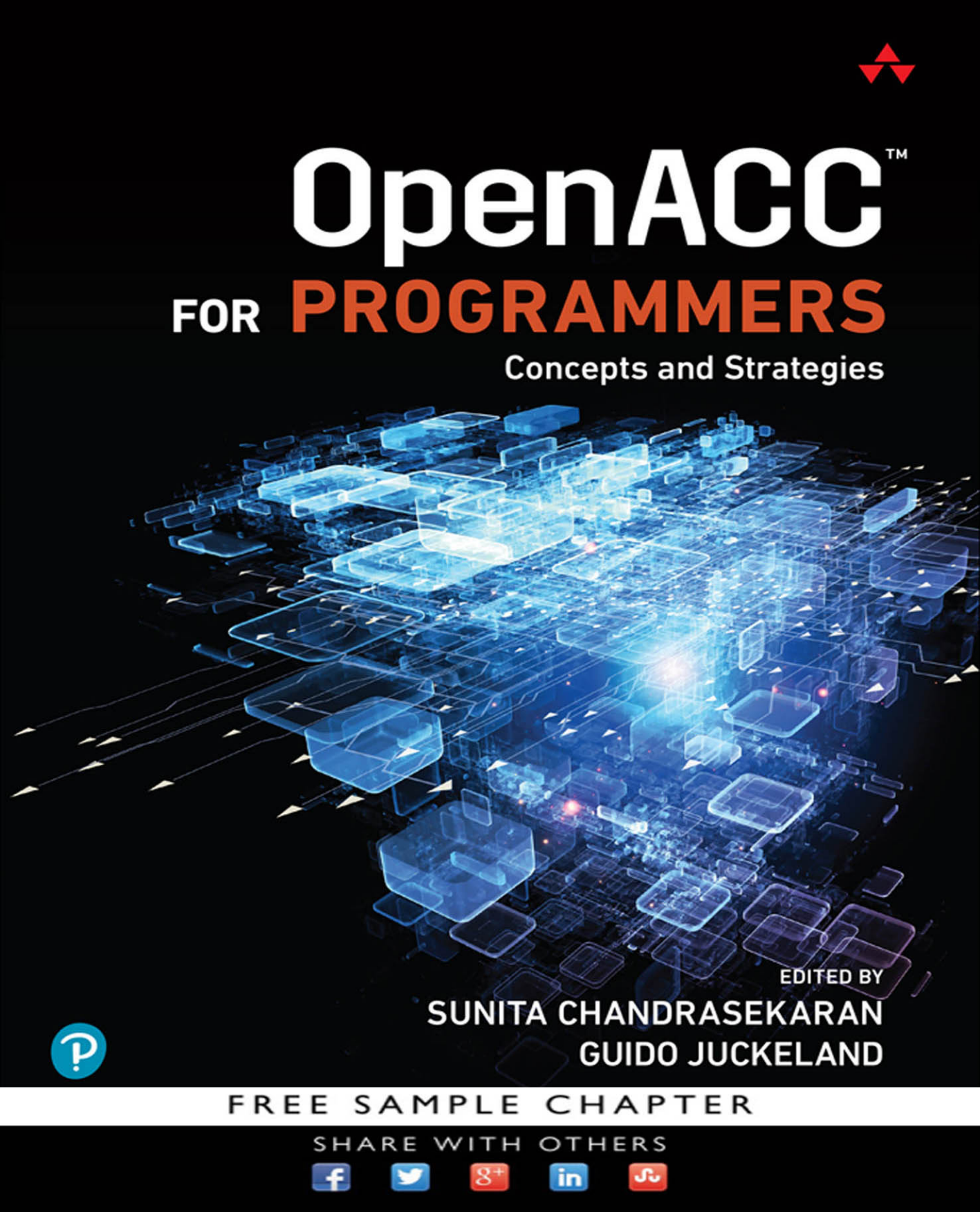
OpenACC™
FOR PROGRAMMERS
Concepts and Strategies
EDITED BY
SUNITA CHANDRASEKARAN
GUIDO JUCKELAND
FREE SAMPLE CHAPTER
SHARE WITH OTHERS

OpenACC™ for Programmers

This page intentionally left blank

OpenACC™ for Programmers

Concepts and Strategies

Edited by
Sunita Chandrasekaran
Guido Juckeland

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2017945500

ISBN-13: 978-0-13-469428-3
ISBN-10: 0-13-469428-7

1 17

To all students, programmers, and computational scientists
hungry for knowledge and discoveries—
may their work make this world a more open, tolerant,
peaceful, livable, and lovable place for all of us,
regardless of gender, origin, race, or belief!

This page intentionally left blank

Contents

Foreword

In the previous century, most computers used for scientific and technical programming consisted of one or more general-purpose processors, often called CPUs, each capable of carrying out a diversity of tasks from payroll processing through engineering and scientific calculations. These processors were able to perform arithmetic operations, move data in memory, and branch from one operation to another, all with high efficiency. They served as the computational motor for desktop and personal computers, as well as laptops. Their ability to handle a wide variety of workloads made them equally suitable for word processing, computing an approximation of the value of pi, searching and accessing documents on the web, playing back an audio file, and maintaining many different kinds of data. The evolution of computer processors is a tale of the need for speed: In a drive to build systems that are able to perform more operations on data in any given time, the computer hardware manufacturers have designed increasingly complex processors. The components of a typical CPU include the arithmetic logic unit (ALU), which performs simple arithmetic and logical operations, the control unit (CU), which manages the various components of the computer and gives instructions to the ALU, and cache, the high-speed memory that is used to store a program's instructions and data on which it operates. Most computers today have several levels of cache, from a small amount of very fast memory to larger amounts of slower memory.

Application developers and users are continuously demanding more compute power, whether their goal is to be able to model objects more realistically, analyze more data in a shorter time, or for faster high-resolution displays. The growth in compute power has enabled, for example, significant advances in the ability of weather forecasters to predict our weather for days, even weeks, in the future and for auto manufacturers to produce fuel-efficient vehicles. In order to meet that demand, the computer vendors were able to shrink the size of the different features of a processor in order to configure more transistors, the tiny devices that are actually responsible for performing calculations. But as they got smaller and more densely packed, they also got hotter and hotter. At some point, it became clear that a new approach was needed if faster processing speeds were to be obtained.

Thus multicore processing systems were born. In such a system, the actual compute logic, or core, of a processor is replicated. Each core will typically have its own ALU and CU but may share one or more levels of cache and other memory with other cores. The cores may be connected in a variety of different ways and will typically share some hardware resources, especially memory. Virtually all of our laptops, desktops, and clusters today are built from multicore processors.

Each of the multiple cores in a processor is capable of independently executing all of the instructions (such as add, multiply, and branch) that are routinely carried out by a traditional, single-core processor. Hence the individual cores may be used to run different programs simultaneously, or they can be used collaboratively to speed up a single application. The actual gain in performance that is observed by an application running on multiple cores in parallel will depend on how well it has exploited the capabilities of the individual cores and how efficiently their interactions have been managed. Challenges abound for the application developer who creates a multicore program. Ideally, each core contributes to the overall outcome continuously. For this to (approximately) happen, the workload needs to be evenly distributed among cores and organized to minimize the time that any core is waiting, possibly because it needs data that is produced on another core. Above all, the programmer must try to avoid nontrivial amounts of sequential code, or regions where only one core is active. This insight is captured in Amdahl's law, which makes the point that, no matter how fast the parallel parts of a program are, the speedup of the overall computation is bounded by the fraction of code that is sequential. To accomplish this, an application may in some cases need to be redesigned from scratch.

Many other computers are embedded in telephone systems, toys, printers, and other electronic appliances, and increasingly in household objects from washing machines to refrigerators. These are typically special-purpose computing chips that are designed to carry out a certain function or set of functions and have precisely the hardware needed for the job. Oftentimes, those tasks are all that they are able to perform. As the demands for more complex actions grow, some of these appliances today are also based on specialized multicore processors, something that increases the available compute power and the range of applications for which they are well suited.

