory allocation scheme may be required. Bounding the amount of recursion can be important to prevent excessive use of memory.

Remainder loop A loop, usually compiler generated, created to go after a highly efficient (main) loop to clean up any remaining iterations that did not fit within the scope of the efficient loop. This is commonly needed with the efficient loop assumes *N* elements per iterations, usually for vectorization, and the remainder loop has to finish less than *N* iterations that are left over. See also peel loop.

Roofline model An estimate of the highest obtainable performance given some first-order constraints. See Chapter 10.

Scalability A measure of the increase in performance as a function of the availability of more hardware to use in parallel.

Scalable An application is scalable if its performance increases when additional parallel hardware resources are added. See **scalability**.

Scatter See gather.

SDLT See SIMD Data Layout Templates.

SDVis See Software Defined Visualization.

Serial Neither concurrent nor parallel.

Serialization When the tasks in a potentially parallel algorithm are executed in a specific serial order, typically due to resource constraints. The opposite of parallelization.

Shared memory When two units of parallel work can access data in the same location. Normally doing this safely requires synchronization. The units of parallel work, processes, threads, tasks, and fibers can all share data this way if the physical memory system allows it. However, processes do not share memory by default and special calls to the operating system are required to set it up.

SIMD Single-instruction-multiple-data referring to the ability to process multiple pieces of data (such as elements of an array) with all the same operation. SIMD is a computer architecture within a widely used classification system known as Flynn's taxonomy, first proposed in 1966.

SIMD Data Layout Templates (SDLT) Templates to help C++ programmers with AoS to SoA conversions, see Chapter 11.

SMP See Symmetric Multiprocessor.

SNAP One of the eight Trinity Applications, see Chapter 25.

SNC-2 cluster mode One of the cluster modes, divides DDR and MCDRAM each into two NUMA nodes, see Chapters 3 and 4.

SNC-4 cluster mode One of the cluster modes, divides DDR and MCDRAM each into four NUMA nodes, see Chapters 3 and 4.

Software-Defined Visualization (SDVis) An open source project to improve the visual fidelity, performance, and efficiency of prominent visualization solutions.

Software thread A virtual hardware thread; in other words, a single flow of execution in software intended to map one for one to a hardware thread. An operating system typically enables many more software threads to exist than there are actual hardware threads, by mapping software threads to hardware threads as necessary.

Spatial locality Nearby when measured in terms of distance (in memory address). Compare with temporal locality. Spatial locality refers to a program behavior where the use of one data element indicates that nearby data, often the next data element, will probably be used soon. Algorithms exhibiting good spatial locality in data usage can benefit from cache lines and prefetching hardware, both common components in modern computers.

Speedup The ratio between the latency for solving a problem with one processing unit versus the latency for solving the same problem with multiple processing units in parallel.

SPMD Single-program-multiple-data refers to the ability to process multiple pieces of data (such as elements of an array) with the same program, in contrast with a more restrictive SIMD architecture. SPMD most often refers to message passing programming on distributed memory computer architectures. SPMD is a subcategory of MIMD computer architectures within a widely used classification system known as Flynn's taxonomy, first proposed in 1966.

STL (Standard Template Library) A part of the C++ standard.

Strangled scaling A programming error in which the performance of parallel code is poor due to high contention or overhead so much so that it may underperform the nonparallel (serial) code.

Structure of Arrays (SoA) A data organization which can help parallel application performance versus SoA, see Chapter 11 and others (see Index).

Symmetric Multiprocessor (SMP) A multiprocessor system with shared memory and running a single-operating system.

Synchronization The coordination, of tasks or threads, in order to obtain the desired runtime order. Commonly used to avoid undesired race conditions.

Task A lightweight unit of potential parallelism with its own control flow. Unlike threads, tasks are usually serialized on a single core and run to completion. When contrasted with "thread," the distinction is made that tasks are pieces of work without any assumptions about where they will run, while threads have a one-to-one mapping of software threads to hardware threads. Threads are a mechanism for executing tasks in parallel, while tasks are units of work that merely provide the *opportunity* for parallel execution; tasks are not themselves a mechanism of parallel execution.

Task parallelism An attempt to classify parallelism as more oriented around tasks than data. We deliberately avoid this term, task parallelism, because its meaning varies. In particular, elsewhere "task parallelism" can refer to tasks generated by functional decomposition *or* to irregular tasks generated by data decomposition. In this book, any parallelism generated by data decomposition, regular or irregular, is considered data parallelism.

TBB See Threading Building Blocks (TBB).

Temporal locality Nearby when measured in terms of time. Compare with spatial locality. Temporal locality refers to a program behavior in which data is likely to be reused relatively soon. Algorithms exhibiting good temporal locality in data usage can benefit from data caching, which is common in modern computers. It is not unusual to be able to achieve both temporal and spatial locality in data usage. Computer systems are generally more likely to achieve optimal performance when both are achieved hence the interest in algorithm design to do so.

Teraflop/s See TFLOP/s.

TFLOP/s (TeraFLOP/s) 10^{12} Floating-Point Operations per second.

TFLOPs (TeraFLOPs) 10^{12} Floating-Point Operations.

