

# **OpenACC** For **Programmers**

**Concepts and Strategies** 



J.

in

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

£

## OpenACC<sup>™</sup> for Programmers

This page intentionally left blank

## OpenACC<sup>™</sup> for Programmers

### **Concepts and Strategies**

<sup>Edited by</sup> 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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

OpenACC is a trademark of NVIDIA Corporation.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned. com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2017945500

Copyright © 2018 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

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
Preface
Acknowledgments
About the Contributors
Chapter 1: OpenACC in a Nutshell
1.1 OpenACC Syntax
1.1.1 Directives
1.1.2 Clauses
1.1.3 API Routines and Environment Variables 5
1.2 Compute Constructs
1.2.1 Kernels
1.2.2 Parallel
1.2.3 Loop
1.2.4 Routine
1.3 The Data Environment
1.3.1 Data Directives
1.3.2 Data Clauses
1.3.3 The Cache Directive
1.3.4 Partial Data Transfers
1.4 Summary

1.5 Exercises	15
Chapter 2: Loop-Level Parallelism	17
2.1 Kernels Versus Parallel Loops	18
2.2 Three Levels of Parallelism	21
2.2.1 Gang, Worker, and Vector Clauses	22
2.2.2 Mapping Parallelism to Hardware	23
2.3 Other Loop Constructs	24
2.3.1 Loop Collapse	24
2.3.2 Independent Clause	25
2.3.3 Seq and Auto Clauses	27
2.3.4 Reduction Clause	28
2.4 Summary	30
2.5 Exercises	31
Chapter 3: Programming Tools for OpenACC	33
3.1 Common Characteristics of Architectures	34
3.2 Compiling OpenACC Code	35
3.3 Performance Analysis of OpenACC Applications	36
3.3.1 Performance Analysis Layers and Terminology	37
3.3.2 Performance Data Acquisition	38
3.3.3 Performance Data Recording and Presentation	39
3.3.4 The OpenACC Profiling Interface	39
3.3.5 Performance Tools with OpenACC Support	41
3.3.6 The NVIDIA Profiler	41
3.3.7 The Score-P Tools Infrastructure for Hybrid Applications	44
3.3.8 TAU Performance System	48

#### CONTENTS

3.4 Identifying Bugs in OpenACC Programs	51
3.5 Summary 5	53
3.6 Exercises	54
Chapter 4: Using OpenACC for Your First Program 5	9
4.1 Case Study	59
4.1.1 Serial Code	51
4.1.2 Compiling the Code	57
4.2 Creating a Naive Parallel Version	58
4.2.1 Find the Hot Spot	58
4.2.2 Is It Safe to Use kernels?	59
4.2.3 OpenACC Implementations	59
4.3 Performance of OpenACC Programs	1
4.4 An Optimized Parallel Version	'3
4.4.1 Reducing Data Movement	13
4.4.2 Extra Clever Tweaks	'5
4.4.3 Final Result	76
4.5 Summary	78
4.6 Exercises	'9
Chapter 5: Compiling OpenACC	1
5.1 The Challenges of Parallelism	32
5.1.1 Parallel Hardware	32
5.1.2 Mapping Loops	33
5.1.3 Memory Hierarchy	35
5.1.4 Reductions	36
5.1.5 OpenACC for Parallelism	37

5.2 Restructuring Compilers 88
5.2.1 What Compilers Can Do
5.2.2 What Compilers Can't Do
5.3 Compiling OpenACC 92
5.3.1 Code Preparation
5.3.2 Scheduling
5.3.3 Serial Code
5.3.4 User Errors
5.4 Summary
5.5 Exercises
Chapter 6: Best Programming Practices
6.1 General Guidelines
6.1.1 Maximizing On-Device Computation
6.1.2 Optimizing Data Locality
6.2 Maximize On-Device Compute
6.2.1 Atomic Operations
6.2.2 Kernels and Parallel Constructs
6.2.3 Runtime Tuning and the If Clause
6.3 Optimize Data Locality
6.3.1 Minimum Data Transfer
6.3.2 Data Reuse and the Present Clause
6.3.3 Unstructured Data Lifetimes
6.3.4 Array Shaping 111
6.4 A Representative Example
6.4.1 Background: Thermodynamic Tables
6.4.2 Baseline CPU Implementation

6.4.3 Profiling	113
6.4.4 Acceleration with OpenACC	114
6.4.5 Optimized Data Locality	116
6.4.6 Performance Study	117
6.5 Summary	118
6.6 Exercises	119
Chapter 7: OpenACC and Performance Portability	. 121
7.1 Challenges	121
7.2 Target Architectures	123
7.2.1 Compiling for Specific Platforms	123
7.2.2 x86_64 Multicore and NVIDIA	123
7.3 OpenACC for Performance Portability	124
7.3.1 The OpenACC Memory Model	124
7.3.2 Memory Architectures	125
7.3.3 Code Generation	125
7.3.4 Data Layout for Performance Portability	126
7.4 Code Refactoring for Performance Portability	126
7.4.1 HACCmk	127
7.4.2 Targeting Multiple Architectures	128
7.4.3 OpenACC over NVIDIA K20x GPU	130
7.4.4 OpenACC over AMD Bulldozer Multicore	130
7.5 Summary	132
7.6 Exercises	133
Chapter 8: Additional Approaches to Parallel Programming	135
8.1 Programming Models	135

8.1.1 OpenACC
8.1.2 OpenMP
8.1.3 CUDA
8.1.4 OpenCL
8.1.5 C++ AMP
8.1.6 Kokkos
8.1.7 RAJA
8.1.8 Threading Building Blocks
8.1.9 C++17
8.1.10 Fortran 2008
8.2 Programming Model Components
8.2.1 Parallel Loops
8.2.2 Parallel Reductions
8.2.3 Tightly Nested Loops
8.2.4 Hierarchical Parallelism (Non-Tightly Nested Loops)
8.2.5 Task Parallelism
8.2.6 Data Allocation
8.2.7 Data Transfers
8.3 A Case Study
8.3.1 Serial Implementation
8.3.2 The OpenACC Implementation
8.3.3 The OpenMP Implementation
8.3.4 The CUDA Implementation
8.3.5 The Kokkos Implementation
8.3.6 The TBB Implementation
8.3.7 Some Performance Numbers