Although the concept of computer gaming has been around since sometime in the 1960s, game consoles for home use were first introduced a decade later and didn't take off until the 1980s. Special-purpose chips were designed specifically for them, too. There was, and is, a very large market for gaming devices, and considerable effort has therefore been expended on the creation of processors that are very

efficient at rapidly constructing images for display on a screen or other output device. In the meantime, the graphics processing units (GPUs) created for this marketplace have become very powerful computing devices. Designed to meet a specific purpose, namely to enable computer gaming, they are both specialized and yet capable of running a great variety of games with potentially widely differing content. In other words, they are not general-purpose computers, but neither are they highly tuned for one very specific sequence of instructions. GPUs were designed to support, in particular, the rendering of sequences of images as smoothly and realistically as possible. When a game scene is created in response to player input—a series of images are produced and displayed at high speed—there is a good deal of physics involved. For instance, the motion of grass can be simulated in order to determine how the individual stalks will sway in the (virtual) wind, and shadow effects can be calculated and used to provide a realistic experience. Thus it is not too surprising that hardware designed for games might be suitable for some kinds of technical computing. As we shall shortly see, that is indeed the case.

Very large-scale applications such as those in weather forecasting, chemistry and pharmaceuticals, economics and financing, aeronautics, and digital movies, require significant amounts of compute power. New uses of computing that require exceptional hardware speed are constantly being discovered. The systems that are constructed to enable them are known as high-performance computing (HPC) clusters. They are built from a collection of computers, known as nodes, connected by a high-speed network. The nodes of many, although not all, such systems are built using essentially the same technology as our desktop systems. When multicore processors entered the desktop and PC markets, they were also configured as nodes of HPC platforms. Virtually all HPC platforms today have multicore nodes.

The developers and operators of HPC systems have been at the forefront of hardware innovation for many decades, and advances made in this area form the backdrop and motivation for the topic of this book. IBM's Roadrunner (installed at the Department of Energy's Los Alamos National Laboratory [LANL] in 2008) was the first computing system to achieve 1 petaflop/s (1,000 trillion floating-point calculations per second) sustained performance on a benchmark (the Linpack TOP500) that is widely used to assess a system's speed on scientific application code. Its nodes were an example of what is often called a hybrid architecture: They not only introduced dual-core processors into the node but also attached Cell processors to the multicores. The idea was that the Cell processor could execute certain portions of the code much faster than the multicore. However, the code for execution on the Cell had to be specifically crafted for it; data had to be transferred from the multicore's memory to Cell memory and the results then returned. This proved to be difficult to accomplish as a result of the tiny amount of memory available on the Cell.

People at large data centers in industry as well as at public institutions had become concerned about the rising cost of providing computing services, especially the cost of the computers' electricity consumption. Specialized cores such as the Cell were expected to offer higher computational efficiency on suitable application code at a very reasonable operating cost. Cores with these characteristics were increasingly referred to as accelerators. At LANL they encountered a major challenges with respect to the deployment of accelerators in hybrid nodes. The application code had to be nontrivially modified in order to exploit the Cell technology. Additionally, the cost of transferring data and code had to be amortized by the code speedup.

Titan (installed at the Department of Energy's Oak Ridge National Laboratory in 2013) was a landmark computing system. At 20 pflop/s (20,000 trillion calculations per second, peak) and with more than 18,000 nodes, it was significantly more powerful than Roadrunner. Its hybrid nodes, each a powerful computing system in its own right, were configured with 16-core AMD processors and an NVIDIA Tesla K20 GPU. Thus graphics processing units had entered the realm of high-performance computing in particular, and of scientific and technical computing in general. The device market had always been concerned with the power consumption of its products, and GPUs promised to deliver particularly high levels of performance with comparatively low power consumption. As with the Cell processor, however, the application programs required modification in order to be able to benefit from the GPUs. Thus the provision of a suitable programming model to facilitate the necessary adaptation was of paramount importance. The programming model that was developed to support Titan's users is the subject of this book.

Today, we are in an era of innovation with respect to the design of nodes for HPC systems. Many of the fastest machines on the planet have adopted the ideas pioneered by Titan, and hence GPUs are the most common hardware accelerators. Systems are emerging that will employ multiple GPUs in each node, sometimes with very fast data transfer capabilities between them. In other developments, technology has been under construction to enable multicore CPUs to share memory—and hence data—directly with GPUs without data transfers. Although there will still be many challenges related to the efficient use of memory, this advancement will alleviate some of the greatest programming difficulties. Perhaps more importantly, many smaller HPC systems, as well as desktop and laptop systems, now come equipped with GPUs, and their users are successfully exploiting them for scientific and technical computing. GPUs were, of course, designed to serve the gaming industry, and this successful adaptation would have been unthinkable without the success stories that resulted from the Titan installation. They, in turn, would not have been possible without an approachable programming model that meets the needs of the scientific application development community.

Other kinds of node architecture have recently been designed that similarly promise performance, programmability, and power efficiency. In particular, the idea of manycore processing has gained significant traction. A manycore processor is one that is inherently designed for parallel computing. In other words, and in contrast to multicore platforms, it is not designed to support general-purpose, sequential computing needs. As a result, each core may not provide particularly high levels of performance: The overall computational power that they offer is the result of aggregating a large number of the cores and deploying them collaboratively to solve a problem. To accomplish this, some of the architectural complexities of multicore hardware are jettisoned; this frees up space that can be used to add more, simpler cores. By this definition, the GPU actually has a manycore design, although it is usually characterized by its original purpose. Other hardware developers are taking the essential idea behind its design—a large number of cores that are intended to work together and are not expected to support the entire generality of application programs—and using it to create other kinds of manycore hardware, based on a different kind of core and potentially employing different mechanisms to aggregate the many cores. Many such systems have emerged in HPC, and innovations in this area continue.

The biggest problem facing the users of Titan, its successor platforms, and other manycore systems is related to the memory. GPUs, and other manycores, have relatively small amounts of memory per core, and, in most existing platforms, data and code that are stored on the multicore host platform must be copied to the GPU via a relatively slow communications network. Worse, data movement expends high levels of electricity, so it needs to be kept to the minimum necessary. As mentioned, recent innovations take on this problem in order to reduce the complexity of creating code that is efficient in terms of execution speed as well as power consumption. Current trends toward ever more powerful compute nodes in HPC, and thus potentially more powerful parallel desktops and laptops, involve even greater amounts of heterogeneity in the kinds of cores configured, new kinds of memory and memory organization, and new strategies for integrating the components. Although these advances will not lead to greater transparency in the hardware, they are expected to reduce the difficulty of creating efficient code employing accelerators. They will also increase the range of systems for which OpenACC is suitable.

—Dr. Barbara Chapman

Professor of Applied Mathematics and Statistics, and of Computer Science, Stony Brook University

Director of Mathematics and Computer Science, Brookhaven National Laboratory

This page intentionally left blank

Preface

Welcome to *OpenACC™ for Programmers*. This book reflects a collaborative effort from 19 highly established authors, from academia and public research as well as industry. It was the common goal of the authors to assemble a collection of chapters that can be used as a systematic introduction to parallel programming using OpenACC. We designed the chapters to build on one another so that they would be useful in a classroom setting. Hence, it is highly likely that you, dear reader, are a student who signed up for this potentially daunting parallel programming class. Please rest assured that you made the right choice with this class. Compute devices no longer come in nonparallel types, and parallel programming is more important than ever.

How This Book Is Organized

It was our goal to introduce OpenACC as one way to express parallelism in small incremental steps to not overwhelm you. Here is how the book is organized.

- The first three chapters serve as an introduction to the concepts behind OpenACC and the tools for OpenACC development.
- Chapters 4–7 take you through your first real-world OpenACC programs and reveal the magic behind compiling OpenACC programs, thereby introducing additional concepts.
- Chapter 8–10 cover advanced topics, such as alternatives to OpenACC, low-level device interaction, multidevice programming, and task parallelism.
- Chapters 11 and 12 serve as a look into the diverse field of research in OpenACC implementation of potential new language features.

Most chapters contain a few exercises at the end to review the chapter contents. The solutions as well as the code examples used in the chapters are available online at https://github.com/OpenACCUserGroup/openacc_concept_strategies_book. This URL also presents a slide deck for each chapter to help teachers kick-start their classes.

