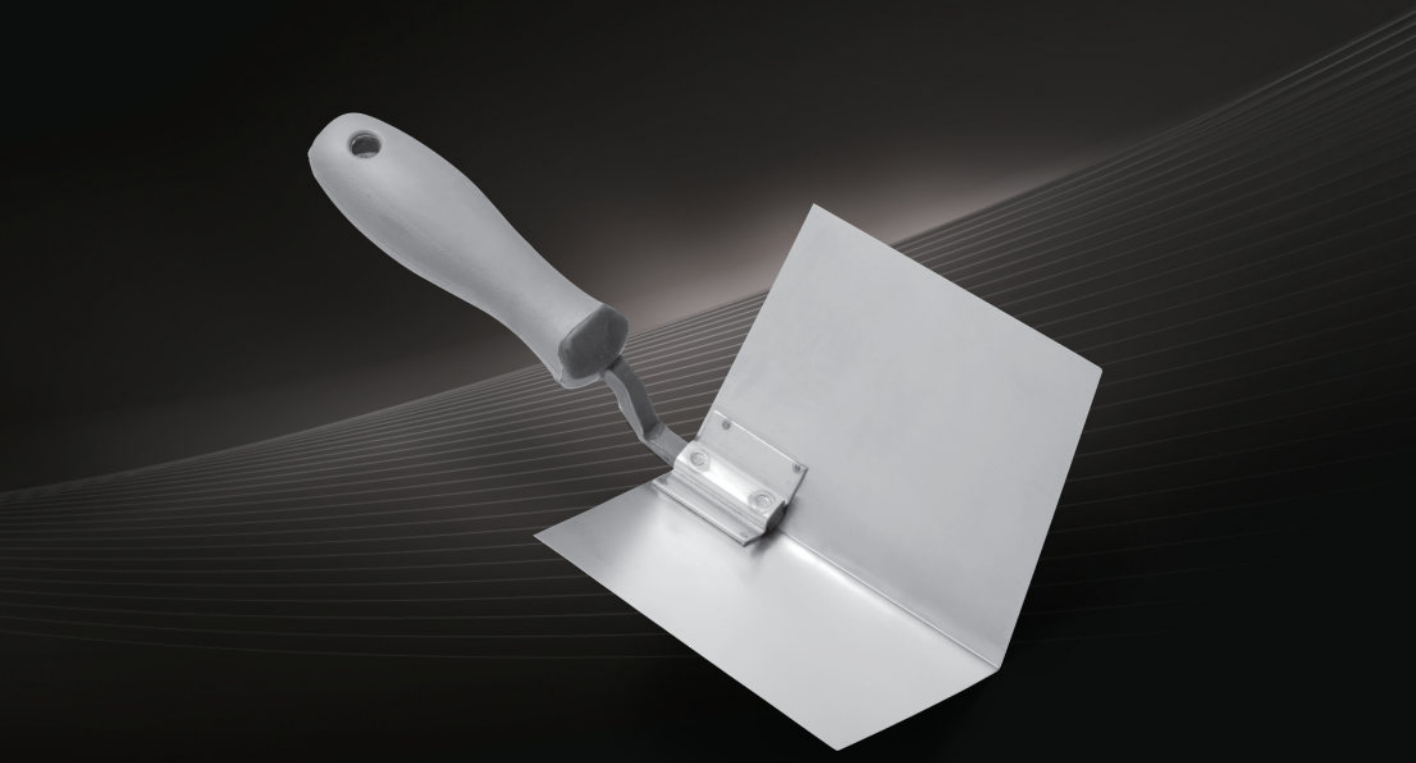
Microsoft

C++ AMP

Accelerated Massive Parallelism with Microsoft® Visual C++®



Kate Gregory
Ade Miller

Microsoft®

C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®

Kate Gregory
Ade Miller

Published with the authorization of Microsoft Corporation by:
O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, California 95472

ISBN: 978-0-7356-6473-9

1 2 3 4 5 6 7 8 9 LSI 7 6 5 4 3 2

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://www.microsoft.com/learning/booksurvey.

Acquisitions and Developmental Editor: Russell Jones

Production Editor: Holly Bauer

Editorial Production: nSight, Inc.

Copyeditor: nSight, Inc.

Indexer: nSight, Inc.

Cover Design: Twist Creative • Seattle

Cover Composition: Zyg Group, LLC

Illustrator: Rebecca Demarest

Dedicated to Brian, who has always been my secret weapon, and my children, now young adults who think it's normal for your mum to write books.

—Kate Gregory

Dedicated to The Susan,
who is so much more than I deserve.

—Ade Miller

Contents at a Glance

	Foreword	*xv*
	Introduction	*xvii*
CHAPTER 1	**Overview and C++ AMP Approach**	**1**
CHAPTER 2	**NBody Case Study**	**21**
CHAPTER 3	**C++ AMP Fundamentals**	**45**
CHAPTER 4	**Tiling**	**63**
CHAPTER 5	**Tiled NBody Case Study**	**83**
CHAPTER 6	**Debugging**	**101**
CHAPTER 7	**Optimization**	**127**
CHAPTER 8	**Performance Case Study—Reduction**	**171**
CHAPTER 9	**Working with Multiple Accelerators**	**203**
CHAPTER 10	**Cartoonizer Case Study**	**223**
CHAPTER 11	**Graphics Interop**	**257**
CHAPTER 12	**Tips, Tricks, and Best Practices**	**283**
APPENDIX	**Other Resources**	**309**
	Index	*313*
	About the Authors	*327*

Contents

Foreword . . . . . *xv*
Introduction . . . . . *xvii*

Chapter 1 Overview and C++ AMP Approach 1

Why GPGPU? What Is Heterogeneous Computing? 1
History of Performance Improvements 1
Heterogeneous Platforms 2
GPU Architecture 4
Candidates for Performance Improvement through Parallelism . . . 5
Technologies for CPU Parallelism 8
Vectorization 8
OpenMP 10
Concurrency Runtime (ConcRT) and Parallel Patterns Library 11
Task Parallel Library 12
WARP—Windows Advanced Rasterization Platform 12
Technologies for GPU Parallelism 13
Requirements for Successful Parallelism 14
The C++ AMP Approach 15
C++ AMP Brings GPGPU (and More) into the Mainstream 15
C++ AMP Is C++, Not C 16
C++ AMP Leverages Tools You Know 16
C++ AMP Is Almost All Library 17
C++ AMP Makes Portable, Future-Proof Executables 19
Summary 20

Chapter 2 NBody Case Study 21

Prerequisites for Running the Example 21
Running the NBody Sample 22
Structure of the Example 28

CPU Calculations 29
Data Structures 29
The *wWinMain* Function 30
The *OnFrameMove* Callback 30
The *OnD3D11CreateDevice* Callback 31
The OnGUIEvent Callback 33
The OnD3D11FrameRender Callback 33
The CPU NBody Classes 34
NBodySimpleInteractionEngine 34
NBodySimpleSingleCore 35
NBodySimpleMultiCore 35
NBodySimpleInteractionEngine::BodyBodyInteraction 35
C++ AMP Calculations 36
Data Structures 37
CreateTasks 38
The C++ AMP NBody Classes 40
NBodyAmpSimple::Integrate 40
BodyBodyInteraction 41
Summary 43

Chapter 3 C++ AMP Fundamentals 45

array<T, N> 45
accelerator and *accelerator_view* 48
index<N> 50
extent<N> 50
array_view<T, N> *51*
parallel_for_each 55
Functions Marked with *restrict(amp)* 57
Copying between CPU and GPU 59
Math Library Functions 61
Summary 62

Chapter 4 Tiling **63**

Purpose and Benefit of Tiling 64

tile_static Memory 65

tiled_extent 66

tiled_index<N1, N2, N3> 67

Modifying a Simple Algorithm into a Tiled One 68

Using *tile_static* memory 70

Tile Barriers and Synchronization 74

Completing the Modification of Simple into Tiled 76

Effects of Tile Size 77

Choosing Tile Size 79

Summary 81

Chapter 5 Tiled NBody Case Study **83**

How Much Does Tiling Boost Performance for NBody? 83

Tiling the n-body Algorithm 85

The NBodyAmpTiled Class 85

NBodyAmpTiled::Integrate 86

Using the Concurrency Visualizer 90

Choosing Tile Size 95

Summary 99

Chapter 6 Debugging **101**

First Steps 101

Choosing GPU or CPU Debugging 102

The Reference Accelerator 106

GPU Debugging Basics 108

Familiar Windows and Tips 108

The Debug Location Toolbar 109

Detecting Race Conditions 110

Seeing Threads 112
Thread Markers 113
GPU Threads Window 113
Parallel Stacks Window 115
Parallel Watch Window 117
Flagging, Grouping, and Filtering Threads 119
Taking More Control 121
Freezing and Thawing Threads 121
Run Tile to Cursor 123
Summary 125

Chapter 7 Optimization 127

An Approach to Performance Optimization 127
Analyzing Performance 128
Measuring Kernel Performance 129
Using the Concurrency Visualizer 131
Using the Concurrency Visualizer SDK 137
Optimizing Memory Access Patterns 138
Aliasing and *parallel_for_each* Invocations 138
Efficient Data Copying to and from the GPU 141
Efficient Accelerator Global Memory Access 146
Array of Structures vs. Structure of Arrays 149
Efficient Tile Static Memory Access 152
Constant Memory 155
Texture Memory 156
Occupancy and Registers 157
Optimizing Computation 158
Avoiding Divergent Code 158
Choosing the Appropriate Precision 161
Costing Mathematical Operations 163
Loop Unrolling 164
Barriers 165
Queuing Modes 168
Summary 169

Chapter 8 Performance Case Study—Reduction 171

The Problem. . . . 171
A Small Disclaimer . . . 172
Case Study Structure . . . 172
Initializations and Workload . . . 174
Concurrency Visualizer Markers . . . 175
TimeFunc() . . . 176
Overhead . . . 178
CPU Algorithms . . . 178
Sequential . . . 178
Parallel . . . 179
C++ AMP Algorithms . . . 179
Simple. . . . 180
Simple with *array_view* . . . 182
Simple Optimized. . . . 183
Naïvely Tiled . . . 185
Tiled with Shared Memory . . . 187
Minimizing Divergence . . . 192
Eliminating Bank Conflicts . . . 193
Reducing Stalled Threads . . . 194
Loop Unrolling . . . 195
Cascading Reductions . . . 198
Cascading Reductions with Loop Unrolling . . . 200
Summary . . . 201

Chapter 9 Working with Multiple Accelerators 203

Choosing Accelerators . . . 203
Using More Than One GPU. . . . 208
Swapping Data among Accelerators . . . 211
Dynamic Load Balancing . . . 216
Braided Parallelism . . . 219
Falling Back to the CPU . . . 220
Summary. . . . 222

Chapter 10 Cartoonizer Case Study **223**

Prerequisites 224

Running the Sample 224

Structure of the Sample 228

The Pipeline 229

Data Structures 229

The *CartoonizerDlg::OnBnClickedButtonStart()* Method 231

The *ImagePipeline* Class 232

The Pipeline Cartoonizing Stage 236

The *ImageCartoonizerAgent* Class 236

The *IFrameProcessor* Implementations 239

Using Multiple C++ AMP Accelerators 246

The *FrameProcessorAmpMulti* Class 246

The Forked Pipeline 249

The *ImageCartoonizerAgentParallel* Class 250

Cartoonizer Performance 252

Summary 255

Chapter 11 Graphics Interop **257**

Fundamentals 257

norm and *unorm* 258

Short Vector Types 259

texture<T, N> 262

writeonly_texture_view<T, N> 269

Textures vs. Arrays 270

Using Textures and Short Vectors 271

HLSL Intrinsic Functions 274

DirectX Interop 275

Accelerator View and Direct3D Device Interop 276

Array and Direct3D Buffer Interop 277

Texture and Direct3D Texture Resource Interop 277

Using Graphics Interop 280

Summary 282

Chapter 12 Tips, Tricks, and Best Practices 283

Dealing with Tile Size Mismatches 283
Padding Tiles 285
Truncating Tiles 286
Comparing Approaches 290
Initializing Arrays 290
Function Objects vs. Lambdas 291
Atomic Operations 292
Additional C++ AMP Features on Windows 8 295
Time-Out Detection and Recovery 296
Avoiding TDRs 297
Disabling TDR on Windows 8 297
Detecting and Recovering from a TDR 298
Double-Precision Support 299
Limited Double Precision 300
Full Double Precision 300
Debugging on Windows 7 300
Configure the Remote Machine 301
Configure Your Project 301
Deploy and Debug Your Project 302
Additional Debugging Functions 302
Deployment 303
Deploying your Application 303
Running C++ AMP on Servers 304
C++ AMP and Windows 8 Windows Store Apps 306
Using C++ AMP from Managed Code 306
From a .NET Application, Windows 7 Windows Store App or Library 306
From a C++ CLR Application 307
From within a C++ CLR Project 307
Summary 307

Appendix **Other Resources** **309**

More from the Authors 309

Microsoft Online Resources 309

Download C++ AMP Guides 309

Code and Support 310

Training 311

Index *313*

About the Authors *327*

Foreword

For most of computing history, we benefited from exponential increases in performance of scalar processors. That has come to an end. We are now at the dawn of the heterogeneous parallel computing era. With all applications being power-sensitive and all computing systems being power-limited, from mobile to cloud, future computing platforms must embrace heterogeneity. For example, a fast-growing portion of the top supercomputers in the world have become heterogeneous CPU + GPU computing clusters. While the first-generation programming interfaces such as CUDA and OpenCL have enabled development of new libraries and applications for these systems, there has been a clear need for much higher productivity in heterogeneous parallel software development.