8.4 Summary
8.5 Exercises
Chapter 9: OpenACC and Interoperability
9.1 Calling Native Device Code from OpenACC
9.1.1 Example: Image Filtering Using DFTs
9.1.2 The host_data Directive and the use_device Clause
9.1.3 API Routines for Target Platforms
9.2 Calling OpenACC from Native Device Code
9.3 Advanced Interoperability Topics
9.3.1 acc_map_data
9.3.2 Calling CUDA Device Routines from OpenACC Kernels 184
9.4 Summary
9.5 Exercises
Chapter 10: Advanced OpenACC 187
Chapter 10: Advanced OpenACC       187         10.1 Asynchronous Operations       187
Chapter 10: Advanced OpenACC         187           10.1 Asynchronous Operations         187           10.1.1 Asynchronous OpenACC Programming         190
Chapter 10: Advanced OpenACC       187         10.1 Asynchronous Operations       187         10.1.1 Asynchronous OpenACC Programming       190         10.1.2 Software Pipelining       195
Chapter 10: Advanced OpenACC       187         10.1 Asynchronous Operations       187         10.1.1 Asynchronous OpenACC Programming       190         10.1.2 Software Pipelining       195         10.2 Multidevice Programming       204
Chapter 10: Advanced OpenACC       187         10.1 Asynchronous Operations       187         10.1.1 Asynchronous OpenACC Programming       190         10.1.2 Software Pipelining       195         10.2 Multidevice Programming       204
Chapter 10: Advanced OpenACC18710.1 Asynchronous Operations18710.1.1 Asynchronous OpenACC Programming19010.1.2 Software Pipelining19510.2 Multidevice Programming20410.2.1 Multidevice Pipeline20410.2.2 OpenACC and MPI208
Chapter 10: Advanced OpenACC18710.1 Asynchronous Operations18710.1.1 Asynchronous OpenACC Programming19010.1.2 Software Pipelining19510.2 Multidevice Programming20410.2.1 Multidevice Pipeline20410.2.2 OpenACC and MPI20810.3 Summary213
Chapter 10: Advanced OpenACC       187         10.1 Asynchronous Operations       187         10.1.1 Asynchronous OpenACC Programming       190         10.1.2 Software Pipelining       195         10.2 Multidevice Programming       204         10.2.1 Multidevice Pipeline       204         10.2.2 OpenACC and MPI       208         10.3 Summary       213         10.4 Exercises       213
Chapter 10: Advanced OpenACC18710.1 Asynchronous Operations18710.1.1 Asynchronous OpenACC Programming19010.1.2 Software Pipelining19510.2 Multidevice Programming20410.2.1 Multidevice Pipeline20410.2.2 OpenACC and MPI20810.3 Summary21310.4 Exercises213Chapter 11: Innovative Research Ideas Using OpenACC, Part I
Chapter 10: Advanced OpenACC       187         10.1 Asynchronous Operations       187         10.1.1 Asynchronous OpenACC Programming       190         10.1.2 Software Pipelining       195         10.2 Multidevice Programming       204         10.2.1 Multidevice Pipeline       204         10.2.2 OpenACC and MPI       208         10.3 Summary       213         10.4 Exercises       213         10.4 Exercises       213         11.1 Sunway OpenACC       215

#### CONTENTS

11.1.2	The Memory Model in the Sunway TaihuLight
11.1.3	The Execution Model
11.1.4	Data Management
11.1.5	Summary
11.2 Com	oiler Transformation of Nested Loops for Accelerators
11.2.1	The OpenUH Compiler Infrastructure
11.2.2	Loop-Scheduling Transformation
11.2.3	Performance Evaluation of Loop Scheduling
11.2.4	Other Research Topics in OpenUH
Chapter 12	2: Innovative Research Ideas Using OpenACC, Part II 237
Reco	nfigurable Computing
12.1.1	Introduction
12.1.2	Baseline Translation of OpenACC-to-FPGA
12.1.3	OpenACC Extensions and Optimization for Efficient FPGA Programming
12.1.4	Evaluation
12.1.5	Summary
12.2 Prog	ramming Accelerated Clusters Using XcalableACC
12.2.1	Introduction to XcalableMP
12.2.2	XcalableACC: XcalableMP Meets OpenACC
12.2.3	Omni Compiler Implementation
12.2.4	Performance Evaluation on HA-PACS
12.2.5	Summary
Index	

## 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 prod-ucts, 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, seguential 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<sup>™</sup>* 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 kickstart 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<sup>™</sup>* 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 Para-Tools, 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 heat-ing 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.

#### Here it is in Fortran:

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++) {</pre>
        for(j = 1; j <= WIDTH; j++) {</pre>
            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++) {</pre>
        for(j = 1; j <= WIDTH; j++) {</pre>
            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
               1000
#define HEIGHT
#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++) {</pre>
      for(j = 1; j <= WIDTH; j++) {</pre>
        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++) {</pre>
      for(j = 1; j <= WIDTH; j++) {</pre>
         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++) {</pre>
    for (j = 0; j <= WIDTH+1; j++) {</pre>
      Temperature previous[i][j] = 0.0;
    }
  }
  for(i = 0; i <= HEIGHT+1; i++) {</pre>
    Temperature previous[i][0] = 0.0;
    Temperature previous[i][WIDTH+1] = (100.0/HEIGHT)*i;
  }
  for(j = 0; j <= WIDTH+1; j++) {</pre>
    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++) {</pre>
    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
                          :: height=1000
 integer, parameter
 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
                                :: height=1000
  integer, parameter
  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++) {</pre>
      for(j = 1; j <= WIDTH; j++) {</pre>
         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++) {</pre>
      for(j = 1; j <= WIDTH; j++) {</pre>
         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);
   1
   iteration++;
```

Listing 4.6 Fortran Laplace code main loop with kernels directives

```
do while ( worst dt > temp tolerance )
   !$acc kernels
   do j=1, width
      do i=1, height
         temperature(i,j) =0.25*(temperature previous(i+1,j)&
                                + temperature previous(i-1,j)&
                                + temperature previous(i,j+1)&
                                + temperature previous (i, j-1))
      enddo
   enddo
   !Sacc end kernels
  worst dt=0.0
   !$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 kernelsenabled 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:

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
```

Listing 4.8 Final Fortran OpenACC Laplace code main loop

```
!$acc data copy(temperature previous), create(temperature)
do while ( worst dt > temp \overline{t} olerance )
   !$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 (\overline{i}, j) = \text{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.

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

- 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.

80

## Index

### A

Abstraction abstract thinking required for parallelism, 81 in C++, 137 in Kokkos, 141 TBB as C++ abstraction layer, 141 acc async noval, 191 acc async sync, 181, 191 acc device, 5, 124 acc device default, 206 acc device nvidia, 206 acc deviceptr, 13, 179, 182 acc get cuda stream, 180-181, 194 acc get device type, 205-206 acc hosteptr, 180 acc map data, 182-184 ACC MULTICORE environment variable, 129-130 acc notify, 52, 56 ACC NUM CORES=8 environment flag, 130 acc set cuda stream, 180-181, 195 acc set device, 206-207 acc set device num, 205-206, 209, 212 acc set device type, 124 Accelerators affinity, 209 architectural characteristics, 34 calling OpenACC from native device code, 182 compiler transformation of nested loops for, 224 computational fluid dynamics case study, 114-116 internode communication, 259 multidevice programming, 204 **OpenACC support**, 177 PGI compiler generating executable for, 123 programming accelerated clusters. See XcalableACC (XACC)