Thread A *software* or *hardware* thread. In general, a "software thread" is any software unit of parallel work with an independent flow of control, and a "hardware thread" is any hardware unit capable of executing a single flow of control (in particular, a hardware unit that maintains a single-program counter). When "thread" is compared with "task," the distinction is made that tasks are pieces of work without any assumptions about where they will run, while threads have a one-to-one mapping of software threads to hardware threads. Threads are a mechanism for implementing tasks. A multitasking or multithreading operating system will multiplex multiple software threads onto a single-hardware

thread by interleaving execution via software-created time slices. A multicore or many-core processor consists of multiple cores to execute at least one independent software thread per core through duplication of hardware. A multithreaded or hyperthreaded processor core will multiplex a single core to execute multiple software threads through interleaving of software threads via hardware mechanisms.

Thread parallelism A mechanism for implementing parallelism in hardware using a separate flow of control for each task.

Threading Building Blocks (TBB) The most popular abstract solution for parallel programming in C++. TBB is an open source project created by Intel that has been ported to a very wide range of operating systems and processors from many vendors. OpenMP and TBB seldom compete for developers in reality. While more popular than OpenMP in terms of the number of developers using it, TBB is popular with C++ programmers whereas OpenMP is most used by Fortran and C programmers.

Throughput The rate at which those tasks are completed, given a set of tasks to be performed. Throughput measures the rate of computation, and it is given in units of tasks per unit time. See **bandwidth** and **latency**.

Tile A unit of design within Knights Landing which contains two cores, a shared L2 and the CHA, see Chapters 2 and 4.

Tiling When you divide a loop into a set of parallel tasks of a suitable granularity. In general, tiling consists of applying multiple steps on a smaller part of a problem instead of running each step on the whole problem one after the other. The purpose of tiling is to increase reuse of data in caches. Tiling can lead to dramatic performance increases when a whole problem does not fit in cache. We prefer the term "tiling" instead of "blocking" and "tile" instead of "block." Tiling and tile have become the more common term in recent times.

TLB An abbreviation for Translation Lookaside Buffer. A TLB is a specialized cache that is used to hold translations of virtual to physical page addresses. The number of elements in the TLB determines how many pages of memory can be accessed simultaneously with good efficiency. Accessing a page not in the TLB will cause a TLB miss. A TLB miss typically causes a trap to the operating system so that the page table can be referenced and the TLB updated.

Trace Analyzer and Collector An Intel tool for analyzing MPI communication traffic in order to detect opportunities for improvement, see Chapter 13.

Translation Lookaside Buffer See TLB.

Trip count The number of times a given loop will execute (trip); same thing as *iteration count*.

TSC The common name for the standard counter in modern x86 processors including the Knights Landing. See Chapter 14.

Uniform Memory Access (UMA) Used to categorize memory design characteristics in a distributed memory architecture. UMA = memory access latency is same for all memory. NUMA = memory access latency is different for different memories. On Knights Landing, the DDR and MCDRAMs each can exhibit UMA or NUMA characteristics depending on the cluster mode selected to operate in. Compare with NUMA, see Chapter 3.

Unroll Complete unrolling of a loop is accomplished by duplicating the body of the loop, for each iteration, into straight code so no loop is needed. For instance: for ($i=0$; $i<3$; i++) $a[i]=i$; can be unrolled to $a[0]=0$; $a[1]=1$; $a[2]=2$; partial unrolling retains the loop but expands the loop body to do multiple iterations each time through the loop. This is commonly done to enable vectorization. Unrolling is a common compiler optimization and has been common in source code in the past although it has a bad idea these days (see Section "Avoid Manual Loop Unrolling" in Chapter 5).

UPC A PGAS programming model, see Chapter 16.

Vector operations Low-level operation that can act on multiple data elements at once in SIMD fashion.

Vector parallelism A mechanism for implementing parallelism in hardware using the same flow of control on multiple data elements.

Vector-Processing Unit (VPU) A Knights Landing microarchitecture unit for vector and floating-point arithmetic execution responsible for providing support for x87, MMX, SSE, AVX, and AVX-512 instructions, as well as integer divides. There are two VPUs connected to the core, see Chapters 2 and 4, and programming Chapters 9–12.

Vectorization The act of transforming code to enable simultaneous computations using vector hardware. Instructions such as MMX, SSE, AVX, AVX2, and AVX-512 instructions utilize vector hardware. The vectorization of code tends to enhance performance because more data is processed per instruction than would be done otherwise. See also **vectorize**.

Vectorize Converting a program from a scalar implementation to a vectorized implementation to utilize vector hardware such as SIMD instructions (e.g., MMX, SSE, AVX, AVX2, and AVX-512). Vectorization is a specialized form of parallelism.

vi A text-based editor that was shipped with most UNIX and BSD systems written by Bill Joy, popular only to those who have yet to discover emacs (according to James)… or with those who are tired of finger-bending key-sequences of emacs (according to Avinash) (Jim can handle either).

Virtual memory The address used by software from the physical addresses of real memory. The translation from virtual addresses to physical addresses is done in hardware that is initialized and controlled by the operating system.

VPU See Vector-Processing Unit.

VTune An Intel analysis tool specializing in use of EMON counters to profile activity on processors and coprocessors, see Chapter 14.

Index

Note: Page numbers followed by *f* indicate figures.

D

E

F

K

L

M

P

Q

R

S

T

U

V

W

X

Y

Z