The major challenge is that any programming interface that raises productivity in this domain must also give programmers enough control to reach their performance goals. C++ AMP from Microsoft is a major step forward in addressing this challenge. The C++ AMP interface is a simple, elegant extension to the C++ language to address two major weaknesses of previous interfaces. First, the previous approaches did not fit well with the C++ software engineering practice. The kernel-based parallel programming models tend to disturb the class organization of applications. Second, their C-based indexing for dynamically allocated arrays complicates the code for managing locality.

I am excited to see that C++ AMP supports the use of C++ loop constructs and objected-oriented features in parallel code to address the first issue and an *array_view* construct to address the second issue. The *array_view* approach is forward-looking and prepares applications to take full advantage of the upcoming unified address space architectures. Many experienced CUDA and OpenCL programmers have found the C++ AMP programming style refreshing, elegant, and effective.

Equally importantly, in my opinion, the C++ AMP interface opens the door for a wide range of innovative compiler transformations, such as data layout adjustment and thread granularity adjustment, to become mainstream. It also enables run-time implementation optimizations on data movement. Such advancements will be needed for a dramatic improvement in programmer productivity.

While C++ AMP is currently only implemented on Windows, the interface is open and will likely be implemented on other platforms. There is great potential for the C++ AMP interface to make an even bigger impact if and when the other platform vendors begin to offer their implementation of the interface.

This book's publication marks an important milestone in heterogeneous parallel computing. With this book, I expect to see many more developers who can productively develop heterogeneous parallel applications. I am honored to write this foreword and be part of this great movement. More important, I salute the C++ AMP engineering team at Microsoft who labored to make this advancement possible.

Wen-mei W. Hwu
Professor and Sanders-AMD Chair in ECE,
University of Illinois at Urbana-Champaign
CTO, MulticoreWare, Inc.

Introduction

C++ Accelerated Massive Parallelism (C++ AMP) is Microsoft's technology for accelerating C++ applications by allowing code to run on data-parallel hardware like graphics-processing units (GPUs.) It's intended not only to address today's parallel hardware in the form of GPUs and APUs, but also to future-proof your code investments by supporting new parallel hardware in the future. C++ AMP is also an open specification. Microsoft's implementation is built on top of DirectX, enabling portability across different hardware platforms. Other implementations can build on other technologies because the specification makes no requirement for DirectX.

The C++ AMP programming model comprises a modern C++ STL-like template library and two extensions to the C++ language that are integrated into the Visual C++ 2012 compiler. It's also fully supported by the Visual Studio toolset with IntelliSense editing, debugging, and profiling. C++ AMP brings the performance of heterogeneous hardware into the mainstream and lowers the barrier to entry for programming such systems without affecting your productivity.

This book shows you how to take advantage of C++ AMP in your applications. In addition to describing the features of C++ AMP, the book also contains several case studies that show realistic implementations of applications with various approaches to implementing some common algorithms. You can download the full source for these case studies and the sample code from each chapter and explore them for yourself.

Who Should Read This Book

This book's goal is to help C++ developers understand C++ AMP, from the core concepts to its more advanced features. If you are looking to take advantage of heterogeneous hardware to improve the performance of existing features within your application or add entirely new ones that were previously not possible due to performance limitations, then this book is for you.

After reading this book you should understand the best way to incorporate C++ AMP into your application where appropriate. You should also be able to use the debugging and profiling tools in Microsoft Visual Studio 2012 to troubleshoot issues and optimize performance.

Assumptions

This book expects that you have at least a working understanding of Windows C++ development, object-oriented programming concepts, and the C++ Standard Library (often called the STL after its predecessor, the Standard Template Library.) Familiarity with general parallel processing concepts is also helpful but not essential. Some of the samples use DirectX, but you don't need to have any DirectX background to use the samples or to understand the C++ AMP code in them.

For a general introduction to the C++ language, consider reading Bjarne Stroustrup's *The C++ Programming Language* (Addison-Wesley, 2000). This book makes use of many new language and library features in C++11, which is so new that at the time of press there are few resources covering the new features. Scott Meyers's *Presentation Materials: Overview of the New C++ (C++11)* provides a good overview. You can purchase it online from Artima Developer, *http://www.artima.com/shop/overview_of_the_new_cpp*. Nicolai M. Josuttis's *The C++ Standard Library: A Tutorial and Reference* (2nd Edition) (Addison-Wesley Professional, 2012) is a good introduction to the Standard Library.

The samples in this book also make extensive use of the Parallel Patterns Library and the Asynchronous Agents Library. *Parallel Programming with Microsoft Visual C++* (Microsoft Press, 2011), by Colin Campbell and Ade Miller, is a good introduction to both libraries. This book is also available free from MSDN, *http://msdn.microsoft.com/en-us/library/gg675934.aspx*.

Who Should Not Read This Book

This book isn't intended to teach you C++ or the Standard Library. It assumes a working knowledge of both the language and the library. This book is also not a general introduction to parallel programming or even multithreaded programming. If you are not familiar with these topics, you should consider reading some of the books referenced in the previous section.

Organization of This Book

This book is divided into 12 chapters. Each focuses on a different aspect of programming with C++ AMP. In addition to chapters on specific aspects of C++ AMP, the book also includes three case studies designed to walk through key C++ AMP features used

in real working applications. The code for each of the case studies, along with the samples shown in the other chapters, is available for download on CodePlex.

Chapter 1 Overview and C++ AMP Approach	An introduction to GPUs, heterogeneous computing, parallelism on the CPU, and how C++ AMP allows applications to harness the power of today's heterogeneous systems.
Chapter 2 NBody Case Study	Implementing an n-body simulation using C++ AMP.
Chapter 3 C++ AMP Fundamentals	A summary of the library and language changes that make up C++ AMP and some of the rules your code must follow.
Chapter 4 Tiling	An introduction to tiling, which breaks a calculation into groups of threads called tiles that can share access to a very fast programmable cache.
Chapter 5 Tiled NBody Case Study	An explanation of the tiled version of the NBody sample described in Chapter 2.
Chapter 6 Debugging	A review of the techniques and tools for debugging a C++ AMP application in Visual Studio.
Chapter 7 Optimization	More details on the factors that affect performance of a C++ AMP application, on how to measure performance, and on how to adjust your code to get the maximum speed.
Chapter 8 Performance Case Study—Reduction	A review of a single simple calculation implemented in a variety of ways and the performance changes brought about by each implementation change.
Chapter 9 Working with Multiple Accelerators	How to take advantage of multiple GPUs for maximum performance, braided parallelism, and using the CPU to ensure that you use the GPU as efficiently as possible.
Chapter 10 Cartoonizer Case Study	An explanation of a complex sample that combines CPU parallelism with C++ AMP parallelism and supports multiple accelerators.
Chapter 11 Graphics Interop	Using C++ AMP in conjunction with DirectX.
Chapter 12 Tips, Tricks, and Best Practices	Instructions on how to deal with less common situations and environments and to overcome some common problems.
Appendix Other Resources	Online resources, support, and training for those who want to learn even more about C++ AMP.

Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow.

- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step.

- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.
- A vertical bar between two or more menu items (for example, File | Close), means that you should select the first menu or menu item, then the next, and so on.

System Requirements

You will need the following hardware and software to build and run the samples in this book:

- Either Microsoft Windows 7 with Service Pack 1 or Windows 8 (x86 or x64). The samples should also build and run on Windows Server 2008 R2 (x64) and Windows Server 2012 (x64), but they have not been tested on these OSs.
- Visual Studio 2012, any edition (the Professional or Ultimate product is required to walk through the profiling examples in chapters 7 and 8).
- The DirectX SDK (June 2010) is required to build the NBody case study.
- A computer that has a 1.6GHz or faster processor. A four-core processor is recommended.
- 1 GB (32-bit) or 2 GB (64-bit) RAM.
- 10 GB of available hard disk space (for installing Visual Studio 2012).
- 5400 RPM hard disk drive.
- A DirectX 11 capable video card (for the C++ AMP samples) running at 1024 x 768 or higher-resolution display (for Visual Studio 2012).
- A DVD-ROM drive (if installing Visual Studio 2012 from a DVD).
- An Internet connection to download software or chapter examples.

Code Samples

Most of the chapters in this book include samples that let you interactively try out new material learned in the main text. The working examples can be seen on the web at:

http://go.microsoft.com/FWLink/?Linkid=260980

Follow the instructions to download the source zip file.

Note In addition to the code samples, your system should have Visual Studio 2012 and the DirectX SDK (June 2010) installed. If they're available, install the latest service packs for each product.

Installing the Code Samples

Follow these steps to install the code samples on your computer:

1. Download the source zip file from the book's CodePlex website, *http://ampbook.codeplex.com/*. You can find the latest download on the Downloads tab. Choose the most recent recommended download.
2. If prompted, review the displayed end user license agreement. If you accept the terms, choose the Accept option and then click Next.
3. Unzip the file into a folder and open the BookSamples.sln file using Visual Studio 2012.

Note If the license agreement doesn't appear, you can access it from the CodePlex site, *http://ampbook.codeplex.com/license*. A copy is also included with the sample code.

Using the Code Samples

The Samples folder that's created by unzipping the sample download contains three subfolders:

- **CaseStudies** This folder contains the three case studies described in chapters 2, 8, and 10. Each case study has a separate folder:
  - **NBody** An n-body gravitational model
  - **Reduction** A series of implementations of the reduce algorithm designed to show performance tradeoffs
  - **Cartoonizer** An image-processing application that cartoonizes sequences of images either loaded from disk or captured by a video camera
- **Chapter 4, 7, 9, 11, 12** Folders containing the code that accompanies the corresponding chapters.
- **ShowAmpDevices** A small utility application that lists the C++ AMP-capable devices present on the host computer.

The top-level Samples folder contains a Visual Studio 2012 solution file, BookSamples.sln. This contains all the projects listed above. It should compile with no warnings or errors in Debug and Release configurations and can target both Win32 and x64 platforms. Each of the projects also has its own solution file, should you wish to load them separately.

Acknowledgments

No book is the effort of any one person. This book has two authors, but many others helped along the way. The authors would like to thank the following people:

The C++ AMP product team at Microsoft who went above and beyond to provide review feedback on draft chapters and answered numerous questions; Amit Agarwal, David Callahan, Charles Fu, Jerry Higgins, Yossi Levanoni, Don McCrady, Łukasz Mendakiewicz, Daniel Moth, Bharath Mysore Nanjundappa, Pooja Nagpal, James Rapp, Simon Wybranski, Lingli Zhang, and Weirong Zhu (Microsoft Corporation).

The C++ AMP team also maintains a blog that provided invaluable source material. Many of the reviewers from the C++ AMP product team listed above also wrote those posts. In addition, the following also wrote material we found particularly helpful: Steve

Deitz, Kevin Gao, Pavan Kumar, Paul Maybee, Joe Mayo, and Igor Ostrovsky (Microsoft Corporation.)

Ed Essey and Daniel Moth (Microsoft Corporation) were instrumental in getting the whole project started and approaching O'Reilly and the authors with the idea of a book about C++ AMP. They also coordinated our work with the C++ AMP product team.

Thank you also Russell Jones and Holly Bauer and Carol Whitney, who handled copy-editing and production, and Rebecca Demarest, the technical illustrator.

We were also lucky enough to be able to circulate early drafts of the book on Safari through O'Reilly's Rough Cuts program. Many people provided feedback on these early drafts. We would like to thank them for their time and interest. Bruno Boucard and Veikko Eeva have been particularly helpful and enthusiastic reviewers.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com:

http://go.microsoft.com/FWLink/?Linkid=260979

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, e-mail Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

CHAPTER 1

Overview and C++ AMP Approach

In this chapter:

Why GPGPU? What Is Heterogeneous Computing? 1
Technologies for CPU Parallelism 8
The C++ AMP Approach 15
Summary 20

Why GPGPU? What Is Heterogeneous Computing?

As developers, we are used to adjusting to a changing world. Our industry changes the world almost as a matter of routine. We learn new languages, adopt new methodologies, start using new user interface paradigms, and take for granted that it will always be possible to make our programs better. When it seems we will "hit a wall" following one path to making version n+1 better than version n, we find another path. The newest path some developers are about to follow is the path of heterogeneous computing.

In this chapter you'll review some of the history of performance improvements to see what wall some developers are facing. You'll learn the basic differences between a CPU and a GPU, two of the possible components of a heterogeneous computing solution, and what kinds of problems are suitable for acceleration using these parallel techniques. Then you'll review the CPU and GPU parallel techniques in use today, followed by an introduction to the concepts behind C++ AMP, to lay the groundwork for the details in the subsequent chapters.