Sunway Taihulight memory model, 217-218 tightly coupled, 261–262 Advanced Institute for Computational Science (AICS). See XcalableACC (XACC) Affine memories, of accelerator, 124 Affinity, accelerator devices, 209 AICS (Advanced Institute for Computational Science). See XcalableACC (XACC) Aliasing, compiler limitations, 90-91 align directive, in XMP, 255-256 allgather clause, internode communication in XMP, 256-257 Allinea DDT, for debugging, 52–53 Allocations. See Data allocations Altera Offline compiler (AOC)<sup>2</sup>, use as backend compiler, 242 Altera Stratix V GS D5 FPGA, 248 Amdahl's law, 103 AOC<sup>2</sup> (Altera Offline compiler), use as backend compiler, 242 **API** routines overview of. 5 for target platforms, 180-181 Applications analysis of OpenACC applications, 36-37 analyzing performance, 196–198 analyzing program performance (Laplace Solver). 71-73 creating program and applying OpenACC to it, 59-61 HeteroIR code of MM program, 240-241 viewing runtime behavior of, 36 viewing temporal evolution of a program, 39 Architectures common characteristics, 34 portability, 123-124 targeting multiple, 128-130

Arithmetic units, coordinating functional units of hardware, 82-83 Arravs data clauses. 12–13 data layout for performance portability, 126 iterating over multidimensional, 17 optimization of data locality, 111-112 programming with XACC directives, 258–259 reducing to scalar, 86-87 shaping, 18 async clause acc async noval, 191 acc async sync, 181, 191 adding directives to work gueues, 191 making pipelining operation asynchronous, 202-204 Asynchronous operations adding directives to work gueues, 191 advanced OpenACC options, 187–190 making pipelining operation asynchronous, 201–204 Asynchronous programming asynchronous work queues, 190-191 defined, 190 interoperating with CUDA streams, 194–195 joining work queues, 193–194 overview of, 190 wait directive. 191-192 Asynchronous work queues advanced OpenACC options, 190-191 interoperability with OpenACC, 194-195 atomic directive for atomic operations, 105–106 types of data management directives, 4 Atomic operations, maximize on-device computation, 105-106 auto clause, in loop parallelization, 27-28 Auxiliary induction variable substitution, compiling OpenACC, 93 AXPBY (vector addition) CUDA implementation of MiniFE, 159 OpenACC implementation of MiniFE, 157 OpenMP implementation of MiniFE, 158 serial implementation of MiniFE, 156 TBB implementation of MiniFE, 165

### В

Backslash (\), in directive syntax, 3 Bakery counter, dynamic scheduling of workers, 94–95

Baseline CPU implementation, in CFD case study, 113 Baseline profiling analyzing application performance, 196–198 of asynchronous pipelining operation, 203–204 as best practice, 101 of translation of OpenACC to FPGAs, 239-243 bcast directive. communication directives. 259 Benchmarks. See also Baseline profiling evaluating loop scheduling performance of OpenUH, 231 evaluating OpenACC translation to FPGAs, 248 evaluating XACC performance on HA-PACS using Himeno, 262-264 evaluating XACC performance on HA-PACS using NPB-CG, 264-267 research topics in OpenUH, 234 Best practices, programming applied to thermodynamic fluid property table, 112-118 general guidelines, 102–105 maximize on-device computation, 105-108 optimize data locality, 108–112 overview of. 101–102 summary and exercises, 118-119 bisection function. in CFD solver. 113-114 Block recycling, CUDA implementation of MiniFE, 161 Blocks, of code blocking data movement, 200-201 blocking the computation, 198–199 Bottlenecks, detecting, 37 Boundary conditions, 60 Branches, removing from code section, 95 Bugs identifying, 51-53 user errors in compiling OpenACC, 95–97 Bulldozer multicore running OpenACC over, 130–131 targeting multiple architectures, 129–130

### С

C++ AMP comparing programming models, 136, 143 data allocations, 153 features, 140 mapping simple loop to parallel loop, 145 tightly nested loops, 148 C/C++ abstractions and templates, 137 array shape specification, 112 asvnc clause example, 191 compiling code for Laplace Solver, 70 creating naive parallel version of Laplace Solver, 69 OpenACC built on top of, 1, 35 optimized version of Laplace Solver, 76-77 using Jacobi iteration to locate Laplace condition, 61-65 wait directive examples, 191-194 XMP code example, 254 C++11, 151-152 C++17 comparing programming models, 136 concept coverage list in comparing programming models, 143 mapping simple loop to parallel loop, 145 programming features, 142 C2R (complex-to-real). Discrete Fourier Transform, 176 cache directive overview of, 13-14 types of data management directives, 4 Cache, Sunway Taihulight data management, 221 Call-graph profiles, 39, 102-103 Call-path profiles, 39 CFD solver case study. See Computational fluid dynamics (CFD) solver, case study CG solve. See Conjugate gradient solver (CG solve) Chief processors, 94-95 Clauses. See also by individual types categories of, 4-5 data clauses, 12-13 Code advantages of OpenACC code, 79 blocking data movement, 200-201 blocking the computation, 198–199 calling native device code from OpenACC, 174-181 calling OpenACC from native device code, 181-182 compiling code for Laplace Solver, 67–68 creating naive parallel versions, 68-71 creating serial code for Laplace Solver, 61-67 portability, 125-126 preparation for compiling OpenACC, 92–93

removing all branches in section of, 95 Code editors, spotting syntax errors, 33 collapse keyword. loop directive. 24-25 Complex-to-real (C2R). Discrete Fourier Transform, 176 Compatibility. See Interoperability Compilers compiler transformation of nested loops for accelerators, 224 compiling code for Laplace Solver, 67–68, 70–71 compiling code for specific platforms, 123 compiling optimized version of Laplace Solver, 78 directives. 3 identifying bugs, 52 OpenACC supported, 35-36 OpenACC-to-FPGA translation, 242-243 OpenUH. See OpenUH compiler runtime implementation of XACC, 260–262 viewing runtime behavior of applications, 36 what compilers can do, 88-90 what compilers cannot do. 90–91 Compiling OpenACC applying OpenACC for parallelism, 87-88 challenges of parallelism, 82 code preparation for, 92-93 coordinating functional units of hardware, 82-83 handling reductions, 86-87 mapping loops, 83-85 memory hierarchy, 85-86 overview of, 81 scheduling, 93-94 serial code, 94–95 summary and exercises, 97–99 user errors, 95-97 what compilers can do, 88-90 what compilers cannot do, 90–91 Complexity, benefits of computers, 81 Components, parallel programming, 142–143 Computation blocking, 198–199 maximization of on-device computation, 103 offloading, 34 Computational fluid dynamics (CFD) solver, case study acceleration with OpenACC, 114-116 baseline CPU implementation, 113 optimized data locality, 116-117 performance study, 117–118