Join OpenACC User Group and Register on Informit.com

Because it has been our pleasure to work with so many friends from the (extended) OpenACC family on this book, we also want to extend an invitation to you to join the OpenACC User Group and become a family member as well. You can find access to all OpenACC resources at https://www.openacc.org.

Register your copy of *OpenACC™ for Programmers* at informit.com/register for convenient access to downloads, updates, and/or corrections as they become available (you must log in or create a new account). Enter the product ISBN (9780134694283) and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products." If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

This book would not have been possible without a multitude of people who are not listed as contributors. The idea of the book was originated by Duncan Poole, the longtime OpenACC president. He wanted to offer not only online material but also really old-fashioned printed material so that students and interested readers can use this book to uncover the magic of parallel programming with OpenACC. When Duncan could not pursue this idea any further, he passed the torch to Sunita and Guido, and the result is now finally in all our hands.

We are eternally grateful to our helpers in keeping the flame going:

- Pat Brooks and Julia Levites from NVIDIA, for bringing us in contact with publishers and answering questions that require inside knowledge
- Laura Lewin and Sheri Replin—our editors—and production manager Rachel Paul and copy editor Betsy Hardinger for guiding us safely through the maze of actually generating a book
- Our chapter reviewers: Mat Colgrove, Rob Faber, Kyle Friedline, Roberto Gomperts, Mark Govett, Andreas Herten, Maxime Hugues, Max Katz, John Larson, Junjie Li, Sanhu Li, Meifeng Lin, Georgios Markomanolis, James Norris, Sergio Pino, Ludwig Schneider, Thomas Schwinge, Anne Severt, and Peter Steinbach

Some chapters would not have been possible without assistants to the contributors. Many thanks to Lingda Li, Masahiro Nakao, Hitoshi Murai, Mitsuhisa Sato, Akihiro Tabuchi, and Taisuke Boku!

Have we already thanked our contributors who went with us on this crazy journey, never let us down, and kept delivering content on time?

THANK YOU all.

—Sunita Chandrasekaran and Guido Juckeland

This page intentionally left blank

About the Contributors

Randy Allen is director of advanced research in the Embedded Systems Division of Mentor Graphics. His career has spanned research, advanced development, and start-up efforts centered around optimizing application performance. Dr. Allen has consulted on or directly contributed to the development of most HPC compiler efforts. He was the founder of Catalytic, Inc. (focused on compilation of MATLAB for DSPs), as well as a cofounder of Forte Design Systems (high-level synthesis). He has authored or coauthored more than 30 papers on compilers for high-performance computing, simulation, high-level synthesis, and compiler optimization, and he coauthored the book *Optimizing Compilers for Modern Architectures*. Dr. Allen earned his AB summa cum laude in chemistry from Harvard University, and his PhD in mathematical sciences from Rice University.

James Beyer is a software engineer in the NVIDIA GPU software organization. He is currently a cochair of the OpenMP accelerator subcommittee as well as a member of both the OpenMP language committee and the OpenACC technical committee. Prior to joining NVIDIA, James was a member of the Cray compiler optimization team. While at Cray he helped write the original OpenACC specification. He was also a member of the Cray OpenMP and OpenACC runtime teams. He received his PhD in CS/CE from the University of Minnesota.

Sunita Chandrasekaran is an assistant professor and an affiliated faculty with the Center for Bioinformatics & Computational Biology (CBCB) at the University of Delaware. She has coauthored chapters in the books *Programming Models for Parallel Computing*, published by MIT Press, and *Parallel Programming with OpenACC*, published by Elsevier, 2016. Her research areas include exploring high-level programming models and its language extensions, building compiler and runtime implementations and validating and verifying implementations and their conformance to standard specifications. She is a member of the OpenMP, OpenACC, and SPEC HPG communities. Dr. Chandrasekaran earned her PhD in computer science engineering from Nanyang Technological University (NTU), Singapore, for creating a high-level software stack for FPGAs.

Barbara Chapman is a professor of applied mathematics and statistics, and of computer science, at Stony Brook University, where she is also affiliated with the Institute for Advanced Computational Science. She also directs Computer Science and Mathematics Research at the Brookhaven National Laboratory. She has performed research on parallel programming interfaces and the related implementation technology for more than 20 years and has been involved in several efforts to develop community standards for parallel programming, including OpenMP, OpenACC, and OpenSHMEM. Her group created the OpenUH compiler that enabled practical experimentation with proposed extensions and implementation techniques. Dr. Chapman has coauthored more than 200 papers and two books. She obtained a BSc with 1st Class Honours in mathematics from the University of Canterbury, and a PhD in computer science from Queen's University of Belfast.

Robert Dietrich studied information systems technology at the TU Dresden and graduated in 2009. His focus as a junior researcher and his diploma thesis were about programming of FPGAs in the context of high-performance computing. After graduation, he worked as research associate on the support of hardware accelerators and coprocessors in known performance tools such as Score-P and Vampir. His research interests revolve around programming and analysis of scalable heterogeneous applications.

Lin Gan is a postdoctoral research fellow in the Department of Computer Science and Technology at Tsinghua University, and the assistant director of the National Supercomputing Center in Wuxi. His research interests include HPC solutions to geo-science applications based on hybrid platforms such as CPUs, FPGAs, and GPUs. Gan has a PhD in computer science from Tsinghua University and has been awarded the ACM Gordon Bell Prize (2016), the Tsinghua-Inspur Computational Geosciences Youth Talent Award (2016), and the most significant paper award by FPL 2015.

David Gutzwiller is a software engineer and head of high-performance computing at NUMECA-USA, based in San Francisco, CA. David joined NUMECA in 2009 after completion of a graduate degree in aerospace engineering from the University of Cincinnati. His graduate research was focused on the automated structural design and optimization of turbomachinery components. Since joining NUMECA, David has worked on the adaptation of the FINE/Turbo and FINE/Open CFD solvers for use in a massively parallel, heterogeneous environment. In collaboration with industry users, David has constructed frameworks for intelligently driven design and optimization leveraging leadership supercomputers at scale.

Oscar Hernandez is a staff member of the Computer Science and Mathematics Division at Oak Ridge National Laboratory. He works on the programming environment for the next-generation leadership class machines for NCCS and OLCF. His research focuses on programming languages and compilers, static analysis tools, performance tools integration, and optimization techniques for parallel languages, especially OpenMP and accelerator directives. He represents ORNL at the OpenACC and OpenMP ARB standard organizations and collaborates with the SPEC/HPG effort.

Adrian Jackson is a research architect at EPCC, The University of Edinburgh. He leads the Intel Parallel Computing Centre at EPCC and specializes in optimizing applications on very large resources and novel computing hardware. He is also active in support and training for high-performance computing, leading the HPC Architectures course in EPCC's MSc program in HPC and running a range of training courses on all aspects of parallel computing around the United Kingdom.

Guido Juckeland just founded the Computational Science Group at Helmholtz-Zentrum Dresden-Rossendorf (HZDR), Germany. He is responsible for designing and implementing end-to-end research IT-workflows together with scientists and IT experts at HZDR. His research focuses on better usability and programmability for hardware accelerators and application performance monitoring as well as optimization. He is the vice-chair of the SPEC High Performance Group (HPG), an active member of the OpenACC technical and marketing committees, and also contributes to the OpenMP tools working group. Guido earned his PhD in computer science from Technische Universität Dresden, Germany, for his work on trace-based performance analysis for hardware accelerators.

Jiri Kraus has more than eight years' experience in HPC and scientific computing. As a senior developer technology engineer with NVIDIA, he works as a performance expert for GPU HPC applications. At the NVIDIA Julich Applications Lab and the Power Acceleration and Design Center (PADC), Jiri collaborates with developers and scientists from the Julich Supercomputing Centre, the Forschungszentrum Julich, and other institutions in Europe. A primary focus of his work is multi-GPU programming models. Before joining NVIDIA, Jiri worked on the parallelization and optimization of scientific and technical applications for clusters of multicore CPUs and GPUs at Fraunhofer SCAI in St. Augustin. He holds a Diploma in mathematics (minor in computer science) from the University of Cologne, Germany.