History of Performance Improvements

In the mid-seventies, computers intended for use by a single person were not the norm. The phrase "personal computer" dates back only to 1975. Over the decades that followed, the idea of a computer on every desk changed from an ambitious and perhaps impossible goal to something pretty ordinary. In fact, many desks today have *more* than one computer, and what's more, so do many living rooms. A lot of people even carry a small computer in their pocket, in the form of a smartphone. For the first 30 years of that expansive growth, computers didn't just get cheaper and more popular—they also

got faster. Each year, manufacturers released chips that had a higher clock speed, more cache, and better performance. Developers got in the habit of adding features and capabilities to their software. When those additions made the software run more slowly, the developers didn't worry much; in six months to a year, faster machines would be available and the software would again become fast and responsive. This was the so-called "free lunch" enabled by ever-improving hardware performance. Eventually, performance on the level of gigaFLOPS—billions of floating points operations per second—became attainable and affordable.

Unfortunately, this "free lunch" came to an end in about 2005. Manufacturers continued to increase the number of transistors that could be placed on a single chip, but physical limitations—such as dissipating the heat from the chip—meant that clock speeds could no longer continue to increase. Yet the market, as always, wanted more powerful machines. To meet that demand, manufacturers began to ship *multicore* machines, with two, four, or more CPUs in a single computer. "One user, one CPU" had once been a lofty goal, but after the free lunch era, users called for more than just one CPU core, first in desktop machines, then in laptops, and eventually in smartphones as well. Over the past five or six years, it's become common to find a parallel supercomputer on every desk, in every living room, and in everyone's pocket.

But simply adding cores didn't make everything faster. Software can be roughly divided into two main groups: parallel-aware and parallel-unaware. The parallel-unaware software typically uses only half, a quarter, or an eighth of the cores available. It churns away on a single core, missing the opportunity to get faster every time users get a new machine with more cores. Developers who have learned how to write software that gets faster as more CPU cores become available achieve close to linear speedups; in other words, a speed improvement that comes close to the number of cores on the machine—almost double for dual-core machines, almost four times for four-core machines, and so on. Knowledgeable consumers might wonder why some developers are ignoring the extra performance that could be available to their applications.

Heterogeneous Platforms

Over the same five-year or six-year period that saw the rise of multicore machines with more than one CPU, the graphics cards in most machines were changing as well. Rather than having two or four CPU cores, GPUs were being developed with dozens, or even hundreds, of cores. These cores are very different from those in a CPU. They were originally developed to improve the speed of graphics-related computations, such as determining the color of a particular pixel on the screen. GPUs can do that kind of work faster than a CPU, and because modern graphics cards contain so many of them, massive parallelism is possible. Of course, the idea of harnessing a GPU for numerical calculations *unrelated* to graphics quickly became irresistible. A machine with a mix of CPU and GPU cores, whether on the same chip or not, or even a cluster of machines offering such a mix, is a heterogeneous supercomputer. Clearly, we are headed toward a heterogeneous supercomputer on every desk, in every living room, and in every pocket.

A typical CPU in early 2012 has four cores, is double hyper-threaded, and has about a billion transistors. A top end CPU can achieve, at peak, about 0.1 TFlop or 100 GFlops doing double-precision calculations. A typical GPU in early 2012 has 32 cores, is 32×-threaded, and has roughly twice as many

transistors as the CPU. A top-end GPU can achieve 3 TFlop—some 30 times the peak compute speed of the CPU—doing single-precision calculations.

Note Some GPUs support double precision and some do not, but the reported performance numbers are generally for single precision.

The reason the GPU achieves a higher compute speed lies in differences other than the number of transistors or even the number of cores. A CPU has a low memory bandwidth—about 20 gigabytes per second (GB/s)—compared to the GPU's 150 GB/s. The CPU supports general code with multitasking, I/O, virtualization, deep execution pipelines, and random accesses. In contrast, the GPU is designed for graphics and data-parallel code with programmable and fixed function processors, shallow execution pipelines, and sequential accesses. The GPU's speed improvements, in fact, are available only on tasks for which the GPU is designed, not on general-purpose tasks. Possibly even more important than speed is the GPU's lower power consumption: a CPU can do about 1 gigaflop per watt (GFlop/watt) whereas the GPU can do about 10 GFlop/watt.

In many applications, the power required to perform a particular calculation might be more important than the time it takes. Handheld devices such as smartphones and laptops are battery-powered, so users often wisely choose to replace applications that use up the battery too fast with more battery-friendly alternatives. This can also be an issue for laptops, whose users might expect all-day battery life while running applications that perform significant calculations. It's becoming normal to expect multiple CPUs even on small devices like smartphones—and to expect those devices to have a GPU. Some devices have the ability to power individual cores up and down to adjust battery life. In that kind of environment, moving some of your calculation to the GPU might mean the difference between "that app I can't use away from the office because it just eats battery" and "that app I can't live without." At the other end of the spectrum, the cost of running a data center is overwhelmingly the cost of supplying power to that data center. A 20 percent saving on the watts required to perform a large calculation in a data center or the cloud can translate directly into bottom-line savings on a significant energy bill.

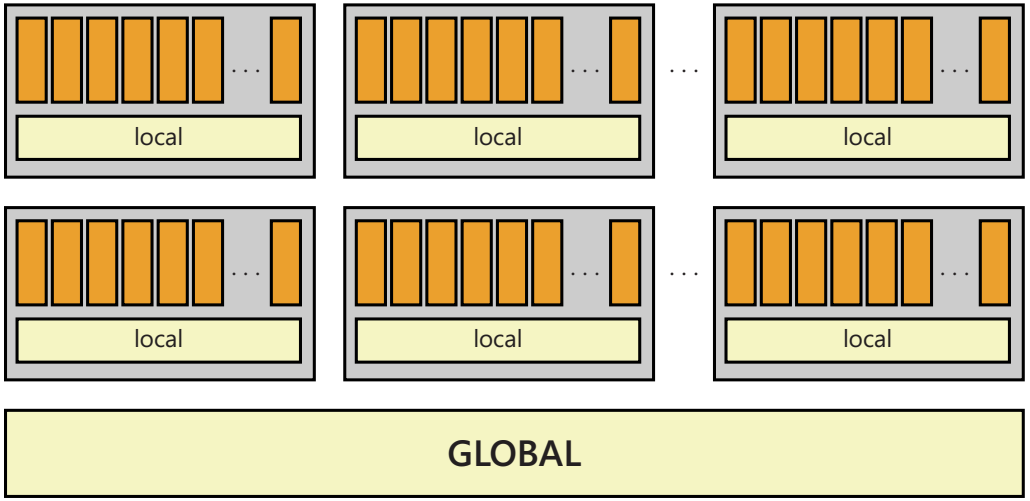
Then there is the matter of the memory accessed by these cores. Cache size can outweigh clock speed when it comes to compute speed, so the CPU has a large cache to make sure that there is always data ready to be processed, and the core will rarely have to wait while data is fetched. It's normal for CPU operations to touch the same data repeatedly, giving a real benefit to caching approaches. In contrast, GPUs have smaller caches but use a massive number of threads, so some threads are always in a position to do work. GPUs can prefetch data to hide memory latency, but because that data is likely to be accessed only once, caching provides less benefit and is less necessary. For this approach to help, you ideally have a huge quantity of data and a fairly simple calculation that operates on the data.

Perhaps the most important difference of all lies in how developers program the two technologies. Many mainstream languages and tools exist for CPU programming. For power and performance, C++ is the number one choice, providing abstractions and powerful libraries without giving up control. For general-purpose GPU programming (GPGPU), the choices are far more restricted and in most

cases involve a niche or exotic programming model. This restriction has meant that—until now—only a handful of fields and problems have been able to capitalize on the power of the GPU to tackle their compute-intensive number-crunching, and it has also meant that mainstream developers have avoided learning how to interact with the GPU. Developers need a way to increase the speed of their applications or to reduce the power consumption of a particular calculation. Today that might come from using the GPU. An ideal solution sets developers up to get those benefits now by using the GPU and later by using other forms of heterogeneous computation.

GPU Architecture

As mentioned earlier, GPUs have shallow execution pipelines, small cache, and a massive number of threads performing sequential accesses. These threads are not all independent; they are arranged in groups. These groups are called *warps* on NVIDIA hardware and *wavefronts* on AMD hardware. In this book, they are referred to as "warps." Warps run together and can share memory and cooperate. Local memory can be read in as little as four clock cycles, while the larger (up to four GB) global memory might take 400–600 cycles. If a group of threads is blocked while reading, another group of threads executes. The GPU can switch these groups of threads extremely fast. Memory is read in a way that provides huge speed advantages when adjacent threads use adjacent memory locations. But if some threads in a group are accessing memory that is nowhere near the memory being accessed by other threads in that group, performance will suffer.



There's a large dissimilarity between CPU and GPU architectures. Developers using higher-level languages have generally been able to ignore CPU architecture. Lower-level tools such as operating systems and optimizing compilers must have that kind of architectural knowledge, but the compiler

and the operating system insulate many "ordinary" applications from hardware details. Best practices or rules of thumb that you might hold as self-evident are perhaps not self-evident; even on the CPU, a simple integer addition that causes a cache miss might take far longer than a disk read that accessed only the buffered file contents from a nearby cache.

Some developers, finding themselves writing highly performance-sensitive applications, might need to learn just how many instructions can be executed in the time lost to a cache miss or how many clock cycles it takes to read a byte from a file (millions, in many cases). At the moment, this kind of knowledge is unavoidable when working with non-CPU architectures such as the GPU. The layers of protection that compilers and operating systems provide for CPU programming are not entirely in place yet. For example, you might need to know how many threads are in a warp or the size of their shared memory cache. You might arrange your computation so that iterations involve adjacent memory and avoid random accesses. To understand the speedups your application can achieve, you must understand, at least at a conceptual level, the way the hardware is organized.

Candidates for Performance Improvement through Parallelism

The GPU works best on problems that are data-parallel. Sometimes it's obvious how to split one large problem up into many small problems that a processor can work on independently and in parallel. Take matrix addition, for example: each element in the answer matrix can be calculated entirely independently of the others. Adding a pair of 100×100 matrices will take 10,000 additions, but if you could split it among 10,000 threads, all the additions could be done at once. Matrix addition is naturally data-parallel.

In other cases, you need to design your algorithm differently to create work that can be split across independent threads. Consider the problem of finding the highest value in a large collection of numbers. You could traverse the list one element at a time, comparing each element to the "currently highest" value and updating the "currently highest" value each time you come across a larger one. If 10,000 items are in the collection, this will take 10,000 comparisons. Alternatively, you could create some number of threads and give each thread a piece of the collection to work on. 100 threads could take on 100 items each, and each thread would determine the highest value in its portion of the collection. That way you could evaluate every number in the time it takes to do just 100 comparisons. Finally, a 101st thread could compare the 100 "local highest" numbers—one from each thread—to establish the overall highest value. By tweaking the number of threads and thus the number of comparisons each thread makes, you can minimize the elapsed time to find the highest value in the collection. When the comparisons are much more expensive than the overhead of making threads, you might take an extreme approach: 5,000 threads each compare two values, then 2,500 threads each compare the winners of the first round, 1,250 threads compare the winners of the second round, and so on. Using this approach, you'd find the highest value in just 14 rounds—the elapsed time of 14 comparisons, plus the overhead. This "tournament" approach can also work for other operations, such as adding all the values in a collection, counting how many values are in a specified range, and so on. The term *reduction* is often used for the class of problems that seek a single number (the total, minimum, maximum, or the like) from a large data set.

It turns out that any problem set involving large quantities of data is a natural candidate for parallel processing. Some of the first fields to take this approach include the following:

- **Scientific modeling and simulation** Physics, biology, biochemistry, and similar fields use simple equations to model immensely complicated situations with massive quantities of data. The more data included in the calculation, the more accurate the simulation. Testing theories in a simulation is feasible only if the simulation can be run in a reasonable amount of time.
- **Real-time control systems** Combining data from myriad sensors, determining where operation is out of range, and adjusting controls to restore optimal operation are high-stakes processes. Fire, explosion, expensive shutdowns, and even loss of life are what the software is working to avoid. Usually the number of sensors being read is limited by the time it takes to make the calculations.
- **Financial tracking, simulation, and prediction** Highly complicated calculations often require a great deal of data to establish trends or identify gaps and opportunities for profit. The opportunities must be identified while they still exist, putting a firm upper limit on the time available for the calculation.
- **Gaming** Most games are essentially a simulation of the real world or a carefully modified world with different laws of physics. The more data you can include in the physics calculations, the more believable the game is—yet performance simply cannot lag.
- **Image processing** Whether detecting abnormalities in medical images, recognizing faces on security camera footage, confirming fingerprint matches, or performing any of dozens of similar tasks, you want to avoid both false negatives and false positives, and the time available to do the work is limited.