Computational fluid dynamics (CFD) solver, case study (continued) profiling, 113–114 thermodynamic tables, 112-113 Compute constructs directive extension for compute unit replication, 243-245 evaluating OpenACC translation to FPGAs, 250 kernels directive, 6-7 loop directive, 8-9 overview of, 4, 6 parallel directive, 8 routine directive, 9-11 Computers. See also Supercomputers, 81 Computing processing element (CPE) Sunway Taihulight data management, 219 Sunway Taihulight execution model, 218–219 Sunway Taihulight memory model, 217-218 in SW26010 manycore CPU, 216-217 Conditions, initial and boundary, 60 Conjugate gradient solver (CG solve) CUDA implementation of MiniFE, 159 implementation of MiniFE, 163 **OpenMP** implementation of MiniFE, 158 overview of, 155 performance comparisons for MiniFE case. 168-169 temporary vectors required for MiniFE, 156 Constructs, OpenACC, 3 Control flow clauses checks on user errors, 95 clause categories, 5 copy clause, data clauses, 13, 76, 220, 260 copyin clause, data clauses, 13, 124 copyout clause, data clauses, 124 CORAL benchmark suite, 127–128 CPE. See Computing processing element (CPE) CPUs. See also Processors assigning MPI rank to, 209-210 baseline implementation, 113 data layout for performance portability, 126 multicore parallelizations, 127 SW26010 manycore CPU, 216-217 Cray compiler compiling for specific platforms, 123 OpenACC support, 35–36 create clause, data clauses, 12-13, 76

#### CUDA

calling CUDA routines from OpenACC, 184-185 comparing programming models, 136, 143 data allocations, 153 data transfers, 154 evaluating loop scheduling performance of OpenUH, 231 evaluating performance of MiniFE case, 167 Fast Fourier Transform (FTT) library, 174 features, 139 hierarchical parallelism (nontightly nested loops), 150 interoperability with OpenACC, 194–195 mapping OpenACC terminology to, 228-229 mapping simple loop to parallel loop, 144 MiniFE solver case study, 159–162 OpenARC support, 242 OpenCL compared with, 139–140 programming NVIDIA GPUs with, 261 streams, 180-181 tightly nested loops, 148 translating OpenACC offload region into CUDA code, 225

### D

Data acquiring performance data, 38-39 blocking data movement, 200-201 clauses, 4, 12-13 events, 40 managing in Sunway Taihulight, 219–222 optimizing locality. See Optimization of data locality portability, 126 recording and presenting performance data, 39 XACC data distribution, 255-256 Data allocations alignment and, 179-180 comparing parallel programming models, 152-153 runtime awareness of, 182 data directive data copy, 76, 220, 260 data create,76 data distribute. 255-256 enter/exit directives, 4, 111 Laplace code performance, 78 for managing data, 4

reducing data movement, 73-75 shared memory systems and, 124 types of data clauses, 12–13 XACC memory model, 257 Data environment cache directive, 13-14 data clauses. 12–13 data directive. 12 overview of. 11 Data-flow analysis, scalar precursor of dependency, 89-90 Data lifetimes data environment concepts. 11 for unstructured data. 111 Data parallelism. See also Parallelism comparing programming models, 136 defined, 188 Data regions creating optimized parallel version of Laplace Solver. 73-78 data environment concepts. 11 in OpenACC memory model, 124 structured and unstructured, 12 Data reuse maximizing, 103 present clause and, 110-111 Data transfer comparing parallel programming models, 153–155 minimizing, 103-104, 109-110 OpenMP implementation of MiniFE requiring, 158-159 runtime implementation of XACC, 261 Sunway Taihulight data management, 219, 221-222 DDT, debugging using Allinea DDT, 52-53 Debugging, 51-53 declare clause, data clauses. 4 delete clause. data clauses. 13 Dependencies asynchronous work queues exposing, 190 comparing dependent and independent tasks, 189-190 operations as series of, 188 what compilers can do, 89–90 Descriptive directives, vs. prescriptive, 96-97 device clauses acc device, 5, 124 acc device default, 206

acc device nvidia, 206 acc deviceptr, 13, 179, 182 Devices management functions of API routines, 5 maximization of on-device computation, 103 DFT (Discrete Fourier Transform), 174–177 Direct memory access (DMA) moving data between memories, 124 MPI with. 211-213 MPI without, 210-211 Sunway Taihulight data management, 220 in SW26010 manycore CPU, 216-217 Directive-based high-performance reconfigurable computing baseline translation of OpenACC to FPGAs, 239 - 243evaluating OpenACC translation to FPGAs, 248-252 OpenACC extensions and optimizations for FPGAs. 243-247 overview of. 237-239 summary of OpenACC translation to FPGAs, 252 Directive-based programming models, 1 Directives. See also by individual types comparing kernels with parallel, 18-21 compilers and, 35 compiling OpenACC, 92 efficiency of, 96 internode communication in XMP, 256–257 OpenACC syntax, 3 prescriptive vs. descriptive, 96 programming with XACC directives, 258-259 types of, 3-4 Discrete Fourier Transform (DFT), 174–177 Discrete memories, types of system memory, 125 distribute directive, data distribution and work mapping in XMP, 255–256 Divergence, checks on user errors, 95 DMA. See Direct memory access (DMA) Dot product CUDA implementation of MiniFE, 159 OpenACC implementation of MiniFE, 157 reduction example, 86-87 serial implementation of MiniFE, 156 TBB implementation of MiniFE, 166 Dynamic scheduling scheduling parallel and vector code, 94 of workers using bakery counter, 94-95

### Ε

EBS (Event-based sampling), supported TAU performance system, 49-50 Efficiency advantages of OpenACC code, 79 directive strategy and, 96 enqueue, events indicating runtime of device tasks. 41 enter data directive data lifetimes and. 111 types of data management directives, 4 **Environment variables** ACC MULTICORE, 129-130 **OpenACC** specification. 3 overview of, 5 Errors, user errors in compiling OpenACC, 95–97 Event-based instruments, data acquisition for performance analysis, 38-39 Event-based sampling (EBS), supported TAU performance system, 49-50 Event callbacks, 40 Event categories, 40 Exascale systems. See also Supercomputers criteria requirements, 238 portability as goal in, 122 Execution model Sunway Taihulight, 218-219 XMP. 255 Execution policy, C++17, 142 exit data directive data lifetimes and. 111 types of data management directives, 4 Explicit parallelism, XMP support for, 256 Expressions storing expression value in temporary, 85 symbolic subscript expressions, 90 Extensions C++ AMP extension of C++, 136, 140 OpenACC extensions and optimizations for FPGAs, 243-247 programming languages, 137

### F

Fast Fourier Transforms (FTTs), 174–177 Field-programmable gate arrays (FPGAs) baseline translation of OpenACC to, 239–243 evaluating OpenACC translation to, 248–252 in high-performance computing, 237–238

OpenACC extensions and optimizations for, 243-247 summary of OpenACC translation to, 252 FIFO (first-in, first out), work queues, 190 Filtering images, using Discrete Fourier Transform, 174-177 First-class concepts implementation of MiniFE, 164 parallel loops, 143 First-in, first out (FIFO), work gueues, 190 firstprivate, variables, 11, 88 Flags, for OpenACC compiler types, 35 Flat function profiles, 39, 102 Floating-point operations (FLOPS), for computebound kernels, 123 Fortran array shape specification, 112 comparing programming models, 136, 143 compiling code for Laplace Solver, 70-71 creating naive parallel version of Laplace Solver, 70 mapping simple loop to parallel loop, 145 OpenACC built on top of, 35 optimized version of Laplace Solver, 77 programming features in Fortran 2008, 142 using Jacobi iteration to locate Laplace condition, 61-63, 66-67 FPGAs. See Field-programmable gate arrays (FPGAs) FFTs (Fast Fourier Transforms), 174–177 Functional units, coordinating functional units of hardware, 82-83