Jeff Larkin is a software engineer in NVIDIA's Developer Technology group, where he specializes in porting and optimizing HPC applications to accelerated computing platforms. Additionally, Jeff is involved in the development and adoption of the OpenMP and OpenACC specifications and has authored many book chapters, blog posts, videos, and seminars to advocate use of directive-based parallel programming. Jeff lives in Knoxville, TN, with his wife and son. Prior to joining NVIDIA, he was a member of the Cray Supercomputing Center of Excellence at Oak Ridge National Laboratory, where he worked with many application development teams including two Gordon Bell prize-winning teams. He has a Master's degree in computer science from the University of Tennessee, and a Bachelor's degree in computer science from Furman University.

Jinpil Lee received his master's and PhD degree in computer science from University of Tsukuba in 2013, under the supervision of Prof. Mitsuhisa Sato. From 2013 to 2015, he was working at KISTI, the national supercomputing center in Korea, as a member of the user support department. Since 2015, he has worked at Riken AICS in Japan as a member of the programming environment research team. He has been doing research on parallel programming models and compilers for modern cluster architectures such as manycore clusters. Currently he is working on developing a programming environment for the next flagship Japanese supercomputer.

Seyong Lee is a computer scientist in the Computer Science and Mathematics Division at Oak Ridge National Laboratory. His research interests include parallel programming and performance optimization in heterogeneous computing environments, program analysis, and optimizing compilers. He received his PhD in electrical and computer engineering from Purdue University, West Lafayette, Indiana. He is a member of the OpenACC Technical Forum, and he has served as a program committee/guest editor/external reviewer for various conferences, journals, and research proposals.

Graham Lopez is a researcher in the Computer Science and Mathematics Division at Oak Ridge National Laboratory, where he works on programming environments preparation with the application readiness teams for the DOE CORAL and Exascale computing projects. Graham has published research in the areas of computational materials science, application acceleration and benchmarking on heterogeneous systems, low-level communication APIs, and programming models. He earned his MS in computer science and PhD in physics from Wake Forest University. Prior to joining ORNL, he was a research scientist at Georgia Institute of Technology, where he worked on application and numerical algorithm optimizations for accelerators.

Sameer Shende serves as the director of the Performance Research Laboratory at the University of Oregon and the president and director of ParaTools, Inc. He has helped develop the TAU Performance System, the Program Database Toolkit (PDT), and the HPCLinux distribution. His research interests include performance instrumentation, measurement and analysis tools, compiler optimizations, and runtime systems for high-performance computing systems.

Xiaonan (Daniel) Tian is a GPU compiler engineer at the PGI Compilers and Tools group at NVIDIA, where he specializes in designing and implementing languages, programming models, and compilers for high-performance computing. Prior to joining NVIDIA, Daniel worked with Dr. Barbara Chapman in her compiler research group at the University of Houston, where he received a PhD degree in computer science. Prior to his work at the University of Houston, Daniel worked on GNU tool-chain porting for a semiconductor company. His research includes computer architectures, directive-based parallel programming models including OpenACC and OpenMP, compiler optimization, and application parallelization and optimization.

Christian Trott is a high-performance computing expert with extensive experience in designing and implementing software for GPU and MIC compute clusters. He earned a Dr. rer. nat. from the University of Technology Ilmenau in theoretical physics focused on computational material research. As of 2015 Christian is a senior member of the technical staff at the Sandia National Laboratories. He is a core developer of the Kokkos programming model, with a large role in advising applications on adopting Kokkos to achieve performance portability for next-generation supercomputers. Additionally, Christian is a regular contributor to numerous scientific software projects including LAMMPS and Trilinos.

John Urbanic is a parallel computing scientist at the Pittsburgh Supercomputing Center, where he spends as much time as possible implementing extremely scalable code on interesting machines. These days that means a lot of MPI, OpenMP, and OpenACC. He now leads the Big Data efforts, which involve such things as graph analytics, machine learning, and interesting file systems. John frequently teaches workshops and classes on all of the above and is most visible as the lead for the NSF XSEDE Monthly Workshop Series, the Summer Boot Camp, and the International HPC Summer School on HPC Challenges in Computational Sciences. John graduated with physics degrees from Carnegie Mellon University (BS) and Pennsylvania State University (MS) and still appreciates working on applications that simulate real physical phenomena. He is an honored recipient of the Gordon Bell Prize but still enjoys working on small embedded systems and real-time applications for various ventures. Away from the keyboard he swaps into a very different alter ego.

This page intentionally left blank

Chapter 4

Using OpenACC for Your First Program

John Urbanic, Pittsburgh Supercomputing Center

In this chapter, you'll parallelize real code. You will start with code that does something useful. Then you'll consider how you might use OpenACC to speed it up. You will see that reducing data movement is key to achieving significant speedup, and that OpenACC gives you the tools to do so. By the end of the chapter you will be able to call yourself an OpenACC programmer—a fledgling one, perhaps, but on your way. Let's jump right into it.

4.1 Case Study

You are reading a book about OpenACC programming, so it's a safe bet the authors are fans of this approach to parallel programming. Although that's a perfectly sensible thing, it has its dangers. It is tempting for enthusiasts to cherry-pick examples that make it seem as if their favored technology is perfect for everything. Anyone with experience in parallel programming has seen this before. We are determined not to do that here.

Our example is so generically useful that it has many applications, and it is often used to demonstrate programming with other parallel techniques as well, such as the somewhat related OpenMP and the very different MPI. So, rest assured, we haven't rigged the game.

Another reason we prefer this example is that both the "science" and the numerical method are intuitive. Although we will solve the Laplace equation for steady-state temperature distribution using Jacobi iteration, we don't expect that you immediately know what that means.

Let's look at the physical problem. You have a square metal plate. It is initially at zero degrees. This is termed, unsurprisingly, the **initial conditions.** You will heat two of the edges in an interesting pattern where you heat the lower-right corner (as pictured in Figure 4.1A) to 100 degrees. You control the two heating elements that lead from this corner such that they go steadily to zero degrees at their farthest edge. The other two edges you will hold at zero degrees. These four edges constitute the **boundary conditions.**

For the metal plate, you would probably guess the ultimate solution should look something like Figure 4.1B.

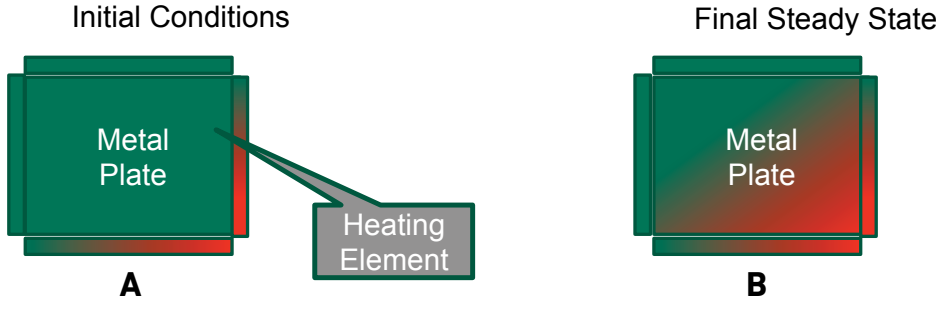

Figure 4.1 A heated metal plate

You have a very hot corner, a very cold corner, and some kind of gradient in between. This is what the ultimate, numerically solved solution should look like.

If you are wondering whether this is degrees centigrade or Fahrenheit, or maybe Kelvin, you are overthinking the problem. If you have a mathematical method or numerical background, you should be interested to know that the equation that governs heat distribution is the Laplace equation:

$$\nabla^2 T = 0$$

Although this equation has many interesting applications, including electrostatics and fluid flow, and many fascinating mathematical properties, it also has a straightforward and intuitive meaning in this context. It simply means that the value of interest (in our case, temperature) at any point is the average of the neighbor's

values. This makes sense for temperature; if you have a pebble and you put a cold stone on one side and a hot stone on the other, you'd probably guess that the pebble would come to the average of the two. And in general, you would be right.

4.1.1 SERIAL CODE

Let's represent the metal plate using a grid, which becomes a typical two-dimensional array in code. The Laplace equation says that every point in the grid should be the average of the neighbors. This is the state you will solve for.

