

Contents

Foreword	xv
Preface	xi
Acknowledgments	xxiii
About the Contributors	xxv
Chapter 1: OpenACC in a Nutshell	1
1.1 OpenACC Syntax	3
1.1.1 Directives	3
1.1.2 Clauses	4
1.1.3 API Routines and Environment Variables	5
1.2 Compute Constructs	6
1.2.1 Kernels	6
1.2.2 Parallel	8
1.2.3 Loop	8
1.2.4 Routine	9
1.3 The Data Environment	11
1.3.1 Data Directives	12
1.3.2 Data Clauses	12
1.3.3 The Cache Directive	13
1.3.4 Partial Data Transfers	14
1.4 Summary	15

1.5 Exercises	15	3.4 Identifying Bugs in OpenACC Programs	51
Chapter 2: Loop-Level Parallelism	17	3.5 Summary	53
2.1 Kernels Versus Parallel Loops	18	3.6 Exercises	54
2.2 Three Levels of Parallelism	21	Chapter 4: Using OpenACC for Your First Program	59
2.2.1 Gang, Worker, and Vector Clauses	22	4.1 Case Study	59
2.2.2 Mapping Parallelism to Hardware	23	4.1.1 Serial Code	61
2.3 Other Loop Constructs	24	4.1.2 Compiling the Code	67
2.3.1 Loop Collapse	24	4.2 Creating a Naive Parallel Version	68
2.3.2 Independent Clause	25	4.2.1 Find the Hot Spot	68
2.3.3 Seq and Auto Clauses	27	4.2.2 Is It Safe to Use kernels?	69
2.3.4 Reduction Clause	28	4.2.3 OpenACC Implementations	69
2.4 Summary	30	4.3 Performance of OpenACC Programs	71
2.5 Exercises	31	4.4 An Optimized Parallel Version	73
Chapter 3: Programming Tools for OpenACC	33	4.4.1 Reducing Data Movement	73
3.1 Common Characteristics of Architectures	34	4.4.2 Extra Clever Tweaks	75
3.2 Compiling OpenACC Code	35	4.4.3 Final Result	76
3.3 Performance Analysis of OpenACC Applications	36	4.5 Summary	78
3.3.1 Performance Analysis Layers and Terminology	37	4.6 Exercises	79
3.3.2 Performance Data Acquisition	38	Chapter 5: Compiling OpenACC	81
3.3.3 Performance Data Recording and Presentation	39	5.1 The Challenges of Parallelism	82
3.3.4 The OpenACC Profiling Interface	39	5.1.1 Parallel Hardware	82
3.3.5 Performance Tools with OpenACC Support	41	5.1.2 Mapping Loops	83
3.3.6 The NVIDIA Profiler	41	5.1.3 Memory Hierarchy	85
3.3.7 The Score-P Tools Infrastructure for Hybrid Applications	44	5.1.4 Reductions	86
3.3.8 TAU Performance System	48	5.1.5 OpenACC for Parallelism	87

5.2 Restructuring Compilers	88
5.2.1 What Compilers Can Do	88
5.2.2 What Compilers Can't Do	90
5.3 Compiling OpenACC	92
5.3.1 Code Preparation	92
5.3.2 Scheduling	93
5.3.3 Serial Code	94
5.3.4 User Errors	95
5.4 Summary	97
5.5 Exercises	97
Chapter 6: Best Programming Practices	101
6.1 General Guidelines	102
6.1.1 Maximizing On-Device Computation	103
6.1.2 Optimizing Data Locality	103
6.2 Maximize On-Device Compute	105
6.2.1 Atomic Operations	105
6.2.2 Kernels and Parallel Constructs	106
6.2.3 Runtime Tuning and the If Clause	107
6.3 Optimize Data Locality	108
6.3.1 Minimum Data Transfer	109
6.3.2 Data Reuse and the Present Clause	110
6.3.3 Unstructured Data Lifetimes	111
6.3.4 Array Shaping	111
6.4 A Representative Example	112
6.4.1 Background: Thermodynamic Tables	112
6.4.2 Baseline CPU Implementation	113

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	138
8.1.2 OpenMP	138
8.1.3 CUDA	139
8.1.4 OpenCL	139
8.1.5 C++ AMP	140
8.1.6 Kokkos	140
8.1.7 RAJA	141
8.1.8 Threading Building Blocks	141
8.1.9 C++17	142
8.1.10 Fortran 2008	142
8.2 Programming Model Components	142
8.2.1 Parallel Loops	143
8.2.2 Parallel Reductions	145
8.2.3 Tightly Nested Loops	147
8.2.4 Hierarchical Parallelism (Non-Tightly Nested Loops)	149
8.2.5 Task Parallelism	151
8.2.6 Data Allocation	152
8.2.7 Data Transfers	153
8.3 A Case Study	155
8.3.1 Serial Implementation	156
8.3.2 The OpenACC Implementation	157
8.3.3 The OpenMP Implementation	158
8.3.4 The CUDA Implementation	159
8.3.5 The Kokkos Implementation	163
8.3.6 The TBB Implementation	165
8.3.7 Some Performance Numbers	167

8.4 Summary	170
8.5 Exercises	170

Chapter 9: OpenACC and Interoperability 173

9.1 Calling Native Device Code from OpenACC	174
9.1.1 Example: Image Filtering Using DFTs	174
9.1.2 The host_data Directive and the use_device Clause	177
9.1.3 API Routines for Target Platforms	180
9.2 Calling OpenACC from Native Device Code	181
9.3 Advanced Interoperability Topics	182
9.3.1 acc_map_data	182
9.3.2 Calling CUDA Device Routines from OpenACC Kernels	184
9.4 Summary	185
9.5 Exercises	185

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 11: Innovative Research Ideas Using OpenACC, Part I 215

11.1 Sunway OpenACC	215
11.1.1 The SW26010 Manycore Processor	216

11.1.2 The Memory Model in the Sunway TaihuLight	217
11.1.3 The Execution Model	218
11.1.4 Data Management	219
11.1.5 Summary	223
11.2 Compiler Transformation of Nested Loops for Accelerators	224
11.2.1 The OpenUH Compiler Infrastructure	224
11.2.2 Loop-Scheduling Transformation	226
11.2.3 Performance Evaluation of Loop Scheduling	230
11.2.4 Other Research Topics in OpenUH	234
Chapter 12: Innovative Research Ideas Using OpenACC, Part II	237
12.1 A Framework for Directive-Based High-Performance Reconfigurable Computing	237
12.1.1 Introduction	238
12.1.2 Baseline Translation of OpenACC-to-FPGA	239
12.1.3 OpenACC Extensions and Optimization for Efficient FPGA Programming	243
12.1.4 Evaluation	248
12.1.5 Summary	252
12.2 Programming Accelerated Clusters Using XcalableACC	253
12.2.1 Introduction to XcalableMP	254
12.2.2 XcalableACC: XcalableMP Meets OpenACC	257
12.2.3 Omni Compiler Implementation	260
12.2.4 Performance Evaluation on HA-PACS	262
12.2.5 Summary	267
Index	269

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.