### G

gang clause levels of parallelism, 21–22 mapping parallelism to hardware, 23–24 overview of, 22–23 Gangs applying OpenACC for parallelism, 87–88 distributing iterations across, 125–126 principles governing loop performance, 96 scheduling parallel loops, 227–230 GCC<sup>2</sup>, OpenACC support, 36 get clause acc\_get\_cuda\_stream, 180–181, 194 acc\_get\_device\_type, 205–206 Ghost zone, defined, 200

Global view, XMP, 254 gmove directive, internode communication in XMP. 257 GNU compiler, 123 GPUDirect, runtime implementation of XACC, 261-262 GPUs assigning MPI rank to, 209-210 CUDA support for NVIDIA GPUs, 139 demonstrating portability using OpenACC, 123 evaluating loop scheduling performance of OpenUH, 230 layout for performance portability, 126 mapping parallelism to hardware, 23-24 running OpenACC over NVIDIA K20X GPU, 130-131 targeting multiple architectures, 128-130

### Η

HA-PACS evaluating XACC performance on. 262–267 runtime implementation of XACC, 261 HACCmk microkernel OpenACC programming model for, 122 overview of, 127-128 targeting multiple architectures, 128–130 Halo, 200 Hardware coordinating functional units of, 82-83 mapping loops onto parallel hardware, 83-85 mapping parallelism to, 23-24 Hardware description language (HDLs), 239 HeteroIR. 240-241 Hierarchical parallelism (nontightly nested loops), 148-151 High-performance computing (HPC) field-programmable gate arrays in, 237 framework for directive-based. See Directivebased high-performance reconfigurable computing MPI in, 208-209 parallel programming models and, 135 portability as goal in, 121 Himeno, evaluating XACC performance on HA-PACS, 262-264 host acc hosteptr, 180 host data directive, 177-180, 211-212

Hot spot finding, 68–69 identifying, 102–103 HPC. *See* High-performance computing (HPC)

### I

IC (integrated circuit), in FPGAs, 239 IDE (integrated development environment), spotting syntax errors, 33 if clause, maximize on-device computation. 107-108 If conversion, removing all branches in section of code. 95 Images, filtering using Discrete Fourier Transform, 174-177 Incremental acceleration and verification, as best practice, 101, 104 Independence, comparing dependent and independent tasks, 189–190 independent clause, adding to loop directive, 25 - 27Initial conditions, 60 Initialization, functions of API routines, 5 Innovation/research data management in Sunway Taihulight, 219-222 evaluating loop scheduling performance of OpenUH, 230-234 execution model in Sunway Taihulight, 218–219 framework for directive-based HPC. See Directive-based high-performance reconfigurable computing loop-scheduling transformation in OpenUH, 226-230 memory model in Sunway Taihulight, 217–218 OpenUH compiler infrastructure, 224–225 overview of, 215 programming accelerated clusters. See XcalableACC (XACC) research topics related to OpenUH, 234–235 summary of Sunway system, 223 Sunway Taihulight, 215–216 SW26010 manycore CPU, 216-217 Instructions, coordinating functional units of hardware, 82-83 int bisection function. in CFD solver. 113-114 int main function. in CFD solver. 113-114 Integrated circuit (IC), in FPGAs, 239

Integrated development environment (IDE), spotting syntax errors, 33 Intermediate representation (IR) HeteroIR. 240 **OpenUH** infrastructure, 224 Internode communication, XMP, 256-257, 259 Interoperability advanced topics, 182–185 calling native device code from OpenACC, 174-181 calling OpenACC from native device code, 181-182 with CUDA streams, 194–195 overview of, 173 summary and exercises, 185–186 interpolate method, LookupTable2D class, 113-114 Interthread block parallelism, in CUDA, 139 IR (intermediate representation) HeteroIR. 240 OpenUH infrastructure, 224 Iteration. See Loops

### J

Jacobi iteration evaluating XACC performance on HA-PACS, 262 locating Laplace condition, 61–67 solving Laplace equation for steady-state temperature distribution, 60

### K

Kernel configuration bound check elimination, OpenACC extensions and optimizations for FPGAs. 243-244 Kernel launch events, 40 Kernel vectorization, OpenACC extensions and optimizations for FPGAs, 243-244 kernels directive analyzing program performance (Laplace Solver), 72–73 applying to OpenACC case study (Laplace Solver), 69-71 calling CUDA device routines, 184–185 evaluating OpenACC translation to FPGAs, 251 extension for kernel vectorization. 244-245 kernel loop scheduling, 228-230 kernel-pipelining transformation, 245-247 Laplace code performance, 78

mapping parallel regions to hardware, 24 maximize on-device computation, 106-107 overview of. 6-7 parallel directive compared with. 18-21 reduction clause, 28-30 types of compute directives. 4 Knights Landing (KNL), 123–124 comparing programming models, 136, 143 data allocations, 153 data layout for performance portability, 126 data transfers, 155 features. 140-141 hierarchical parallelism (nontightly nested loops), 151 mapping simple loop to parallel loop, 145 MiniFE solver case study, 163-165 parallel reductions, 146 performance comparisons for MiniFE case, 167 task parallelism, 151–152 tightly nested loops, 148

### L

Languages. See Programming languages Laplace Solver case study analyzing program performance, 71–73 compiling code, 67-68, 70-71 creating naive parallel versions, 68-71 creating optimized parallel version, 73–78 creating program and applying OpenACC to it, 59-61 evaluating loop scheduling performance of OpenUH, 231-233 solving Laplace equation for steady-state temperature distribution, 60 summary and exercises, 78-80 using Jacobi iteration to locate Laplace condition. 61-67 Libraries passing device pointers to host libraries, 211-212 routines in OpenACC specification, 3 Local view, XMP, 254 Locality awareness, research topics in OpenUH, 234 Locality of data, optimizing. See Optimization of data locality LookupTable2D class, 113-114 loop directive adding auto clause to, 27-28

adding independent clause to, 25-27 adding seg clause to, 27 in code generation, 125 collapse keyword. 24-25 combining with parallel for parallelization, 20 data distribution and work mapping in XMP, 255-256 executing loops on MIMD hardware, 87–88 internode communication in XMP. 256-257 levels of parallelism, 21–22 overview of. 8-9 reduction clause. 28-30 runtime implementation of XACC, 260 types of compute directives, 4 work sharing in XMP, 258 Loop parallelization. See also Parallel loops; Parallelism collapse keyword, 24-25 independent clause, 25-27 kernels vs. parallel loops, 18-21 levels of parallelism, 21-23 loop construct options, 24 mapping parallelism to hardware, 23-24 overview of. 17-18 reduction clause. 28-30 seq and auto clauses, 27-28 summary and exercises, 30-31 Loop unrolling, 243-244, 250 Loops applying to CFD solver. See Computational fluid dynamics (CFD) solver, case study creating optimized parallel version of Laplace Solver. 75-78 distributing iterations across gangs, workers, or vectors, 125-126 evaluating loop scheduling performance of OpenUH, 230-234 extension for loop unrolling, 244 loop-scheduling transformation in OpenUH, 226-230 mapping to parallel hardware, 83–85 nontightly nested loops (hierarchical parallelism), 148-151 parallel loops, 143–145 principles governing performance of, 96 symbolic loop bounds and steps creating issues for compilers, 91 tightly nested loops, 147