The simulation starting point—the set of initial conditions—is far from this. You have zero everywhere except some big jumps along the edges where the heating elements are. You want to end up with something that resembles the desired solution.

There are many ways you can find this solution, but let's pick a particularly straightforward one: Jacobi iteration. This method simply says that if you go over your grid and set each element equal to the average of the neighbors, and keep doing this, you will eventually converge on the correct answer. You will know when you have reached the right answer because when you make your averaging pass, the values will already be averaged (the Laplace condition) and so nothing will happen. Of course, these are floating-point numbers, so you will pick some small error, which defines "nothing happening." In this case, we will say that when no element changes more than one-hundredth of a degree, we are done. If that isn't good enough for you, you can easily change it and continue to a smaller error.

Your serial algorithm looks like this at the core.

```
for(i = 1; i <= HEIGHT; i++) {
    for(j = 1; j <= WIDTH; j++) {
        Temperature[i][j] = 0.25 * (Temperature_previous[i+1][j]
                                  + Temperature_previous[i-1][j]
                                  + Temperature_previous[i][j+1]
                                  + Temperature_previous[i][j-1]);
    }
}
```

Here it is in Fortran:

```
do j=1,width
    do i=1,height
        temperature(i,j) =0.25*(temperature_previous(i+1,j)&
                              + temperature_previous(i-1,j)&
                              + temperature_previous(i,j+1)&
                              + temperature_previous(i,j-1))
    enddo
enddo
```

Note that the C and Fortran code snippets are virtually identical in construction. This will remain true for the entire program.

This nested loop is the guts of the method and in some sense contains all the science of the simulation. You are iterating over your metal plate in both dimensions and setting every interior point equal to the average of the neighbors (i.e., adding together and dividing by 4). You don't change the very outside elements; those are the heating elements (or boundary conditions). There are a few other items in the main iteration loop as it repeats until convergence. Listing 4.1 shows the C code, and Listing 4.2 shows the Fortran code.

Listing 4.1 C Laplace code main loop

```
while ( worst_dt > TEMP_TOLERANCE ) {

    for(i = 1; i <= HEIGHT; i++) {
        for(j = 1; j <= WIDTH; j++) {
            Temperature[i][j] = 0.25 * (Temperature_previous[i+1][j]
                                + Temperature_previous[i-1][j]
                                + Temperature_previous[i][j+1]
                                + Temperature_previous[i][j-1]);
        }
    }

    worst_dt = 0.0;

    for(i = 1; i <= HEIGHT; i++){
        for(j = 1; j <= WIDTH; j++){
            worst_dt = fmax( fabs(Temperature[i][j]-
                                  Temperature_previous[i][j]),
                             worst_dt);
            Temperature_previous[i][j] = Temperature[i][j];
        }
    }

    if((iteration % 100) == 0) {
        track_progress(iteration);
    }

    iteration++;
}
```

Listing 4.2 Fortran Laplace code main loop

```
do while ( worst_dt > temp_tolerance )

    do j=1,width
       do i=1,height
           temperature(i,j) =0.25*(temperature_previous(i+1,j)&
                                 + temperature_previous(i-1,j)&
                                 + temperature_previous(i,j+1)&
                                 + temperature_previous(i,j-1))
        enddo
    enddo

    worst_dt=0.0

    do j=1,width
       do i=1,height
          worst_dt = max( abs(temperature(i,j) - &
                              temperature_previous(i,j)),&
                          worst_dt )
          temperature_previous(i,j) = temperature(i,j)
       enddo
    enddo

    if( mod(iteration,100).eq.0 ) then
        call track_progress(temperature, iteration)
    endif

    iteration = iteration+1

enddo
```

The important addition is that you have a second array that keeps the temperature data from the last iteration. If you tried to use one array, you would find yourself using some updated neighboring elements and some old neighboring elements from the previous iteration as you were updating points in the grid. You need to make sure you use only elements from the last iteration.

While you are doing this nested loop copy to your backup array (and moving all this data around in memory), it's a good time to look for the worst (most changing) element in the simulation. When the worst element changes only by 0.01 degree, you know you are finished.

It might also be nice to track your progress as you go; it's much better than staring at a blank screen for the duration. So, every 100 iterations, let's call a modest output routine.

That is all there is to it for your serial Laplace Solver. Even with the initialization and output code, the full program clocks in at fewer than 100 lines. (See Listing 4.3 for the C code, and Listing 4.4 for Fortran.)

Listing 4.3 Serial Laplace Solver in C

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

#define WIDTH      1000
#define HEIGHT     1000
#define TEMP_TOLERANCE 0.01

double Temperature[HEIGHT+2][WIDTH+2];
double Temperature_previous[HEIGHT+2][WIDTH+2];

void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {

  int i, j;
  int iteration=1;
  double worst_dt=100;
  struct timeval start_time, stop_time, elapsed_time;

  gettimeofday(&start_time,NULL);

  initialize();

  while ( worst_dt > TEMP_TOLERANCE ) {

    for(i = 1; i <= HEIGHT; i++) {
      for(j = 1; j <= WIDTH; j++) {
        Temperature[i][j] = 0.25 * (Temperature_previous[i+1][j]
                                  + Temperature_previous[i-1][j]
                                  + Temperature_previous[i][j+1]
                                  + Temperature_previous[i][j-1]);
      }
    }

    worst_dt = 0.0;

    for(i = 1; i <= HEIGHT; i++){
      for(j = 1; j <= WIDTH; j++){
         worst_dt = fmax( fabs(Temperature[i][j]-
                               Temperature_previous[i][j]),
```

```
                          worst_dt);
          Temperature_previous[i][j] = Temperature[i][j];
        }
      }

      if((iteration % 100) == 0) {
        track_progress(iteration);
      }

      iteration++;
    }

    gettimeofday(&stop_time,NULL);
    timersub(&stop_time, &start_time, &elapsed_time);

    printf("\nMax error at iteration %d was %f\n",
           iteration-1, worst_dt);
    printf("Total time was %f seconds.\n",
           elapsed_time.tv_sec+elapsed_time.tv_usec/1000000.0);
}

void initialize(){

    int i,j;

    for(i = 0; i <= HEIGHT+1; i++){
      for (j = 0; j <= WIDTH+1; j++){
        Temperature_previous[i][j] = 0.0;
      }
    }

    for(i = 0; i <= HEIGHT+1; i++) {
      Temperature_previous[i][0] = 0.0;
      Temperature_previous[i][WIDTH+1] = (100.0/HEIGHT)*i;
    }

    for(j = 0; j <= WIDTH+1; j++) {
      Temperature_previous[0][j] = 0.0;
      Temperature_previous[HEIGHT+1][j] = (100.0/WIDTH)*j;
    }
}

void track_progress(int iteration) {

    int i;

    printf("---------- Iteration number: %d ------------\n",
           iteration);
    for(i = HEIGHT-5; i <= HEIGHT; i++) {
      printf("[%d,%d]: %5.2f  ", i, i, Temperature[i][i]);
    }
    printf("\n");
}
```

Listing 4.4 Fortran version of serial Laplace Solver

```
program serial
  implicit none

  integer, parameter             :: width=1000
  integer, parameter             :: height=1000
  double precision, parameter    :: temp_tolerance=0.01

  integer                        :: i, j, iteration=1
  double precision               :: worst_dt=100.0
  real                           :: start_time, stop_time

  double precision, dimension(0:height+1,0:width+1) :: &
                              temperature, temperature_previous

  call cpu_time(start_time)

  call initialize(temperature_previous)

  do while ( worst_dt > temp_tolerance )

    do j=1,width
      do i=1,height
        temperature(i,j) = 0.25* (temperature_previous(i+1,j)&
                            + temperature_previous(i-1,j)&
                            + temperature_previous(i,j+1)&
                            + temperature_previous(i,j-1))
      enddo
    enddo

    worst_dt=0.0

    do j=1,width
      do i=1,height
        worst_dt = max( abs(temperature(i,j) - &
                            temperature_previous(i,j)),&
                        worst_dt )
        temperature_previous(i,j) = temperature(i,j)
      enddo
    enddo

    if( mod(iteration,100).eq.0 ) then
      call track_progress(temperature, iteration)
    endif

    iteration = iteration+1

  enddo

  call cpu_time(stop_time)

  print*, 'Max error at iteration ', iteration-1, ' was ', &
          worst_dt
  print*, 'Total time was ',stop_time-start_time, ' seconds.'
end program serial
```