In these fields, when you achieve a 10× speedup in the application that is crunching the numbers, you gain one of two abilities. In the simplest case, you can now include more data in the calculations without the calculations taking longer. This generally means that the results will be more accurate or that end users of the application can have more confidence in their decisions. Where things really get interesting is when the speedup makes possible things that were impossible before. For example, if you can perform a 20-hour financial calculation in just two hours, you can do that work overnight while the markets are closed, and people can take action in the morning based on the results of that calculation. Now, what if you were to achieve a 100× speedup? A calculation that formerly required 1,000 hours—over 40 days—is likely to be based on stale data by the time it completes. However, if that same calculation takes only 10 hours—overnight—the results are much more likely to still be meaningful.

Time windows aren't just a feature of financial software—they apply to security scanning, medical imaging, and much more, including a rather scary set of applications in password cracking and data mining. If it took 40 days to crack your password by brute force and you changed it every 30 days, your password was safe. But what happens when the cracking operation takes only 10 hours?

A 10× speedup is relatively simple to achieve, but a 100× speedup is much harder. It's not that the GPU can't do it—the problem is the contribution of the nonparallelizable parts of the application.

Consider three applications. Each takes 100 arbitrary units of time to perform a task. In one, the non-parallelizable parts (say, sending a report to a printer) take up 25 percent of the total time. In another, they require only 1 percent, and in the third, only 0.1 percent. What happens as you speed up the parallelizable part of each of these applications?

		App1	App2	App3
	% sequential	25%	1%	0.1%
Original	Sequential time	25	1	0.1
	Parallel time	75	99	99.9
	Total time	100	100	100
10×	Sequential time	25	1	0.1
	Parallel time	7.5	9.9	9.99
	Total time	32.5	10.9	10.09
	Speedup	**3.08**	**9.17**	**9.91**
100×	Sequential time	25	1	0.1
	Parallel time	0.75	0.99	0.999
	Total time	25.75	1.99	1.099
	Speedup	**3.88**	**50.25**	**90.99**
Infinite	Sequential time	25	1	0.1
	Parallel time	0	0	0
	Total time	25	1	0.1
	Speedup	**4.00**	**100.00**	**1000.00**

With a 10× speedup in the parallel part, the first application now spends much more time in the sequential part than in the parallelizable part. The overall speedup is a little more than 3×. Finding a 100× speedup in the parallel part doesn't help much because of the enormous contribution of the sequential part. Even an infinite speedup, reducing the time in the parallel part to zero, can't erase the sequential part and limits the overall speedup to 4×. The other two applications fare better with the 10× speedup, but the second app can't benefit from all of the 100× speedup, gaining only 50× overall. Even with an infinite speedup, the second app is limited to 100× overall.

This seeming paradox—that the contribution of the sequential part, no matter how small a fraction it is at first, will eventually be the final determiner of the possible speedup—is known as Amdahl's Law. It doesn't mean that 100× speedup isn't possible, but it does mean that choosing algorithms to minimize the nonparallelizable part of the time spent is very important for maximum improvement. In addition, choosing a data-parallel algorithm that opens the door to using the GPGPU to speed up the application might result in more overall benefit than choosing a very fast and efficient algorithm that is highly sequential and cannot be parallelized. The right decision for a problem with a million data points might not be the right decision for a problem with 100 million data points.

Technologies for CPU Parallelism

One way to reduce the amount of time spent in the sequential portion of your application is to make it less sequential—to redesign the application to take advantage of CPU parallelism as well as GPU parallelism. Although the GPU can have thousands of threads at once and the CPU far less, leveraging CPU parallelism as well still contributes to the overall speedup. Ideally, the technologies used for CPU parallelism and GPU parallelism would be compatible. A number of approaches are possible.

Vectorization

An important way to make processing faster is SIMD, which stands for Single Instruction, Multiple Data. In a typical application, instructions must be fetched one at a time and different instructions are executed as control flows through your application. But if you are performing a large data-parallel operation like matrix addition, the instructions (the actual addition of the integers or floating-point numbers that comprise the matrices) are the same over and over again. This means that the cost of fetching an instruction can be spread over a large number of operations, performing the same instruction on different data (for example, different elements of the matrices.) This can amplify your speed tremendously or reduce the power consumed to perform your calculation.

Vectorization refers to transforming your application from one that processes a single piece of data at a time, each with its own instructions, into one that processes a vector of information all at once, applying the same instruction to each element of the vector. Some compilers can do this automatically to some loops and other parallelizable operations.

Microsoft Visual Studio 2012 supports manual vectorization using SSE (Streaming SIMD Extensions) intrinsics. Intrinsics appear to be functions in your code, but they map directly to a sequence of assembly language instructions and do not incur the overhead of a function call. Unlike in inline assembly, the optimizer can understand intrinsics, allowing it to optimize other parts of your code accordingly. Intrinsics are more portable than inline assembly, but they still have some possible portability problems because they rely on particular instructions being available on the target architecture. It is up to the developer to ensure that the target machine has a chip that supports these intrinsics. Not surprisingly, there is an intrinsic for that: *__cpuid()* generates instructions that fill four integers with information about the capabilities of the processor. (It starts with two underscores because it is compiler-specific.) To check if SSE3 is supported, you would use the following code:

```
int CPUInfo[4] = { -1 };
__cpuid(CPUInfo, 1);
bool bSSEInstructions = (CpuInfo[3] >> 24 && 0x1);
```

Note The full documentation of *__cpuid*, including why the second parameter is 1 and the details of which bit to check for SSE3 support, as well as how to check for support of other features you might use, is in the "__cpuid" topic on MSDN at *http://msdn.microsoft.com/en-us/library/hskdteyh(v=vs.100).aspx*.

Which intrinsic you would use depends on how you are designing your work to be more parallel. Consider the case in which you need to add many pairs of numbers. The single intrinsic *_mm_hadd_epi32* will add four pairs of 32-bit numbers at once. You fill two memory-aligned 128-bit numbers with the input values and then call the intrinsic to add them all at once, getting a 128-bit result that you can split into the four 32-bit numbers representing the sum of each pair. Here is some sample code from MSDN:

```
#include <stdio.h>
#include <tmmintrin.h>

int main ()
{
    __m128i a, b;

    a.m128i_i32[0] = -1;
    a.m128i_i32[1] = 1;
    a.m128i_i32[2] = 0;
    a.m128i_i32[3] = 65535;
    b.m128i_i32[0] = -65535;
    b.m128i_i32[1] = 0;
    b.m128i_i32[2] = 128;
    b.m128i_i32[3] = -32;

    __m128i res = _mm_hadd_epi32(a, b);

    std::wcout << "Original a: " <<
    a.m128i_i32[0] << "\t" << a.m128i_i32[1] << "\t" <<
    a.m128i_i32[2] << "\t" << a.m128i_i32[3] << "\t" << std::endl;
    std::wcout << "Original b: " <<
    b.m128i_i32[0] << "\t" << b.m128i_i32[1] << "\t" <<
    b.m128i_i32[2] << "\t" << b.m128i_i32[3] << std::endl;
    std::wcout << "Result res: " <<
    res.m128i_i32[0] << "\t" << res.m128i_i32[1] << "\t" <<
    res.m128i_i32[2] << "\t" << res.m128i_i32[3] <<std::endl;

    return 0;
}
```

The first element of the result contains *a0* + *a1*, the second contains *a2* + *a3*, the third contains *b0* + *b1*, and the fourth contains *b2* + *b3*. If you can redesign your code to do your additions in pairs and to group the pairs into clumps of four, you can parallelize your code using this intrinsic. There are intrinsics to perform a variety of operations (including add, subtract, absolute value, negate—even do dot products using 16 × 16 8-bit integers) in several "widths," or number of calculations at a time.

One drawback of vectorization with these intrinsics is that the readability and maintainability of the code falls dramatically. Typically, code is written "straight up" first, tested for correctness, and then, when profiling reveals areas of the code that are performance bottlenecks and candidates for vectorization, adapted to this less-readable state.

In addition, Visual Studio 2012 implements auto-vectorization and auto-parallelization of your code. The compiler will automatically vectorize loops if it is possible. Vectorization reorganizes a loop—for example, a summation loop—so that the CPU can execute multiple iterations at the same

time. By using auto-vectorization, loops can be up to eight times faster when executed on CPUs that support SIMD instructions. For example, most modern processors support SSE2 instructions, which allow the compiler to instruct the processor to do math operations on four numbers at a time. The speedup is achieved even on single-core machines, and you don't need to change your code at all.

Auto-parallelization reorganizes a loop so that it can be executed on multiple threads at the same time, taking advantage of multicore CPUs and multiprocessors to distribute chunks of the work to all available processors. Unlike auto-vectorization, you tell the compiler which loops to parallelize with the *#pragma parallelize* directive. The two features can work together so that a vectorized loop is then parallelized across multiple processors.

OpenMP

OpenMP (the MP stands for multiprocessing) is a cross-language, cross-platform application programming interface (API) for CPU-parallelism that has existed since 1997. It supports Fortran, C, and C++ and is available on Windows and a number of non-Windows platforms. Visual C++ supports OpenMP with a set of compiler directives. The effort of establishing how many cores are available, creating threads, and splitting the work among the threads is all done by OpenMP. Here is an example:

```
// size is a compile-time constant
double* x = new double[size];
double* y = new double[size + 1];
// get values into y
#pragma omp parallel for
for (int i = 1; i < size; ++i)
{
    x[i] = (y[i - 1] + y[i + 1]) / 2;
}
```

This code fragment uses vectors *x* and *y* and visits each element of *y* to build *x*. Adding the *pragma* and recompiling your program with the /openmp flag is all that is needed to split this work among a number of threads—one for each core. For example, if there are four cores and the vectors have 10,000 elements, the first thread might be given *i* values from 1 to 2,500, the second 2,501 to 5,000, and so on. At the end of the loop, *x* will be properly populated. The developer is responsible for writing a loop that is parallelizable, of course, and this is the truly hard part of the job. For example, this loop is not parallelizable:

```
for (int i = 1; i <= n; ++i)
    a[i] = a[i - 1] + b[i];
```

This code contains a loop-carried dependency. For example, to determine *a[2502]* the thread must have access to *a[2501]*—meaning the second thread can't start until the first has finished. A developer can put the pragma into this code and not be warned of a problem, but the code will not produce the correct result.

One of the major restrictions with OpenMP arises from its simplicity. A loop from 1 to *size*, with *size* known when the loop starts, is easy to divide among a number of threads. OpenMP can only handle loops that involve the same variable (*i* in this example) in all three parts of the *for-loop* and only when the test and increment also feature values that are known at the start of the loop.

This example:

```
for (int i = 1; (i * i) <= n; ++i)
```

cannot be parallelized with *#pragma omp parallel for* because it is testing the square of *i*, not just *i*. This next example:

```
for (int i = 1; i <= n; i += Foo(abc))
```

also cannot be parallelized with *#pragma omp parallel for* because the amount by which *i* is incremented each time is not known in advance.

Similarly, loops that "read all the lines in a file" or traverse a collection using an iterator cannot be parallelized this way. You would probably start by reading all the lines sequentially into a data structure and then processing them using an OpenMP-friendly loop.

Concurrency Runtime (ConcRT) and Parallel Patterns Library

The Microsoft Concurrency Runtime is a four-piece system that sits between applications and the operating system:

- **PPL (Parallel Patterns Library)** Provides generic, type-safe containers and algorithms for use in your code
- **Asynchronous Agents Library** Provides an actor-based programming model and in-process message passing for lock-free implementation of multiple operations that communicate asynchronously
- **Task Scheduler** Coordinates tasks at run time with work stealing
- **The Resource Manager** Used at run time by the Task Scheduler to assign resources such as cores or memory to workloads as they happen

The PPL feels much like the Standard Library, leveraging templates to simplify constructs such as a parallel loop. It is made dramatically more usable by lambdas, added to C++ in C++11 (although they have been available in Microsoft Visual C++ since the 2010 release).

For example, this sequential loop:

```
for (int i = 1; i < size; ++i)
{
    x[i] = (y[i - 1] + y[i + 1]) / 2;
}
```

can be made into a parallel loop by replacing the *for* with a *parallel_for*:

```
#include <ppl.h>
// . . .
concurrency::parallel_for(1, size, [=](int i)
{
    x[i] = (y[i-1] + y[i+1])/2;
});
```

The third parameter to *parallel_for* is a lambda that holds the old body of the loop. This still requires the developer to know that the loop is parallelizable, but the library bears all the other work. If you are not familiar with lambdas, see the "Lambdas in C++11" section in Chapter 2, "NBody Case Study," for an overview.

A *parallel_for* loop is subject to restrictions: it works with an index variable that is incremented from the start value to one less than the end value (an overload is available that allows incrementing by values other than 1) and doesn't support arbitrary end conditions. These conditions are very similar to those for OpenMP. Loops that test if the square of the loop variable is less than some limit, or that increment by calling a function to get the increment amount, are not parallelizable with *parallel_for*, just as they are not parallelizable with OpenMP.