using Jacobi iteration to locate Laplace condition, 61–67

#### Μ

Management processing element (MPE) Sunway Taihulight execution model, 218-219 Sunway Taihulight memory model, 217–218 in SW26010 manycore CPU, 216-217 map, acc map data, 182-184 Mapping loops onto parallel hardware, 83-85 OpenACC terminology to CUDA, 228-229 parallelism to hardware, 23-24 simple loop to parallel loop, 144–145 work in XACC, 255-256 Matrix multiplication (MM) evaluating loop scheduling performance of OpenUH, 231-233 in OpenACC, 240 Maximize on-device computation atomic operations, 105–106 as best practice, 101 kernels and parallel constructs, 106-107 overview of, 103 runtime tuning and if clause, 107–108 Memory hierarchy in compiling OpenACC, 85–86 management functions of API routines, 5 Memory models portability, 124-125 Sunway Taihulight, 217–218 XACC. 257 Message Passing Interface (MPI) combining OpenACC with, 187 with direct memory access, 211–213 interprocess communication, 37 overview of, 208-210 runtime implementation of XACC, 261–264 without direct memory access, 210-211 MIMD. See Multiple-instruction multiple data (MIMD) MiniFE solver case study CUDA implementation, 159–162 implementation, 163-165 **OpenACC** implementation, 157–158 OpenMP implementation, 158–159 overview of, 155 performance comparisons, 167–169

MiniFE solver case study (continued) serial implementation, 156-157 TBB implementation, 165–167 MM (Matrix multiplication) evaluating loop scheduling performance of OpenUH, 231-233 in OpenACC, 240 MPE. See Management processing element (MPE) MPI. See Message Passing Interface (MPI) Multicore systems ACC MULTICORE environment variable, 129-130 OpenACC programming model for, 122 Multidevice programming MPI and, 208-210 MPI with direct memory access, 211–213 MPI without direct memory access, 210-211 multidevice pipeline, 204-208 overview of, 204 Multigrid, in performance study of CFD solver. See also Processors, 117 Multiple-instruction multiple data (MIMD) coordinating functional units of hardware, 83 executing loops on MIMD hardware, 87 privatization and, 86 sequential loops and, 84-85 Multithreading, performance analysis, 37

### Ν

NAS parallel benchmarks, evaluating XACC performance on HA-PACS, 264–267 Nested loops compiler transformation for accelerators, 224 iterating over multidimensional array, 17 nontightly nested loops (hierarchical parallelism), 148-151 offloading computation intensive, 226 tightly nested loops, 147 using Jacobi iteration to locate Laplace condition, 61-67 Non-uniform memory access (NUMA), 23-24 nontightly nested loops (hierarchical parallelism), 150 notify, acc notify, 52, 56 NPB-CG kernel, evaluating XACC performance on HA-PACS. 264-267 num, ACC NUM CORES=8 environment flag, 130 NUMA (non-uniform memory access), 23-24

NVIDIA GPUs assigning MPI rank to, 210 CUDA support, 139 demonstrating portability using OpenACC, 123–124 interoperating OpenACC asynchronous work queues with CUDA streams, 194–195 programming, 261 running OpenACC over NVIDIA K20X GPU, 130–131 targeting multiple architectures, 128–130 NVIDIA profilers, 40–43 nvprof, NVIDIA command-line profiler, 41–43 nvvp, NVIDIA Visual Profiler, 41–43

### 0

Oak Ridge Leadership Computing Facility (OLCF), 130 Oak Ridge National Laboratory (ORNL), 237 Offloading computation, 34, 226 performance analysis, 37 translating OpenACC offload region into CUDA code, 225 OLCF (Oak Ridge Leadership Computing Facility), 130 **OMNI** compiler runtime implementation of XACC, 260-262 as source-to-source compiler, 259-260 Open Accelerator Research Compiler (Open ARC) baseline translation of OpenACC to FPGAs, 240-241 FPGA prototype system built on, 238 OpenACC, advanced options async clause, 191 asynchronous operations, 187–190 asynchronous work queues, 190–191 blocking data movement, 200-201 blocking the computation, 198–199 interoperating with CUDA streams, 194–195 joining work queues, 193–194 making pipelining operation asynchronous, 201-204 MPI and, 208-210 MPI with direct memory access, 211-213 MPI without direct memory access, 210-211 multidevice pipeline, 204-208 multidevice programming, 205

overview of, 187 software pipelining, 195-198 summary and exercises, 213 wait clause. 191-192 OpenACC, comparing parallel programming languages concept coverage list, 143 data allocations, 153 data transfers, 154 features, 138 implementation of MiniFE solver, 157–158 mapping simple loop to parallel loop, 144 multidevice image-filtering code, 206-208 nontightly nested loops (hierarchical parallelism), 150 overview of, 136 parallel reductions, 146 performance comparisons for MiniFE case, 167 tightly nested loops, 147 **OpenACC**, specification basics API routines and environment variables. 5 cache directive. 13–14 clauses, 4-5 compute constructs. 6 data clauses. 12–13 data directives, 12 data environment. 11 directives, 3-4 kernels. 6-7 loop construct. 8–9 overview of, 1-2 parallel directive, 8 routine directive, 9-11 summary and exercises, 14-15 syntax, 3 OpenCL comparing programming models, 136 data allocations, 153 data transfers, 155 features, 139-140 mapping simple loop to parallel loop, 145 nontightly nested loops (hierarchical parallelism), 150 OpenARC support, 242 tightly nested loops, 148 OpenMP comparing programming models, 136 concept coverage list, 143

data allocations, 153 data transfers, 154 features, 138 implementation of MiniFE solver, 158-159 mapping simple loop to parallel loop, 144 multidevice image-filtering code. 206-208 nontightly nested loops (hierarchical parallelism), 150 parallel reductions, 146 performance comparisons for MiniFE case, 167 task parallelism, 151–152 tightly nested loops, 147 OpenUH compiler evaluating loop scheduling performance, 230 - 234infrastructure, 224-225 loop-scheduling transformation, 226-230 research topics, 234-235 Optimization advantages of OpenACC code, 79 compiling optimized version of Laplace Solver, 75-78 incremental optimization as advantage of OpenACC code, 78 Optimization of data locality array shaping, 111–112 as best practice, 101 computational fluid dynamics case study, 116-117 data lifetimes for unstructured data, 111 data reuse and present clause, 110-111 locality awareness research in OpenUH, 234 minimum data transfer, 109–110 overview of, 103-105, 108 ORNL (Oak Ridge National Laboratory), 237