```
subroutine initialize( temperature_previous )
  implicit none
  integer, parameter             :: width=1000
  integer, parameter             :: height=1000
  integer                        :: i,j
  double precision, dimension(0:height+1,0:width+1) :: &
                    temperature_previous

  temperature_previous = 0.0

  do i=0,height+1
    temperature_previous(i,0) = 0.0
    temperature_previous(i,width+1) = (100.0/height) * i
  enddo

  do j=0,width+1
    temperature_previous(0,j) = 0.0
    temperature_previous(height+1,j) = ((100.0)/width) * j
  enddo
end subroutine initialize

subroutine track_progress(temperature, iteration)
  implicit none
  integer, parameter             :: width=1000
  integer, parameter             :: height=1000
  integer                        :: i,iteration

  double precision, dimension(0:height+1,0:width+1) :: &
        temperature

  print *, '------- Iteration number: ', iteration, ' ------'
  do i=5,0,-1
    write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no') &
              height-i,width-i,temperature(height-i,width-i)
  enddo
  print *
end subroutine track_progress
```

4.1.2 COMPILING THE CODE

Take a few minutes to make sure you understand the code fully. In addition to the main loop, you have a small bit of initialization, a timer to aid in optimizing, and a basic output routine. This code compiles as simply as

```
pgcc laplace.c
```

Here it is for the PGI compiler:

```
pgcc laplace.f90
```

We use PGI for performance consistency in this chapter. Any other standard compiler would work the same. If you run the resulting executable, you will see something like this:

```
. . .
. . .
---------- Iteration number: 3200 ------------
. . . [998,998]: 99.18  [999,999]: 99.56  [1000,1000]: 99.86
---------- Iteration number: 3300 ------------
. . . [998,998]: 99.19  [999,999]: 99.56  [1000,1000]: 99.87

Max error at iteration 3372 was 0.009995
Total time was 21.344162 seconds.
```

The output shows that the simulation looped 3,372 times before all the elements stabilized (to within our 0.01 degree tolerance). If you examine the full output, you can see the elements converge from their zero-degree starting point.

The times for both the C and the Fortran version will be very close here and as you progress throughout optimization. Of course, the time will vary depending on the CPU you are using. In this case, we are using an Intel Broadwell running at 3.0 GHz. At the time of this writing, it is a very good processor, so our eventual speedups won't be compared against a poor serial baseline.

This is the last time you will look at any code outside the main loop. You will henceforth exploit the wonderful ability of OpenACC to allow you to focus on a small portion of your code—be it a single routine, or even a single loop—and ignore the rest. You will return to this point when you are finished.

4.2 Creating a Naive Parallel Version

In many other types of parallel programming, you would be wise to stare at your code and plot various approaches and alternative algorithms before you even consider starting to type. With OpenACC, the low effort and quick feedback allow you to dive right in and try some things without much risk of wasted effort.

4.2.1 FIND THE HOT SPOT

Almost always the first thing to do is find the **hot spot:** the point of highest numerical intensity in your code. A profiler like those you've read about will quickly locate and

rank these spots. Often, as is the case here, it is obvious where to start. A large loop is a big flag, and you have two of them within the main loop. This is where we focus.

4.2.2 IS IT SAFE TO USE KERNELS?

The biggest hammer in your toolbox is the `kernels` directive. Refer to Chapter 1 for full details on `kernels`. Don't resist the urge to put it in front of some large, nested loop. One nice feature about this directive is that it is safe out of the box; until you start to override its default behavior with additional directives, the compiler will be able to see whether there are any code-breaking dependencies, and it will make sure that the device has access to all the required data.

4.2.3 OPENACC IMPLEMENTATIONS

Let's charge ahead and put `kernels` directives in front of the two big loops. The C and Fortran codes become the code shown in Listings 4.5 and 4.6.

Listing 4.5 C Laplace code main loop with kernels directives

```
while ( worst_dt > TEMP_TOLERANCE ) {

   #pragma acc kernels
   for(i = 1; i <= HEIGHT; i++) {
      for(j = 1; j <= WIDTH; j++) {
         Temperature[i][j] = 0.25 * (Temperature_previous[i+1][j]
                                   + Temperature_previous[i-1][j]
                                   + Temperature_previous[i][j+1]
                                   + Temperature_previous[i][j-1]);
      }
   }

   worst_dt = 0.0;

   #pragma acc kernels
   for(i = 1; i <= HEIGHT; i++){
      for(j = 1; j <= WIDTH; j++){
         worst_dt = fmax( fabs(Temperature[i][j]-
                               Temperature_previous[i][j]),
                          worst_dt);
        Temperature_previous[i][j] = Temperature[i][j];
      }
   }

   if((iteration % 100) == 0) {
      track_progress(iteration);
   }

   iteration++;
}
```

Listing 4.6 Fortran Laplace code main loop with kernels directives

```
do while ( worst_dt > temp_tolerance )

   !$acc kernels
   do j=1,width
      do i=1,height
         temperature(i,j) =0.25*(temperature_previous(i+1,j)&
                               + temperature_previous(i-1,j)&
                               + temperature_previous(i,j+1)&
                               + temperature_previous(i,j-1))
      enddo
   enddo
   !$acc end kernels

   worst_dt=0.0

   !$acc kernels
   do j=1,width
      do i=1,height
         worst_dt = max( abs(temperature(i,j) - &
                             temperature_previous(i,j)),&
                         worst_dt )
         temperature_previous(i,j) = temperature(i,j)
      enddo
   enddo
   !$acc end kernels

   if( mod(iteration,100).eq.0 ) then
      call track_progress(temperature, iteration)
   endif

   iteration = iteration+1

enddo
```

The compilation is also straightforward. All you do is activate the directives using, for example, the PGI compiler, for the C version:

```
pgcc -acc laplace.c
```

Or for the Fortran version:

```
pgf90 -acc laplace.f90
```

If you do this, the executable pops right out and you can be on your way. However, you probably want to verify that your directives actually did something. OpenACC's defense against compiling a loop with dependencies or other issues is to simply ignore the directives and deliver a "correct," if unaccelerated, executable. With the PGI compiler, you can request feedback on the C OpenACC compilation by using this:

```
pgcc -acc -Minfo=acc laplace.c
```

Here it is for Fortran:

```
pgf90 -acc -Minfo=acc laplace.f90
```

Similar options are available for other compilers. Among the informative output, you see the "Accelerator kernel generated" message for both of your kernels-enabled loops. You may also notice that a reduction was automatically generated for `worst_dt`. It was nice of the compiler to catch that and generate the reduction automatically. So far so good.

If you run this executable, you will get something like this:

```
. . .
. . .
---------- Iteration number: 3200 ------------
. . .[998,998]: 99.18  [999,999]: 99.56  [1000,1000]: 99.86
---------- Iteration number: 3300 ------------
. . .[998,998]: 99.19  [999,999]: 99.56  [1000,1000]: 99.87

Max error at iteration 3372 was 0.009995
Total time was 35.258830 seconds.
```

This was executed on an NVIDIA K80, the fastest GPU available at the time of this writing. For our efforts thus far, we have managed to slow down the code by about 70 percent, which is not impressive at all.

4.3 Performance of OpenACC Programs

Why did the code slow down? The first suspect that comes to mind for any experienced GPU programmer is data movement. The device-to-host memory bottleneck is usually the culprit for such a disastrous performance as this. That indeed turns out to be the case.

You could choose to use a sophisticated performance analysis tool, but in this case, the problem is so egregious you can probably find enlightenment with something as simple as the PGI environment profiling option:

```
export PGI_ACC_TIME=1
```

If you run the executable again with this option enabled, you will get additional output, including this:

```
Accelerator Kernel Timing data
 main  NVIDIA  devicenum=0
  time(us): 11,460,015
```