Other algorithms, *parallel_for_each* and *parallel_invoke*, support other ways of going through a data set. To work with an iterable container, like those in the Standard Library, use *parallel_for_each* with a forward iterator, or for better performance use a random access iterator. The iterations will not happen in a specified order, but each element of the container will be visited. To execute a number of arbitrary actions in parallel, use *parallel_invoke*—for example, passing three lambdas in as arguments.

It's worth mentioning that the Intel Threading Building Blocks (TBB) 3.0 is compatible with PPL, meaning that using PPL will not restrict your code to Microsoft's compiler. TBB offers "semantically compatible interfaces and identical concurrent STL container solutions" so that your code can move to TBB if you should need that option.

Task Parallel Library

The Task Parallel Library is a managed (.NET Framework) approach to parallel development. It provides parallel loops as well as tasks and futures for developers who use C#, F#, or VB. The CLR Thread Pool dispatches and manages threads. Managed developers have other parallel options, including PLINQ.

WARP—Windows Advanced Rasterization Platform

The Direct3D platform supports a driver model in which arbitrary hardware can plug into Microsoft Windows and execute graphics-related code. This is how Windows supports GPUs, from simple graphics tasks, such as rendering a bitmap to the screen, all the way to DirectCompute, which allows fairly arbitrary computations to occur on the GPU. However, this framework also allows for having graphics drivers that are implemented using CPU code. In particular, WARP is a software-only implementation of one such graphics device that is shipped together with the operating system. WARP is capable of

executing both simple graphics tasks—as well as complicated compute tasks—on the CPU. It leverages both multithreading and vectorization in order to efficiently execute Direct3D tasks. WARP is often used when a physical GPU is not available, or for smaller data sets, where WARP often proves to be the more agile solution.

Technologies for GPU Parallelism

OpenGL, the Open Graphics Library, dates back to 1992 and is a specification for a cross-language, cross-platform API to support 2D and 3D graphics. The GPU calculates colors or other information specifically required to draw an image on the screen. OpenCL, the Open Computing Language, is based on OpenGL and provides GPGPU capabilities. It's a language of its own similar in appearance to C. It has types and functionality that are not in C and is missing features that are in C. Using OpenCL does not restrict a developer to deployment on specific video cards or hardware. However, because it does not have a binary standard, you might need to deploy your OpenCL source to be compiled as you go or precompile for a specific target machine. A variety of tools are available to write, compile, test, and debug OpenCL applications.

Direct3D is an umbrella term for a number of technologies, including Direct2D and Direct3D APIs for graphics programming on Windows. It also includes DirectCompute, an API to support GPGPU that is similar to OpenCL. DirectCompute uses a nonmainstream language, HLSL (High Level Shader Language) that looks like C but has significant differences from C. HLSL is widely used in game development and has much the same capabilities as the OpenCL language. Developers can compile and run the HLSL parts of their applications from the sequential parts running on the CPU. As with the rest of the Direct3D family, the interaction between the two parts is done using COM interfaces. Unlike OpenCL, DirectCompute compiles to bytecode, which is hardware portable, meaning you can target more architectures. It is, however, Windows-specific.

CUDA, the Compute Device Unified Architecture, refers to both hardware and the language that can be used to program against it. It is developed by NVIDIA and can be used only when the application will be deployed to a machine with NVIDIA graphics cards. Applications are written in "CUDA C," which is not C but is similar to it. The concepts and capabilities are similar to those of OpenCL and DirectCompute. The language is "higher level" than OpenCL and DirectCompute, providing simpler GPU invocation syntax that is embedded in the language. In addition, it allows you to write code that is shared between the CPU and the GPU. Also, a library of parallel algorithms, called Thrust, takes inspiration from the design of the C++ Standard Library and is aimed at dramatically increasing developer productivity for CUDA developers. CUDA is under active development and continues to gain new capabilities and libraries.

Each of these three approaches to harnessing the power of the GPU has some restrictions and problems. Because OpenCL is cross-platform, cross-hardware (at least in source code form), and cross-language, it is quite complicated. DirectCompute is essentially Windows-only. CUDA is essentially NVIDIA-only. Most important, all three approaches require learning not only a new API and a new way of looking at problems but also an entirely new programming language. Each of the three languages is "C-like" but is not C. Only CUDA is becoming similar to C++; OpenCL and DirectCompute cannot offer C++ abstractions such as type safety and genericity. These restrictions mean that

mainstream developers have generally ignored GPGPU in favor of techniques that are more generally accessible.

Requirements for Successful Parallelism

When writing an application that will leverage heterogeneity, you are of course required to be aware of the deployment target. If the application is designed to run on a wide variety of machines, the machines might not all have video cards that support the workloads you intend to send to them. The target might even be a machine with no access to GPU processing at all. Your code should be able to react to different execution environments and at least work wherever it is deployed, although it might not gain any speedup.

In the early days of GPGPU, floating-point calculations were a challenge. At first, double-precision operations weren't fully available. There were also issues with the accuracy of operations and error-handling in the math libraries. Even today, single-precision floating-point operations are faster than double-precision operations and always will be. It might be necessary to put some effort into establishing what precision your calculations need and whether the GPU can really do those faster than the CPU. In general, GPUs are converging to offer double-precision math and moving toward IEEE 754-compliant math, in addition to the quick-and-dirty math that they have supported in earlier generations of hardware.

It is also important to be aware of the time cost of moving input data to the GPU for processing and retrieving output results from the GPU. If this time cost exceeds the savings from processing the data on the GPU, you have complicated your application for no benefit. A GPU-aware profiler is a must to ensure that actual performance improvement is happening with production quantities of data.

Tool choice is significant for mainstream developers. Past GPGPU applications often had a small corps of users who might have also been the developers. As GPGPU moves into the mainstream, developers who are using the GPU for extra processing are also interacting with regular users. These users make requests for enhancements, want their application to adopt features of new platforms as they are released, and might require changes to the underlying business rules or the calculations that are being performed. The programming model, the development environment, and the debugger must all allow the developer to accommodate these kinds of changes. If you must develop different parts of your application in different tools, if your debugger can handle only the CPU (or only the GPU) parts of your application, or if you don't have a GPU-aware profiler, you will find developing for a heterogeneous environment extraordinarily difficult. Tool sets that are usable for developers who support a single user or who only support themselves as a user are not necessarily usable for developers who support a community of nondeveloper users. What's more, developers who are new to parallel programming are unlikely to write ideally parallelized code on the first try; tools must support an iterative approach so that developers can learn about the performance of their applications and the consequences of their decisions on algorithms and data structures.

Finally, developers everywhere would love to return to the days of the "free lunch." If more hardware gets added to the machine or new kinds of hardware are invented, ideally your code could just

benefit from that without having to change much—or at all. It might even be possible to benefit from improved hardware using the same executable that was deployed to the old hardware and not even need to recompile.

The C++ AMP Approach

C++ AMP is a library and a small language extension that enables heterogeneous computing within a single C++ application. (AMP stands for Accelerated Massive Parallelism.) Visual Studio has new tools and capabilities to support debugging and profiling C++ AMP applications, including GPU debugging and GPU concurrency visualization. With C++ AMP, mainstream C++ developers can use familiar tools to create applications that are portable and future-proof and that can achieve dramatic acceleration for data-parallel-friendly applications.

C++ AMP Brings GPGPU (and More) into the Mainstream

One mission of C++ AMP is to bring GPGPU programming to every developer whose applications can benefit from it. The video cards required to support it are now almost ubiquitous. The overarching mission, however, is larger than just GPGPU: C++ AMP is a way to harness heterogeneous computing platforms, such as GPUs and CPU vector units, and make them accessible to millions of mainstream developers in ways that are not otherwise possible. Although the shift to data-parallel programming—and especially to portable implementations expressed in C++—is an enormous undertaking, it is not the first such transformation that has happened to the software development experience.

Many of the techniques or technologies that change our industry and our world start out in research or academia and are used by only a tiny number of developers who use very specialized tools and are able to do very difficult things. To change the industry and the world, those techniques have to come out to the masses and be considered mainstream. This process has happened with other technologies—GUI interfaces, for example. At first only a few developers had the specialized skills required to work with controls, react to mouse events, and so on. As libraries, frameworks, and tools were developed and released, more and more developers were able to produce GUI applications, and they are now considered the norm. Some libraries, frameworks, and tools are more popular than others, and all contribute to the ecosystem that supports GUI development.

A similar process happened with object-oriented development. At first a few researchers were advocating a new way of designing and building software while the mainstream continued to develop procedural applications. As frameworks and tools have been developed and released, adoption has increased to a point where object-oriented development is considered the norm and used, to varying degrees, by essentially all developers in the majority of mainstream languages.

Such a change might be happening with touch and with natural user interfaces, and it is definitely happening with the concurrency revolution. The first phase was CPU concurrency. The second phase is heterogeneous concurrency. Bringing that ease and normality to heterogeneous computing will require tools, libraries, and frameworks. C++ AMP and Visual Studio are just what mainstream developers need to harness the power of the GPU and beyond.

An interesting possibility is that mainstream developers might find themselves benefiting from C++ AMP without directly using it. If library developers adopt C++ AMP, code that uses those libraries will gain the speedup without having to understand how it was done. The opportunity to create domain-specific libraries could be significant.

C++ AMP Is C++, Not C

There are a number of other approaches to GPGPU development and all of them involve C-like languages. Although C is a powerful and high-performance language, C++ is clearly the number one choice for performance-conscious developers who'd like to work in a modern programming language. C++ provides abstraction and type-safe genericity that enable developers to tackle larger problems and use more powerful libraries and constructs, and these features are available when using C++ AMP, too. You can use templates, overloading, and exceptions in the same way as you do in other parts of your applications.

Because C++ AMP is C++, not C and not a C-like language, the extra types you need for concurrent development are not extensions or additions to the language; they are template types. This gives you type-safe *genericity*—you can distinguish between an array of floats and an array of ints—while reducing your learning curve. Adding abstractions and useful types to C is one of the very problems C++ was designed to solve.

In the past, standard C++ (say, C++11) has supported only CPU programming. The C++ Parallel Patterns Library, PPL, offers a set of types and algorithms in the style of the Standard Library that support multicore development in C++. This lets C++ developers take advantage of new hardware by using the language and tools they are already using. C++ AMP brings that same comfort and convenience to heterogeneous computing.

C++ AMP Leverages Tools You Know

C++ AMP is fully supported by Visual Studio 2012 and will be usable on Windows machines right away. That alone will open the doors to all the developers who use C++ in Visual Studio. These developers will not need to learn a new tool or a new language to start using the power of the GPU. They will have to learn to think in a data-parallel way and to evaluate the costs, calculated in execution time or watts consumed, of their decisions about algorithms and data structures. Using familiar tools will make the overall skills gap one that can be bridged. Visual Studio provides IntelliSense, GPU debugging, profiling, and other features that enable developers to do far more than just write and compile code.

Visual Studio is popular even with developers who aren't targeting Windows. What's more, C++ AMP development is not necessarily restricted to Windows or to Visual Studio users; it has been released as an open specification, and work is underway for other vendors to add C++ AMP to their toolsets. For example, AMD will put it into their FSA reference compiler for Windows and non-Windows platforms.

C++ AMP Is Almost All Library

The key to writing in the language you know is to keep it as the language you know. C++ AMP is an extension to C++ and does include a couple of keywords that are not in C++11. However, it is just two keywords, not a large collection of language changes. Further, the new main keyword, *restrict*, is in use in C99 and is therefore a reserved word, one unlikely to cause collisions with existing codebases. Everything else that makes C++ AMP work involves a library of types and functions. Developers who are comfortable with the Standard Library or with PPL will immediately be comfortable with C++ AMP.

Here's a simple example. Consider this traditional code for adding two vectors. None of this is parallel:

```
void AddArrays(int n, const int* const pA, const int* const pB, int* const pC)
{
    for (int i = 0; i < n; ++i)
    {
        pC[i] = pA[i] + pB[i];
    }
}
```

The preceding code is both easy to read and easy to understand. The following code shows the types of changes that make this operation massively parallel and leverage the GPU:

```
#include <amp.h>
using namespace concurrency;

void AddArrays(int n, const int* const pA, const int* const pB, int* const pC)
{
    array_view<int, 1> a(n, pA);
    array_view<int, 1> b(n, pB);
    array_view<int, 1> c(n, pC);

    parallel_for_each(c.extent, [=](index<1> idx) restrict(amp)
        {
            c[idx] = a[idx] + b[idx];
        });
}
```

As you can see, the code wasn't really changed much. The changes include the following:

1. Including amp.h to use the library
2. Because the types and functions are in the concurrency namespace, adding a *using* statement to reduce your typing
3. Using array views to manage copying the data to or from the accelerator
4. Changing the language for to a library *parallel_for_each* and using a lambda as the last parameter to that function call
5. Using the *restrict(amp)* clause to identify accelerator-compatible code

These are the only changes required. There are no changes to project settings or environment variables. There is no code elsewhere that this needs to call. This is the whole thing.