### Ρ

Parallel loops (continued) comparing parallel programming models, 143-145 implementation of MiniFE, 163 loop-scheduling transformation in OpenUH, 226-227 making pipelining operation asynchronous, 202-204 runtime implementation of XACC. 260 TBB implementation of MiniFE, 165 work sharing in XMP, 258 Parallel programming C++ AMP extension, 140 C++17. 142 case study- MiniFE solver. See MiniFE solver case study components, 142-143 CUDA, 139 data allocations. 152–153 data transfers. 153-155 Fortran, 141 hierarchical parallelism (nontightly nested loops), 148–151 OpenACC, 138 OpenCL, 139-140 OpenMP, 138 overview of, 135 parallel loops, 143–145 parallel reductions, 145–147 programming models, 135–137 RAJA, 141 summary and exercises, 170-171 task parallelism, 151–152 Threading Building Blocks (TBB), 141 tightly nested loops, 147 Parallel reductions comparing parallel programming models, 145-147 implementation of MiniFE, 164 TBB implementation of MiniFE, 166–167 Parallelism abstract thinking required for, 81 applying OpenACC for, 87-88 challenges of, 82 effective parallelism, 92 functions of API routines. 5 gang, worker, and vector clauses, 22-23 kernels vs. parallel loops, 18–21

loop-level, 17-18 loop-scheduling transformation in OpenUH, 227-230 mapping to hardware, 23-24 maximize on-device computation, 106-107 multicore parallelizations, 127 principles governing loop performance, 96 scheduling parallel and vector code. 93–94 three levels of. 21-22 XMP support for explicit parallelism, 256 ParaProf profiler, 48-49, 51 Partially shared memories, types of system memory, 125 Partitioned global address space (PGAS) programming model, 142 PathScale compiler, 36 PDT. source analysis tool. 48 PEACH2, 261 Pen, holding processors in, 94–95 PerfExplorer, for profile data mining, 48 Performance acquiring performance data, 38–39 analysis layers and terminology, 37-38 analysis of OpenACC applications, 36–37 comparisons, 167-169 computational fluid dynamics (CFD) case study, 117-118 evaluating loop scheduling of OpenUH, 230-234 evaluating OpenACC translation to FPGAs, 248-252 evaluating XACC on HA-PACS, 262-267 NVIDIA profiler, 40-43 profiling interface, 39-40 recording and presenting data, 39 Score-P infrastructure, 44-48 TAU system, 48-51 tools supported by OpenACC, 40 PGAS (partitioned global address space) programming model, 142 PGI compiler analyzing application performance, 196–197 compiling code for Laplace Solver, 67–68, 70–71 compiling optimized version of Laplace Solver, 78 generating executable for accelerator platforms, 123 OpenACC support, 35–36 PGI ACC TIME environment variable, 36, 71-73

PGPProf profiler analyzing application performance, 196–198 making pipelining operation asynchronous. 203-204 timeline for multidevice software pipelined image filter, 199 Pipelining. See Software pipelining Portability advantages of OpenACC code, 79 challenges, 121–123 code generation for, 125-126 data lavout for. 126 HACCmk microkernel, 127–128 memory systems and, 124-125 of OpenACC, 1-2 overview of, 121 refactoring code for. 126 running OpenACC over Bulldozer multicore, 130-131 running OpenACC over NVIDIA K20X GPU, 130–131 summary and exercises. 132–134 targeting multiple architectures, 128–130 types of target architectures, 123–124 #pragma acc routine, 115 Prescriptive directives, vs. descriptive, 96 present clause data reuse, 110-111 optimization of data locality, 110-111 overview of, 13 printf, debugging, 51-52 private clause compiling OpenACC, 92 specifying scalar variables as private, 88 variables, 11 Privatization, simultaneous semantics and, 86 Procedures, uses of routine directive, 9-10 Process parallelization, performance analysis, 37 Processors baseline CPU implementation, 113 coordinating functional units of hardware, 82–83 holding all but chief processor in a pen, 94–95 mapping parallelism to hardware, 23–24 SW26010 manycore CPU, 216-217 Profiling analyzing application performance, 196-198 best practices, 102 computational fluid dynamics case study, 113-114

data recording via, 39 interface supported by OpenACC, 39-40 NVIDIA profiler, 40–43 Score-P performance infrastructure, 44-48 TAU performance system, 48–51 tools supported by OpenACC, 40 Program counters, 82–83 Programming asynchronous programming, 190 best practices. See Best practices, programming as series of steps, 187 Programming languages comparing capabilities of, 136 extensions, 137 XMP. See XcalableMP (XMP) Programming models C++ AMP extension. 140 C++17. 142 CUDA, 139 Fortran 2008, 142 OpenACC, 138 OPENCL, 139-140 **OPENMP. 138** overview of, 135-137 **RAJA**. 141 Threading Building Blocks (TBB), 141 XACC. See XcalableACC (XACC) Programming tools acquiring performance data, 38-39 architectural characteristics, 34 bug identification, 51-53 compilers, 35-36 NVIDIA profiler, 40–43 overview of, 33 performance analysis layers and terminology, 37 - 38performance analyzers, 36-37 profiling interface, 39-40 recording and presenting performance data, 39 Score-P performance infrastructure, 44-48 summary and exercises, 53-57 TAU performance system, 48–51 tools supported by OpenACC, 40 Programs. See Applications

### Q

Queues. See Work queues

### R

R2C (Real-to-complex), Discrete Fourier Transform, 176 RA IA comparing programming models, 136, 143 features, 141 mapping simple loop to parallel loop, 145 parallel reductions, 146 tightly nested loops, 148 Real-to-complex (R2C), Discrete Fourier Transform, 176 reduction clause adding to kernels, parallel, or loop directive, 28-30 compiling OpenACC, 92-93 internode communication in XMP. 256-257 Reductions communication directives, 259 parallel reductions. See Parallel reductions of vector or array to scalar, 86-87 Refactoring code, for portability, 126 reflect, communication directives, 259 Research. See Innovation/research Reuse, See Data reuse routine directive acc routine. 115 overview of, 9-11 types of compute directives, 4 Routines API routines. 5 API routines for target platforms, 180–181 calling CUDA device routines from OpenACC kernels, 184–185 identifying hot spots, 102-103 guerying/setting device type, 205-206 Runtime tuning, maximize on-device computation, 107-108