```
31: compute region reached 3372 times
 33: kernel launched 3372 times
  grid: [32x250]  block: [32x4]
   device time(us): total=127,433 max=54 min=37 avg=37
  elapsed time(us): total=243,025 max=2,856 min=60 avg=72
31: data region reached 6744 times
 31: data copyin transfers: 3372
  device time(us): total=2,375,875 max=919 min=694 avg=704
 39: data copyout transfers: 3372
  device time(us): total=2,093,889 max=889 min=616 avg=620
41: compute region reached 3372 times
 41: data copyin transfers: 3372
  device time(us): total=37,899 max=2,233 min=6 avg=11
 43: kernel launched 3372 times
  grid: [32x250]  block: [32x4]
   device time(us): total=178,137 max=66 min=52 avg=52
  elapsed time(us): total=297,958 max=2,276 min=74 avg=88
 43: reduction kernel launched 3372 times
  grid: [1]  block: [256]
   device time(us): total=47,492 max=25 min=13 avg=14
  elapsed time(us): total=136,116 max=1,011 min=32 avg=40
 43: data copyout transfers: 3372
  device time(us): total=60,892 max=518 min=13 avg=18
41: data region reached 6744 times
 41: data copyin transfers: 6744
  device time(us): total=4,445,950 max=872 min=651 avg=659
 49: data copyout transfers: 3372
  device time(us): total=2,092,448 max=1,935 min=616 avg=620
```

The problem is not subtle. The line numbers 31 and 41 correspond to your two `kernels` directives. Each resulted in a lot of data transfers, which ended up using most of the time. Of the total sampled time of 11.4 seconds (everything is in microseconds here), well over 10s was spent in the data transfers, and very little time in the compute region. That is no surprise given that we can see multiple data transfers for every time a kernels construct was actually launched. How did this happen?

Recall that the `kernels` directive does the safe thing: When in doubt, copy any data used within the kernel to the device at the beginning of the kernels region, and off at the end. This paranoid approach guarantees correct results, but it can be expensive. Let's see how that worked in Figure 4.2.

What OpenACC has done is to make sure that each time you call a device kernels, any involved data is copied to the device, and at the end of the kernels region, it is all copied back. This is safe but results in two large arrays getting copied back and forth twice for each iteration of the main loop. These are two 1,000 × 1,000 double-precision arrays, so this is (2 arrays) × (1,000 × 1,000 grid points/array) × (8 bytes/grid point) = 16MB of memory copies every iteration.

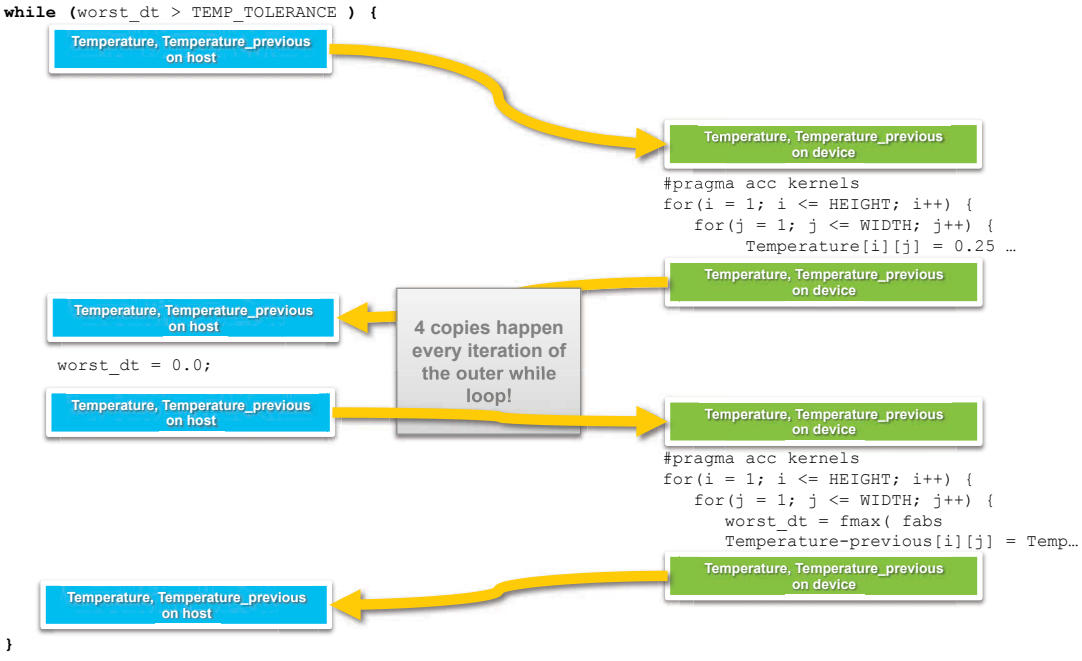

Figure 4.2 Multiple data copies per iteration

Note that we ignore `worst_dt`. In general, the cost of copying an 8-byte scalar (non-array) variable is negligible.

4.4 An Optimized Parallel Version

So far we have marked the parallel regions for acceleration. Now it is time to introduce data regions to optimize data transfers.

4.4.1 REDUCING DATA MOVEMENT

Now that you have identified the problem, you know you must apply some `data` directives. OpenACC lets you completely control the residency of the data. It has routines to set up data during program initialization, to automatically migrate data going into or out of any region or block of code, and to update at any given point in the code. So don't worry about what OpenACC *can* do. Worry about what you *want* to do.

Pause here and see whether you can come up with a strategy to minimize data movement. What directives does that strategy translate to? Feel free to experiment with the code on your own before reading the answer, which is provided later.

In general, we want the entire simulation to be executed on the device. That is certainly the ideal case and eliminates all the data transfer costs. But most of the time you can't achieve that objective; the entire problem may not fit in device memory, there may be portions of the code that must execute on the host, or IO may be required at some point.

But let's start with that objective in mind. If you load your data onto the device at the beginning of the main loop, when do you next need it on the host? Think the first iteration through as a start: there is no reason for the two big arrays to return to the host between the two kernels. They can stay on the device.

What about `worst_dt`? It is insignificant in size, so you don't care what it does as long as it is available when needed, as per the default `kernels` behavior. Once you start to use data regions, you uncouple the execution from the data regions and could prevent unnecessary data movement. Because there is no real performance gain, you won't override the default by including it in any `data` directives. It will continue to be set to 0 on the host, get to a maximum in the second nested loop (actually a reduction from all of the "local maximums" found by each processing element (PE) on the device), and get copied back to the host so that it can be checked as the condition to continue the `while` loop every iteration. Again, this is all default `kernels` behavior, so we don't worry about the details.

After that, you run into the output routine. It isn't an issue for the first 100 iterations, so let's ignore it for a moment and continue around the loop for the second iteration. At the start of the second iteration, you would like both big arrays to be on the device. That is just where you left them! So it looks as if you can just keep the data on the device between iterations of the `while` loop. The obvious `data` directives would be `data copy` clauses applied to the `while` loop.

```
// C
#pragma acc data copy(Temperature_previous, Temperature)
while ( worst_dt > TEMP_TOLERANCE ) {
. . .

! Fortran
!$acc data copy(temperature_previous, temperature)
do while ( worst_dt > temp_tolerance )
. . .
```

This is indeed the key. It will significantly speed up the code, and you will get the right answer at the end.

However, you do need to address that `track_progess()` output routine that gets invoked every 100 iterations. You need for the temperature to be back on the host at that point. Otherwise, the host copy of `temperature` will remain at the initial condition of all zeros until the data copy happens at the termination of the `while` loop, which is the end of the data region. Many programmers encounter this oversight when they apply the `data` directives, run the code to a quick completion in the expected 3,372 iterations, and assume victory, only to notice that all of their printed output has been zeros. Make sure you understand exactly how this happens, because it is a good example of what can occur when we decouple the data and execution regions using `data` directives.

The fix is easy. You just need an `update` at that point.

```
// C
. . .
if((iteration % 100) == 0) {
   #pragma acc update host(Temperature)
   track_progress(iteration);
}
. . .