What happens behind the scenes? One simplified explanation is that the lambda, the kernel that is passed to the *parallel_for_each*, is compiled to HLSL when your application is compiled. The run time for C++ AMP, a DLL that is included with the Visual C++ redistributable package, compiles the HLSL bytecode to hardware-specific machine code at run time. You don't need to know this to use C++ AMP; it is taken care of by the library.

In the code sample just presented, you don't see any code to copy the two input arrays, *pA* and *pB*, to the accelerator or any code to copy the result back into *pC*. The *array_view* objects handle this. An *array_view* is a portable view that works with, and abstracts over, CPU and GPU memories, whether they are colocated on the same chip or are two parts. You can build an *array_view* wrapping a C-style array as in this example or wrapping over a *std::vector*, if that is where your data is.

You may also hint about copy requirements. Consider the following start of a function:

```
void MatrixMultiply(std::vector<float>& C,
    const std::vector<float>& A, const std::vector<float>& B,
    int M, int N, int W)
{
    array_view<const float, 2> a(M, W, A);
    array_view<const float, 2> b(W, N, B);
    array_view<float, 2> c(M, N, C);
    c.discard_data();
```

The first two *array_view* objects specify that they are arrays of const float. This means there is no need to sync them back from the accelerator after the processing is complete—they can take a one-way trip there. Similarly, the third *array_view* is of float, but although it is associated with *C*, the call to *discard_data()* indicates that whatever values happen to be in the memory are not meaningful to anyone, so there is no need to copy the initial values in *C* over to the accelerator. This makes setting up the *array_view* very quick. The results will be copied back from the accelerator when the *array_view* objects are accessed on the CPU or when they go out of scope, whichever happens first.

This hinting needs no new language keywords and can be accomplished just with template overloading. There is no new paradigm for the developer to learn.

The original mathematical logic (such as it is) remains untouched and perfectly readable. There's no mention of polygons, triangles, meshes, vertices, textures, memory, or anything other than adding up matrix elements to get a sum. This is why C++ AMP can make heterogeneous computing mainstream.

The details of the parameters to the *parallel_for_each* and the use of the new *restrict* keyword will be in the case study in the next chapter.

C++ AMP Makes Portable, Future-Proof Executables

Once your code is compiled, the same executable can run on a variety of machines, as long as the machine has a DirectX 11 driver: Windows 7 and later or Windows Server 2008 R2 and later. You are not restricted to a particular vendor or video card family.

When coded appropriately, your application can react to the environment in which it's running and take advantage of whatever acceleration is available. If the machine has hardware with a DX11 driver, it will speed up. Deployment is simply a matter of copying the executable and some dependent dynamic-link libraries (DLLs) (included in the Visual C++ redistributable) to the target machine.

For example, a single executable was written and copied to several different machines. It produces the following output on a virtual machine without access to the GPU:

```
CPU exec time: 112.206 (ms)
No accelerator available
```

And it produces the following output on a machine (more powerful than the laptop hosting the virtual machine) with a typical recent mainstream video card, the NVIDIA GeForce GT 420:

```
CPU exec time: 27.2373 (ms)
GPU exec time including copy-in/out: 19.8738 (ms)
```

This dramatic speed improvement is made possible by a simple query that establishes which accelerators are available:

```
std::vector<accelerator> accelerators = accelerator::get_all();
```

You can then check the returned vector. If it's empty, no accelerators are available. It's a best practice to always ensure that there is an accelerator before trying to execute code that depends on one. Getting into that habit enables your applications to work on a variety of target machines while imposing minimal restrictions on your end users. As a developer with Visual Studio installed, you will always have an accelerator (which might just be an emulator provided for debugging), so forgetting to check at run time for the existence of at least one accelerator could easily lead to the classic "works on my development machine" scenario.

C++ AMP not only makes executables that work on a variety of machines, but it's also designed to be future-proof. In the future, code you write to take advantage of GPU acceleration might be deployed to the cloud and might run over a number of machines, or it could run multithreaded on the CPU only. Heterogeneity in the future will mean more than just CPU+GPU; therefore, C++ AMP is not just a GPU solution, but also a heterogeneous computing solution that supports efficient mapping of data-parallel algorithms to many hardware platforms.

With multicore programming now becoming mainstream, you can leverage 4, 8, or 16 cores on a relatively ordinary computer. With some additional effort, you could also leverage the vector unit on each of these cores (using SSE, AVX, or WARP). GPGPU programming means you can spread your work across hundreds of hardware threads today and even more in the near future. With the cloud, using Infrastructure as a Service (IaaS) or Hardware as a Service (HaaS) offerings, you could conceivably leverage tens of thousands of hardware threads. But imagine being able to combine the two and reach the GPU cores on those cloud machines, reaching tens of millions of hardware threads. What could that enable?

Summary

This chapter provided background about the types of problems for which heterogeneous computing is suited and the history of application performance improvements over the last few decades. It introduced C++ AMP and explained the motivation for the design of C++ AMP. The remainder of this book explains the library and language extensions in more detail, demonstrates how to use more advanced techniques to achieve the maximum performance improvement for your applications, shows how to use the Visual Studio support, and provides guidance for mainstream developers interested in using C++ AMP as a way to harness heterogeneity now.

CHAPTER 2

NBody Case Study

In this chapter:

Prerequisites for Running the Example 21
Running the NBody Example 22
Structure of the Example 28
CPU Calculations 29
The CPU NBody Classes 34
C++ AMP Calculations 36
The C++ AMP NBody Classes 40
Summary 43

This chapter studies a particular example of C++ AMP in action in order to show the concepts that were introduced in the previous chapter and to look ahead to some concepts that will be discussed in the following chapters. The NBody example demonstrates one solution to the n-body problem: how to simulate a large number of stars moving under the influence of gravity. In order to calculate the motion of each star, its interaction with every other star must be calculated, so the amount of computation required scales as n^2, where n is the number of stars. Modern GPUs make it possible to run real-time simulations using many thousands of stars on a single PC, something that was previously possible only on much larger and more expensive hardware. This example is popular in the CUDA community and you might already know it.

Prerequisites for Running the Example

To follow along, you should download the samples source code from *http://ampbook.codeplex.com/* (the NBody sample code is in the NBody folder in the CaseStudies folder) and ensure that you have the following software installed:

- Microsoft Windows 7
- Microsoft Visual Studio 2012
- DirectX SDK, June 2010 release

It's not necessary to have a video card with a DirectX 11 driver to run the example, but to see any benefit of moving calculations to the GPU, you will need such a card. In fact, the "reference accelerator" provided with Visual Studio is much slower than any real accelerator and truly usable only when debugging. A simple way to test which accelerators you have is to use the ShowAmpDevices sample, which you can download from *http://ampbook.codeplex.com/*. It is located in the Samples folder.

More Info You can find more information on DirectX 11 support at *http://www.danielmoth.com/Blog/What-DX-Level-Does-My-Graphics-Card-Support-Does-It-Go-To-11.aspx*. This blog post includes a link to an executable that works much like the sample used here—you can use either one, but the book sample includes the source code if you want to see how it works.

The utility produces output like this:

```
Found 1 accelerator device(s) that are compatible with C++ AMP:
 1: NVIDIA GeForce GT 420, has_display=true, is_emulated=false

Hit enter to exit.
```

If you have no accelerator, you will see confirmation of that:

```
No accelerators found that are compatible with C++ AMP.

Hit enter to exit.
```

You can still run the sample using the reference accelerator, but it will be very slow.

On a Windows 8 machine with Visual Studio 2012 installed, you will at least have an emulator to use as an accelerator:

```
Found 1 accelerator device(s) that are compatible with C++ AMP:
 1: Microsoft Basic Render Driver, has_display=false, is_emulated=true

Hit enter to exit.
```

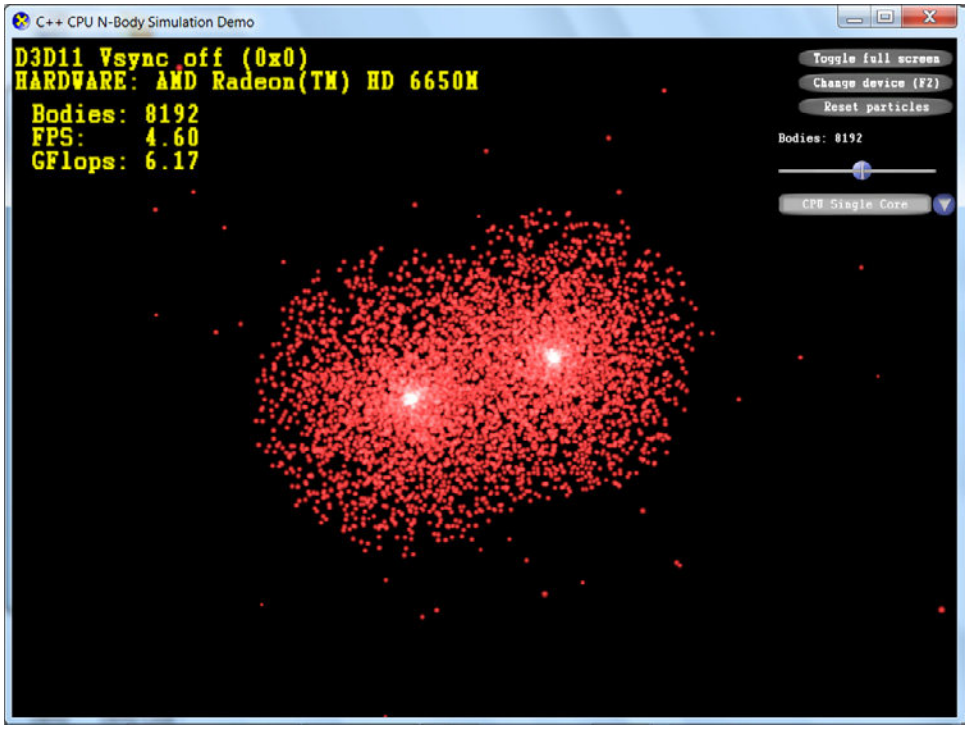
This is the WARP emulator, discussed in more detail later in this book.

If you don't get a response that at least one accelerator was found when you run this utility, troubleshoot your Visual Studio 2012 and DirectX SDK installation—you are unlikely to be able to run the example successfully.

Running the NBody Sample

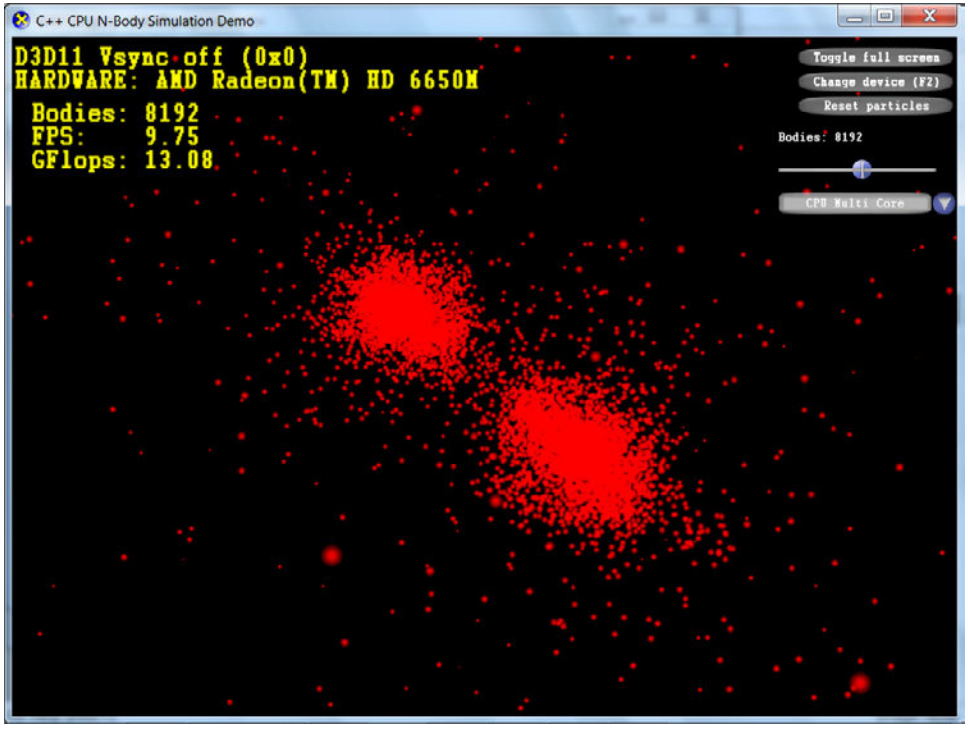
The NBody sample is designed to be run on a machine with a powerful accelerator and corresponding dramatic benefit when using C++ AMP. A fairly mainstream GPU might not produce the same benefit, but you will still see a speedup. There are two separate executables: one that exercises the CPU and one that uses whatever accelerators are on the machine.

To try it yourself, start with the CPU version. Build a Release build and then run it with Ctrl+F5 or by navigating to the newly created executable (NBodyGravityCPU\Release\NBodyGravityCPU.exe) and double-clicking it. You will see a collection of red dots moving around in three dimensions. This is the n-body problem—the movement of each dot (which represents an astronomical body in the original version of the problem) is affected by gravitational effects from each of the other dots. The more bodies there are, the more complex the calculation. Take note of the number of bodies used, the FPS (frames per second), and the GFlops (giga floating-point operations per second) you'll see listed at the upper left of the screen. Make sure the drop-down list under the slider has CPU Single Core selected, as in the following image.