### S

Sampling data acquisition for performance analysis, 38 with TAU performance system, 48 Scalar expansion, simultaneous semantics and, 86 Scalars data-flow analysis as precursor of dependency, 89–90 reducing vector or array to, 86–87 specifying variables as private, 88 Scheduling dynamic scheduling of workers using bakery counter, 94-95 evaluating loop scheduling performance of OpenUH, 230-234 loop-scheduling transformation in OpenUH, 226-230 mapping loops, 83-85 parallel and vector code, 93-94 Score-P performance infrastructure, 44-48 Scratch pad memory (SPM) Sunway Taihulight data management, 219, 221 Sunway Taihulight execution model, 219 Sunway Taihulight memory model, 217–218 in SW26010 manycore CPU, 216-217 Semantics parallel hardware, 83-84 simultaneous, 84 seq clause adding to loop directive, 27-28 for sequential execution of loop, 9 Sequential loops adding seg clause to loop directive, 27-28 executina. 9 vs. simultaneous or parallel loops, 87 Serial code compiling OpenACC, 94-95 implementation of MiniFE solver, 156–157 using Jacobi iteration to locate Laplace condition. 61-67 Serialization, of tasks, 190 set clause acc set cuda stream, 180-181, 195 acc set device, 206-207 acc set device num, 205-206, 209, 212 acc set device type, 124 Shadow elements, arrays, 258 Shared memories C++17, 142 HACCmk microkernel, 127-128 OpenMP, 138 types of system memory, 125 Shut down, functions of API routines, 5 SIMD (Single-instruction multiple-data), 83 SIMT (Single-instruction multiple-thread), 83 Simultaneous semantics, 84 Single-instruction multiple-data (SIMD), 83 Single-instruction multiple-thread (SIMT), 83

Single-program multiple-data (SPMD), 255 Single source, advantages of OpenACC code, 79 SMs (Streaming multiprocessors), 83 Software pipelining evaluating OpenACC translation to FPGAs, 251 kernel-pipelining transformation, 245–247 making pipelining operation asynchronous, 201-204 multidevice pipeline, 204-208 overview of, 195-198 timeline for multidevice software pipelined image filter, 199 Source-to-source code translation. in XACC. 259-260 Sparse matrix vector multiplication (SPMV) CUDA implementation of MiniFE, 162 implementation of MiniFE, 164–165 OpenACC implementation of MiniFE, 157–158 OpenMP implementation of MiniFE, 159 performance comparisons for MiniFE case, 168 serial implementation of MiniFE, 156–157 SPM. See Scratch pad memory (SPM) SPMD (Single-program multiple-data), 255 SPMV. See Sparse matrix vector multiplication (SPMV) Static scheduling, 94 Storage model, Sunway Taihulight memory model, 218 Streaming multiprocessors (SMs), 83 Streams CUDA, 180-181 OpenACC interoperating with CUDA streams, 194-195 Strength reduction, compiling OpenACC, 93 Structured data data lifetimes for, 111 types of data directives, 12 Subarrays, optimization of data locality, 112 Sunway Taihulight data management in, 219-222 execution model in, 218-219 memory model, 217–218 overview of. 215-216 summary of, 223 SW26010 manycore CPU, 216-217 Supercomputers. See also Sunway Taihulight criteria requirements, 238 multiple compute nodes, 253

portability as goal in, 122 SW26010 manycore CPU, 216-217 swap/swapin/swapout. Sunway Taihulight data management, 221-222 Switches, in compiler interpretative of directives, 35 SYCL, layout for performance portability, 126 Symbolic loop bounds and steps, what compilers cannot do. 91 Symbolic subscript expressions, what compilers cannot do. 90 Synchronization comparing synchronous and asynchronous tasks, 189–190 of work queues, 191 Synchronization directives, 4 Svntax API routines and environment variables. 5 clauses, 4-5 directives. 3-4 overview of, 3 spotting errors, 33 System memory, 125

### Т

Task parallelism. See also Parallelism comparing dependent and independent tasks, 189-190 comparing programming models, 136, 151–152 defined, 188 functions of API routines. 5 TAU performance system, 48–51 TAUdb, performance data management, 48 tau exec, activating TAU performance measurement, 49 TBB. See Threading Building Blocks (TBB) TCA (Tightly coupled accelerators), 261–262 Templates C++. 137 XMP execution model, 255 Temporaries, storing expression value in, 85 Tests, runtime tuning and if clause, 107–108 Thermodynamic tables, 112–113 Thread parallelism, comparing programming models, 136 Thread safety, refactoring for, 105–106 Threading Building Blocks (TBB) comparing programming models, 143 data allocations, 153

Threading Building Blocks (TBB) (continued) features, 141 mapping simple loop to parallel loop, 145 MiniFE solver case study, 165-167 nontightly nested loops (hierarchical parallelism), 151 parallel reductions, 147 performance comparisons for MiniFE case, 167 task parallelism, 151–152 tightly nested loops, 148 Tightly coupled accelerators (TCA), 261–262 Tightly nested loops, in programming models, 147 Timelines, viewing temporal evolution of a program, 39 TotalView, debugging using, 52–53 Tracing data recording via, 39 generating with Score-P, 44-46 Vampir trace visualization, 47–48

### U

Unstructured data data lifetimes for, 111 types of data directives, 12 update directive data update, 4 interoperating OpenACC asynchronous work queues with CUDA streams, 194–195 making pipelining operation asynchronous, 202–204 use in blocking data movement, 200–201 using with MPI routines, 210–211 USE\_DEVICE clause, 177–180 User errors, compiling OpenACC, 95–97

### ۷

Vampir, trace visualization, 47–48 Variables auxiliary induction variable substitution, 93 data clauses, 12–13 private and firstprivate, 11 specifying scalar variables as private, 88 Vector addition. See AXPBY (vector addition) vector clause levels of parallelism, 21–22 mapping parallelism to hardware, 23–24 overview of, 22–23 Vectors directive extension for kernel vectorization, 244–245 distributing iterations across, 125–126 reducing vector or array to scalar, 86–87 scheduling parallel and vector code, 93–94 scheduling parallel loops, 227–230 temporary vectors required for MiniFE, 156 Verification, incremental acceleration and verification, 104

### W

wait directive events indicating runtime of device tasks, 41 joining work queues, 193-194 types of synchronization directives, 4 using with asynchronous operations, 191–192 using with MPI routines, 210-212 while loop, creating optimized parallel version of Laplace Solver, 75–78 WHIRL, OpenUH infrastructure, 224 Work distribution clauses, 4 Work mapping, XACC, 255-256 Work queues advanced OpenACC options, 190-191 interoperating OpenACC asynchronous work queues with CUDA streams, 194-195 joining, 193-194 worker clause levels of parallelism, 21-22 mapping parallelism to hardware, 23-24 overview of. 22-23 Workers applying OpenACC for parallelism, 87-88 distributing iterations across, 125-126 evaluating OpenACC translation to FPGAs, 248-249 principles governing loop performance, 96 scheduling parallel loops, 227–230

### Х

x86\_64 multicore, demonstrating portability using OpenACC, 123–124 XcalableACC (XACC) evaluating performance on HA-PACS, 262–267 implementation of OMNI compiler, 260–262 memory model, 257 overview of, 253 programming with XACC directives, 258–259 source-to-source code translation, 259–260 summary, 267 XcalableMP (XMP) data distribution and work mapping, 255–256 execution model in, 255 internode communication, 256–257 overview of, 253–254 Xeon E5-2698, 230 Xeon Phi KNL, 123–124