! Fortran
. . .
if( mod(iteration,100).eq.0 ) then
   !$acc update host(temperature)
   call track_progress(temperature, iteration)
endif
. . .
```

It is important to realize that all the tools for convenient data management are already in OpenACC. Once you decide how you want to manage the data conceptually, some combination of `data copy`, `declare`, `enter/exit`, and `update` clauses should allow you to accomplish that as you wish. If you find yourself fighting the scope or blocking of your code to make the directives match your wishes, take a breath and ask yourself whether the other clauses will allow you to accomplish this more naturally.

4.4.2 EXTRA CLEVER TWEAKS

There is one more tweak you can apply to the code before you declare victory. If you look a little more carefully at the code, you might notice that you don't actually need to copy both big arrays into the `while` loop. It happens that `temperature_previous` is the array that is initialized in the initialization routine, and `temperature` uses these values to set itself in the first iteration. So you don't need to copy it in.

Continuing with that line of thought, you don't need for both arrays to exit the `while` loop with the final data; one will suffice. Once again, `temperature_previous` has the correct values so that you can abandon `temperature` on the device. This means that `temperature` is really just a temporary array used on the device, and there is no need to copy it in or out. That is exactly what the `data create` clause is for.

Note that this last optimization is really not very important. The big win was recognizing that you were copying the large arrays needlessly every iteration. You were copying two large arrays into and out of each of the two `kernels` each loop:

(2 arrays) × (in and out) × (2 pairs of loops) × (3,372 iterations) = 26,976 copies

Getting rid of all those transfers with a `data copy` was the big win. Using `data create` instead of `copy` for the `Temperature` array saved one copy in at the beginning of the entire run, and one copy out at the end. It wasn't significant. So don't feel bad if you didn't spot that opportunity.

Likewise, using an `update` for the track progress routine caused 33 transfers over the course of the run. It was a quick fix for the problem. In comparison to the original 26,876 copies, having 33 remaining is nothing. However now that you are down to one copy in and one copy out for the whole run, it does have an impact on the order of 5 percent of the new and significantly reduced total run time. Given the huge performance improvement you have achieved, you may not care, but for those of you seeking perfection, see Exercise 1 at the end of the chapter.

4.4.3 FINAL RESULT

Listing 4.7 shows the final C version of the OpenACC enabled routine, and Listing 4.8 shows the Fortran version.

Listing 4.7 Final C OpenACC Laplace code main loop

```
#pragma acc data copy(Temperature_previous), create(Temperature)
while ( worst_dt > TEMP_TOLERANCE ) {

   #pragma acc kernels
   for(i = 1; i <= HEIGHT; i++) {
      for(j = 1; j <= WIDTH; j++) {
         Temperature[i][j] = 0.25 * (Temperature_previous[i+1][j]
                                   + Temperature_previous[i-1][j]
                                   + Temperature_previous[i][j+1]
                                   + Temperature_previous[i][j-1]);
      }
   }

   worst_dt = 0.0;
```

```
   #pragma acc kernels
   for(i = 1; i <= HEIGHT; i++){
      for(j = 1; j <= WIDTH; j++){
         worst_dt = fmax( fabs(Temperature[i][j]-
                              Temperature_previous[i][j]),
                          worst_dt);
         Temperature_previous[i][j] = Temperature[i][j];
      }
   }

   if((iteration % 100) == 0) {
   #pragma acc update host(Temperature)
      track_progress(iteration);
   }

   iteration++;
}
```

Listing 4.8 Final Fortran OpenACC Laplace code main loop

```
!$acc data copy(temperature_previous), create(temperature)
do while ( worst_dt > temp_tolerance )

   !$acc kernels
   do j=1,width
      do i=1,height
         temperature(i,j) =0.25*(temperature_previous(i+1,j)&
                               + temperature_previous(i-1,j)&
                               + temperature_previous(i,j+1)&
                               + temperature_previous(i,j-1))
      enddo
   enddo
   !$acc end kernels

   worst_dt=0.0

   !$acc kernels
   do j=1,width
      do i=1,height
         worst_dt = max( abs(temperature(i,j) - &
                             temperature_previous(i,j)),&
                         worst_dt )
         temperature_previous(i,j) = temperature(i,j)
      enddo
   enddo
   !$acc end kernels

   if( mod(iteration,100).eq.0 ) then
      !$acc update host(temperature)
      call track_progress(temperature, iteration)
   endif

   iteration = iteration+1

enddo
!$acc end data
```

You compile exactly as before. If you again use the compiler verbose information option (`-Minfo=acc` for PGI), you see that the generated copies are now outside the `while` loop, as intended. Here is the result.

```
. . .
. . .
---------- Iteration number: 3200 ------------
. . .[998,998]: 99.18  [999,999]: 99.56  [1000,1000]: 99.86
---------- Iteration number: 3300 ------------
. . .[998,998]: 99.19  [999,999]: 99.56  [1000,1000]: 99.87

Max error at iteration 3372 was 0.009995
Total time was 1.054768 seconds.
```

This is much better. Table 4.1 sums it up. With only a handful of directives, you have managed to speed up the serial code more than 20 times. But you had to think about your data migration in order to get there. This is typical of accelerator development.

Table 4.1 Laplace code performance

OPTIMIZATION	TIME (SECONDS)	SPEEDUP
Serial	21.3	
`kernels` directive	35.2	0.60
`data` directives	1.05	20.3

To review, you looked for the large loops and placed `kernels` directives there. Then (prompted by terrible performance) you thought about how the data should really flow between the host and the device. Then you used the appropriate `data` directives to make that happen. Further performance improvements are possible (see the exercises), but you have achieved the lion's share of what can be done.

4.5 Summary

Here are all the OpenACC advantages you have used in this chapter.

- **Incremental optimization.** You focused on only the loop of interest here. You have not had to deal with whatever is going on in `track_progress()` or any other section of the code. We have not misled you with this approach. It will usually remain true for an 80,000-lines of code program with 1,200 subroutines. You may be able to focus on a single computationally intense section of the code to great effect. That might be 120 lines of code instead of our 20, but it sure beats the need to understand the dusty corners of large chunks of legacy code.

- **Single source.** This code is still entirely valid serial code. If your colleagues down the hall are oblivious to OpenACC, they can still understand the program results by simply ignoring the funny-looking comments (your OpenACC directives)—as can an OpenACC-ignorant compiler. Or a compute platform without accelerators. This isn't guaranteed to be true; you can utilize the OpenACC API instead of directives, or rearrange your code to make better use of parallel regions; and these types of changes will likely break the pure serial version. But it can be true for many nontrivial cases.

- **High level.** We have managed to avoid any discussion of the hardware specifics of our accelerator. Beyond the acknowledgment that the host-device connection is much slower than the local memory connection on either device, we have not concerned ourselves with the fascinating topic of GPU architecture at all.

- **Efficient.** Without an uber-optimized low-level implementation of this problem using CUDA or OpenCL, you have to take our word on this, but you could not do much better even with those much more tedious approaches. You can exploit the few remaining optimizations using some advanced OpenACC statements. In any event, the gains will be small compared with what you have already achieved.

- **Portable.** This code should run efficiently on any accelerated device. You haven't had to embed any platform-specific information. This won't always be true for all algorithms, and you will read more about this later in Chapter 7, "OpenACC and Performance Portability."

With these advantages in mind, we hope your enthusiasm for OpenACC is growing. At least you can see how easy it is to take a stab at accelerating a code. The low risk should encourage you to attempt this with your applications.

4.6 Exercises

1. We noted that the `track_progress` routine introduces a penalty for the periodic array copies that it initiates. However, the output itself is only a small portion of the full array. Can you utilize the `data` directive's array-shaping options to minimize this superfluous copy (see Section 1.3.4)?

2. The sample problem is small by most measures. But it lends itself easily to scaling. How large a square plate problem can you run on your accelerator? Do so, and compare the speedup relative to the serial code for that case.

3. This code can also be scaled into a 3D version. What is the largest 3D cubic case you can accommodate on your accelerator?

4. We have focused only on the main loop. Could you also use OpenACC directives on the initialize and output routines? What kinds of gains would you expect?

5. If you know OpenMP, you may see an opportunity here to speed up the host (CPU) version of the code and improve the serial performance. Do so, and compare to the speedup achieved with OpenACC.

Index

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W

X