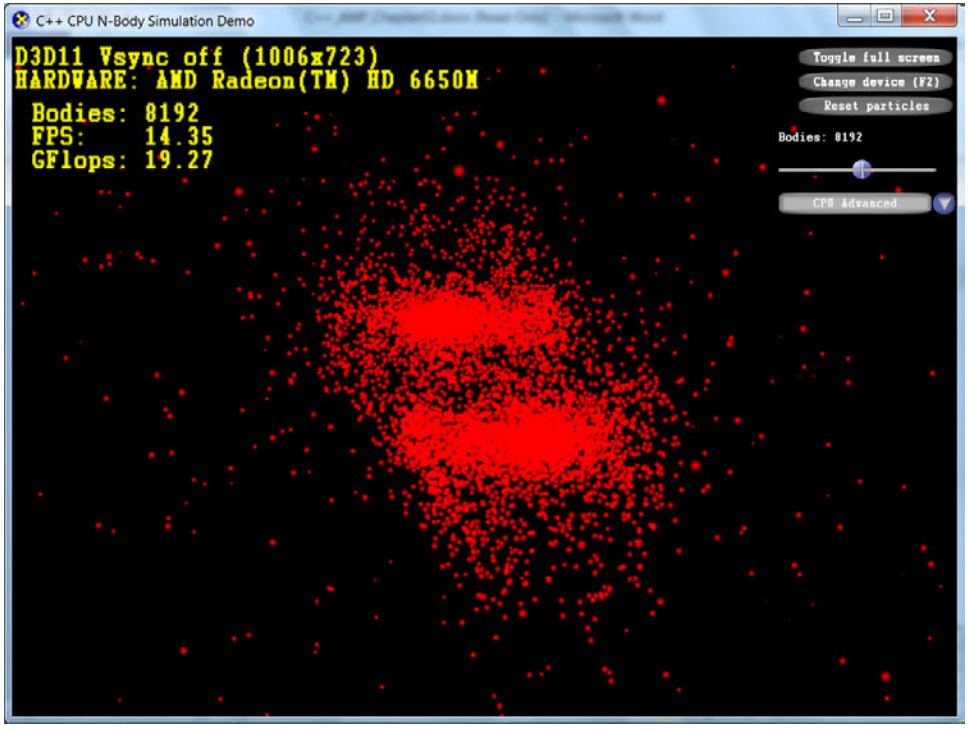


Note Your numbers might not match the values in the images in this section, but the point is to note the change in the numbers as you use different speedup techniques.

Using the slider at the right, increase the number of bodies until the dots noticeably slow down. Reset the particles if you like. Now use the drop-down list below the slider to change the run mode from "CPU Single Core" to "CPU Multi Core." You should notice a color change (to a brighter red) if nothing else:

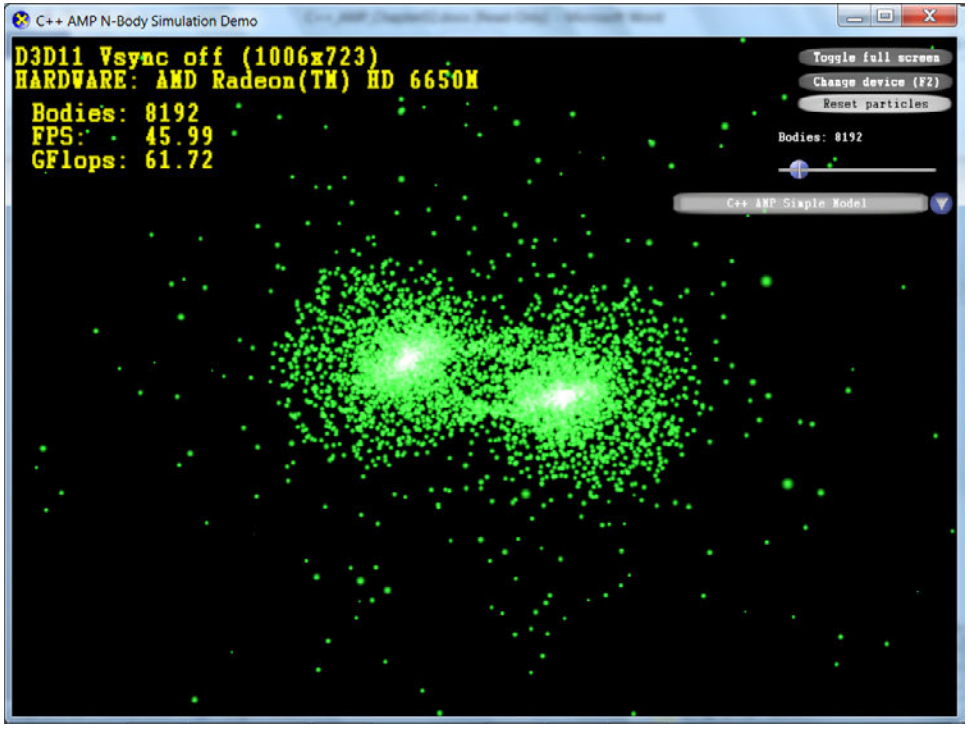


Depending on the number of CPU cores your machine has, you might also notice a speed increase. These images were produced on a four-core i7 and show about a 2× speedup using the multicore capabilities. Next, try setting the drop-down list to "CPU Advanced." This mode uses some CPU-specific optimization to get an even bigger speedup:

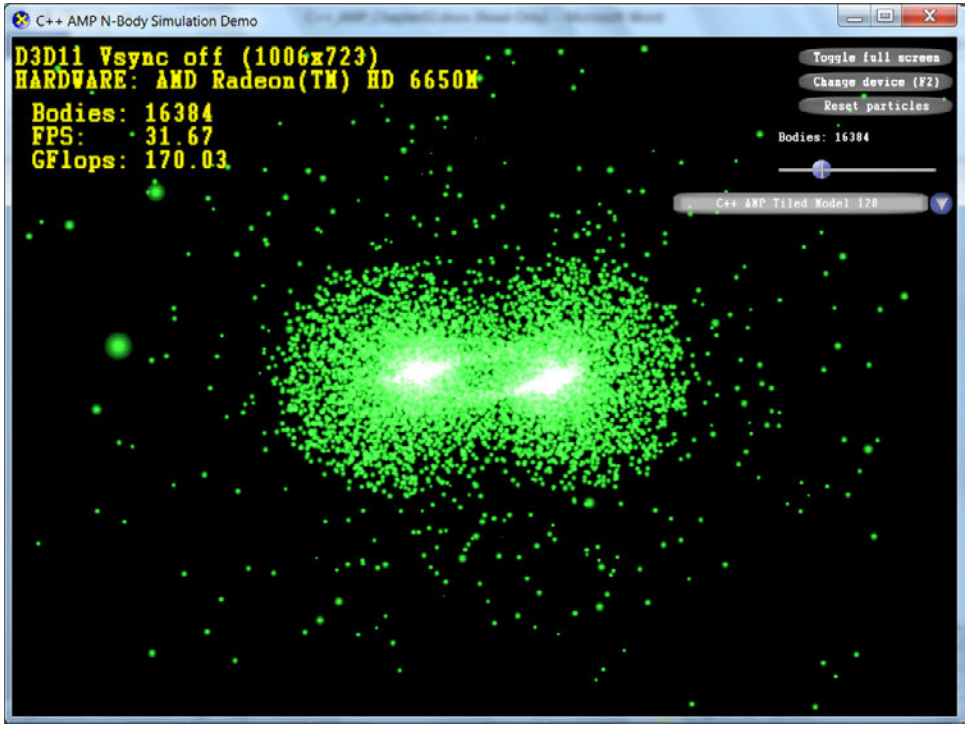


Note Discussion of these optimizations is out of scope for this book. The results are included here to ensure that any comparisons of C++ AMP-accelerated code to CPU-only code are fair.

Make a note of the number of particles you were using and then close the executable. Now, run NBodyGravityAmp\Release\NBodyGravityAmp.exe, which is a very similar application and which uses the same n-body calculations but takes advantage of C++ AMP. When this version starts, the selection in the drop-down box is "C++ AMP Tiled 256" and the particles are green. Change the drop-down selection to "C++ AMP Simple Model." Make sure the number of particles is the same as you used in the previous CPU demonstration. Depending on your video card, this version might be faster or slower than the multi-CPU mode, but it should always be faster than the single-CPU mode (unless you are using the reference accelerator, which is very slow).



If the particles are now moving quickly, drag the slider further to the right until they slow down, and then select "C++ AMP Tiled Model 128" from the drop-down list—just so you can see the improvement.



This chapter won't discuss tiling (the tiled NBody sample is covered in Chapter 5, "Tiled NBody Case Study," but you can certainly see the effect on the frame rate and the calculation speed. With a more substantial video card, such as the ATI Radeon HD 5800, you can achieve over 700 GFlops.

Note If you are fortunate enough to have more than one GPU available, you will see additional options on the drop-down list. Select the "C++ AMP Tiled Model: 2 GPUs" option to see multiple GPUs collaborating in parallel to achieve even higher performance.

Note If you run a Debug version of this code (perhaps out of curiosity), you will notice two things. First, it will be much slower because it runs on a reference accelerator. Lower the number of particles as much as you can when running a Debug build. Debugging in general, and the reference accelerator in particular, are covered in Chapter 6, "Debugging." Second, if you see leak warnings such as **"Detected memory leaks!"** when the application closes, you should know that they are minor and caused by the order in which library code unloads. You can't change your code (or the book samples) to eliminate them, and in any event, these leaks cause no harm.

On a single-GPU machine, such as the one on which these images were produced, the tiled C++ AMP solution achieves many multiples of the GFlops and frame rate compared to the single-CPU mode for the same number of particles. That is a dramatic benefit. But how hard is it to achieve? The rest of this chapter drills into the code to show you how it is done.

Structure of the Example

Like many DirectX samples, this solution uses the DirectX sample framework to handle drawing the user interface. Four main areas of responsibility are in the solution:

- Drawing the UI, including controls such as the drop-downs and buttons in addition to the moving particles
- Reacting to UI changes, such as clicking Reset Particles or dragging the slider to change the number of particles
- Calculating the position and velocity of each particle in each time step using the CPU
- Calculating the position and velocity of each particle in each time step using C++ AMP

The solution is organized into two projects; each project has folders underneath it to organize the code and to make it simpler to navigate. Each project contains both some common code and some that is unique to that project. The organization you see in Solution Explorer is not the same as the way the files are organized on the hard drive—code that appears duplicated in each project, such as resource.h or NBodyGravity.rc, in fact exists in a single file that is included in both projects.

The DXUT and UI folders, also shared by the projects, contain code that is out of scope for this chapter. That code supports the task of getting information onto the screen for the user, but it's not relevant to performing the calculations of the n-body problem or to leveraging C++ AMP. If you're unfamiliar with DirectX, your best approach is to ignore the UI code and concentrate on the position and velocity calculations, which will be described in this chapter. The "Using Graphics Interop" section of Chapter 11, "Graphics Interop," outlines how the NBody sample renders calculation results using DirectX.

First, the CPU calculations, including the simple algorithm for n-body calculations, the data structures, and so on, will be explained, with reference to the files included in the NBodyGravityCPU project. The downloadable sample includes some highly optimized CPU calculations that will not be discussed here. They are included in the sample for a fairer comparison of CPU and GPU performance, but this chapter is focused on explaining the C++ AMP code and how it accelerates the performance of the sample. Next, the NBodyGravityAMP project and the changes required to accelerate this code with C++ AMP will be explained.

CPU Calculations

This section discusses several types of CPU calculations performed in the NBodyGravityAMP simulation.

Data Structures

Each particle in the simulation is represented by an instance of a struct called *ParticleCpu*. As defined in ParticleCpu.h, the struct looks like this:

```
#define SSE_ALIGNMENTBOUNDARY 16

__declspec(align(SSE_ALIGNMENTBOUNDARY))
struct ParticleCpu
{
    float_3 pos;
    float ssePpadding1;
    float_3 vel;
    float ssePpadding2;
    float_3 acc;
    float ssePpadding3;
    float_4 cacheLinePadding;
};
```

Each particle has a position, a velocity, and an acceleration, and each of these is represented by a three-dimensional vector of floats. The struct is padded to the size of a single cache line to improve memory access performance and to limit false sharing of cache lines. The struct is also aligned in memory to allow the SSE instructions to access the structure efficiently.

The particles are kept in a single large vector, allocated in NBodyGravityCPU.cpp:

```
__declspec(align(SSE_ALIGNMENTBOUNDARY))
    std::vector<ParticleCpu>              g_particlesOld(g_maxParticles);
__declspec(align(SSE_ALIGNMENTBOUNDARY))
    std::vector<ParticleCpu>              g_particlesNew(g_maxParticles);
ParticleCpu*                              g_pParticlesOld = &g_particlesOld[0];
ParticleCpu*                              g_pParticlesNew = &g_particlesNew[0];
```

The code does not work directly with *g_particlesOld* but instead goes through *g_pParticlesOld*. The second vector, *g_ParticlesNew*, and its pointer, *g_pParticlesNew*, are used during each calculation step. The *g_ParticlesOld* and *g_pParticlesNew* pointers are used to provide an indirection into the input and output particle arrays and to allow them to be swapped after each time step.

The code in NBodyGravityCPU.cpp is designed to build the UI and deal with user input, such as making a selection in the combo box or using the slider to set the number of bodies. Everything related to actual calculations is done in the classes that implement *INBodyCpu*, which will be discussed shortly. There is one shared pointer to an *INBodyCpu* instance, declared as a global:

```
std::shared_ptr<INBodyCpu>          g_pNBody;   // The current integrator
```

The instance is actually created in *OnD3D11CreateDevice()* and whenever the user changes the calculation type in *OnGUIEvent*, both of which are discussed below. *INBodyCpu*, defined in INBodyCpu.h, is an abstract base class:

```
class INBodyCpu
{
public:
    virtual void Integrate(ParticleCpu* const pParticlesIn,
        ParticleCpu* const pParticlesOut, int numParticles) const = 0;
};
```

The derived classes, *NBodySimpleSingleCore*, *NBodySimpleMultiCore*, and *NBodyAdvanced*, implement the *Integrate()* function and are discussed later in this chapter. Collectively, these are called the NBody classes because they represent the implementations of the *INBodyCpu* abstract base class.

The *wWinMain* Function

In the *NBodyGravityCPU* project, NBodyGravityCpu.cpp contains the entry point for the application in the form of a *wWinMain()* function.

The *wWinMain()* function starts with a series of calls that establish callbacks for the application as a whole. These callback functions are all in NBodyGravityCPU.cpp. Three in particular are worth noting:

```
DXUTSetCallbackFrameMove(OnFrameMove);
// ...
DXUTSetCallbackD3D11DeviceCreated(OnD3D11CreateDevice);
// ...
DXUTSetCallbackD3D11FrameRender(OnD3D11FrameRender);
```

These will each be discussed shortly.

After the callbacks for the application as a whole are in place, *InitApp()* is called to build the individual controls, such as the buttons, drop-down box, and slider, and to set up callbacks for their events.

The *OnFrameMove* Callback

The *OnFrameMove* callback is responsible for updating the scene (moving all the particles) but not for rendering them on the screen. Here is its code:

```
void CALLBACK OnFrameMove(double fTime, float fElapsedTime, void* pUserContext)
{
    g_pNBody->Integrate(g_pParticlesOld, g_pParticlesNew, g_numParticles);

    // Advanced integrator updates particles in place, so no need to swap the buffers.
    if (g_eComputeType != kCpuAdvanced)
        std::swap(g_pParticlesOld, g_pParticlesNew);

    // Update the camera's position based on user input
    g_camera.FrameMove(fElapsedTime);
}
```

The appropriate NBody class's implementation of *Integrate()* calculates the new particle properties (position, velocity, and acceleration) from the old ones. This code first calls *Integrate()* and then swaps the new properties with the old ones to update the position and velocity of every particle.

Note The swap step can be skipped for the Advanced CPU implementation, which is not discussed in this chapter.

The call to *FrameMove* updates the DirectX camera position. The new particle properties will be used to draw this frame in *OnD3D11FrameRender()*.

The key here is the pointer to an instance of the NBody class. It is set up in *OnD3D11CreateDevice()* and updated in *OnGUIEvent()*.

The *OnD3D11CreateDevice* Callback

OnD3D11CreateDevice() is called after the Direct3D device has been created. This typically happens when the application is initialized, but it can also occur when toggling between full-screen and windowed mode. Among a number of calls more relevant to rendering images of the particles on screen are these lines:

```
    // Create NBody object
    g_pNBody = NBodyFactory(g_eComputeType);

    V_RETURN(CreateParticleBuffer(pd3dDevice));
    V_RETURN(CreateParticlePosVeloBuffers(pd3dDevice));
```

NBodyFactory creates the instance of an NBody class used throughout NBodyGravityCPU.cpp:

```
std::shared_ptr<INBodyCpu> NBodyFactory(ComputeType type)
{
    switch (type)
    {
    case kCpuSingle:
        return std::make_shared<NBodySimpleSingleCore>(g_softeningSquared, g_dampingFactor,
            g_deltaTime, g_particleMass);
        break;
    case kCpuMulti:
        return std::make_shared<NBodySimpleMultiCore>(g_softeningSquared, g_dampingFactor,
            g_deltaTime, g_particleMass);
        break;
    case kCpuAdvanced:
        {
            int tileSize = GetLevelOneCacheSize() / sizeof(ParticleCpu);
            return std::make_shared<NBodyAdvanced>(g_softeningSquared, g_dampingFactor,
                g_deltaTime, g_particleMass, tileSize);
        }
        break;
    default:
        assert(false);
        return nullptr;
```

```
        break;
    }
}
```

CreateParticleBuffer() sets up a vertex buffer for Direct3D to use to display the particles. *CreateParticlePosVeloBuffers()* sets up the particles in the *g_particles* array. It also creates some C++ AMP arrays because the shared DirectX render code uses particle information that is already on the GPU to render the particles. Even though the CPU version of this sample doesn't use the GPU for computations, it simplifies the rendering code to have the CPU code put the results into C++ AMP arrays for rendering.

This sample uses a uniform distribution of particles within a sphere as a starting point for the position and velocity of the particles. This is the *LoadParticles()* method:

```
void LoadParticles()
{
    const float centerSpread = g_Spread * 0.50f;
    for(size_t i = 0; i < g_maxParticles; i += g_particleNumStepSize)
    {
        LoadClusterParticles(&g_pParticlesOld[i],
            float_3(centerSpread, 0.0f, 0.0f),
            float_3( 0, 0, -20),
            g_Spread,
            g_particleNumStepSize / 2);
        LoadClusterParticles( &g_pParticlesOld[i + g_particleNumStepSize / 2],
            float_3(-centerSpread, 0.0f, 0.0f),
            float_3( 0, 0, 20),
            g_Spread,
            (g_particleNumStepSize + 1) / 2);
    }
}
```

This code loads two "clumps" of particles (the two clouds you see at the start of each run or when you click the Reset Particles button). Each pass through the loop adds 128 particles to each clump because *g_particleNumStepSize* is defined to be 256. The *LoadClusterParticles()* function sets the positions of each particle randomly within a specific radius:

```
void LoadClusterParticles(ParticleCpu* const pParticles, float_3 center, float_3 velocity,
    float spread, int numParticles)
{
    std::random_device rd;
    std::default_random_engine engine(rd());
    std::uniform_real_distribution<float> randRadius(0.0f, spread);
    std::uniform_real_distribution<float> randTheta(-1.0f, 1.0f);
    std::uniform_real_distribution<float> randPhi(0.0f, 2.0f * static_cast<float>(std::_Pi));

    std::for_each(pParticles, pParticles + numParticles,
        [=, &engine, &randRadius, &randTheta, &randPhi](ParticleCpu& p)
    {
        float_3 delta = PolarToCartesian(randRadius(engine),
            acos(randTheta(engine)), randPhi(engine));
```

```
        p.pos = center + delta;
        p.vel = velocity;
        p.acc = 0.0f;
    });
}
```

Because *g_Spread* is set to 400.0 in NBodyGravityCPU.cpp, the two clumps of particles are centered around (200.0, 0.0, 0.0) and (-200, 0.0, 0.0) and should end up in a sphere about 400 units across. They will all have the same initial velocity, which will change over time due to the gravitational effects of other particles.

The OnGUIEvent Callback

The pointer to the NBody class is initialized in *OnD3D11CreateDevice* and based on the compute type, which defaults to CPU Advanced. The user can change the compute type with the drop-down list. In *OnGUIEvent()* you can see the code that sets it:

```
case IDC_COMPUTETYPECOMBO:
    {
        CDXUTComboBox* pComboBox = static_cast<CDXUTComboBox*>(pControl);
        g_eComputeType = static_cast<ComputeType>(pComboBox->GetSelectedIndex());
        g_particleColor = g_particleColors[g_eComputeType];
        g_pNBody = NBodyFactory(g_eComputeType);

        WCHAR szTemp[256];
        swprintf_s(szTemp, L"Bodies: %d", g_numParticles);
        g_HUD.GetStatic(IDC_NBODIES_LABEL)->SetText(szTemp);
        g_FpsStatistics.clear();
    }
    break;
```

The code is not C++ AMP-specific but just part of the UI of this sample. It sets the compute type, the *g_pNBody* pointer, and the particle color, and it updates some of the statistics that appear in the top left of the screen.

The OnD3D11FrameRender Callback

Most of the work done by this function is out of scope for this discussion, but it is worth noting that this is how the particles end up on the screen:

```
RenderParticles(pd3dImmediateContext, view, projection);
```

RenderParticles() uses the array view and arrays set up in *CreateParticlePosVeloBuffers()*. It also sets the properties of the D3D Device Context and connects the shaders.

With that, you can ignore the rest of NBodyGravityCPU.cpp and concentrate on the NBody classes.

The CPU NBody Classes

The classes that implement *INBodyCpu* are at the heart of the application. Each class has an implementation of *Integrate()* that performs the same calculations using different computational approaches. These classes are all defined in NBodyCpu.cpp. Two of the classes, *NBodySimpleSingleCore* and *NBodySimpleMultiCore*, have a private member variable representing an engine for the calculation:

```
private:
    std::shared_ptr<NBodySimpleInteractionEngine> m_engine;
```

The third derived class, *NBodyAdvanced*, uses an *NBodyAdvancedInteractionEngine* that won't be discussed here. It ensures that the CPU version is not unnecessarily naive and simple. Hand-optimized algorithms for the CPU are often not data-parallel; the appropriate way to compare approaches is to start with the simple one and then optimize two ways, once for CPU execution and once for GPU, and to compare those. Comparing a very simple implementation on the CPU and that same implementation on the GPU is not realistic. In real applications, the code for the CPU has been optimized as much as possible, and this sample uses SSE and SSE4 when possible. Even though that code will not be shown or explained, it is in the sample for you to read.

NBodySimpleInteractionEngine

This class, used by both simple CPU approaches, has a constructor and one public function, *InvokeBodyBodyInteraction()*. A private function, *SelectCpuImplementation*, sets a function pointer to one of three possible implementations, depending on the SSE support on the computer where the code is running:

```
void NBodySimpleInteractionEngine::SelectCpuImplementation()
{
    switch (GetSSEType())
    {
    case kCpuSSE4:
        m_funcptr = &NBodySimpleInteractionEngine::BodyBodyInteractionSSE4;
        break;
    case kCpuSSE:
        m_funcptr = &NBodySimpleInteractionEngine::BodyBodyInteractionSSE;
        break;
    default:
        m_funcptr = &NBodySimpleInteractionEngine::BodyBodyInteraction;
    }
}
```

After this function pointer is set, the two simple derived classes can use the engine.

NBodySimpleSingleCore

The single-core CPU algorithm uses the interaction engine to find the new velocity and position of each particle:

```
void NBodySimpleSingleCore::Integrate(ParticleCpu* const pParticlesIn,
    ParticleCpu* const pParticlesOut, int numParticles) const
{
    for (int i = 0; i < numParticles; ++i)
    {
        pParticlesOut[i] = pParticlesIn[i];
        m_engine->InvokeBodyBodyInteraction(pParticlesIn, pParticlesOut[i], numParticles);
    }
}
```

It simply loops through all the particles one at a time, calculating their new position and velocity with the help of the interaction engine.

NBodySimpleMultiCore

An easy way to improve the performance of the "loop through all the particles" implementation is to use the Parallel Patterns Library (PPL) to process multiple particles at once. One of the joys of using PPL is how little you need to change your code to use multiple CPU cores. The for loop of *NBodySimpleSingleCore::Integrate()* is wrapped in a *parallel_for()* from the concurrency namespace, and the third parameter to that function is a lambda that incorporates the code from the original for loop:

```
void NBodySimpleMultiCore::Integrate(ParticleCpu* const pParticlesIn,
    ParticleCpu* const pParticlesOut, int numParticles) const
{
    parallel_for(0, numParticles, [=, this, &pParticlesOut](int i)
    {
        pParticlesOut[i] = pParticlesIn[i];
        m_engine->InvokeBodyBodyInteraction(pParticlesIn, pParticlesOut[i], numParticles);
    });
}
```

This code is thread-safe because all threads read from a read-only copy of the particles stored in *pParticlesIn*, and only one thread writes to a given array element in *pParticlesOut*. Ensuring that the *ParticleCpu* struct is aligned and occupies a whole cache line reduces the amount of false cache line sharing and improves performance.

NBodySimpleInteractionEngine::BodyBodyInteraction

The SSE versions of the algorithm are a bit harder to read than the simple one, and it's not necessary that you understand them to use C++ AMP, so they won't be presented here. In contrast, the non-SSE version is quite readable:

```
void NBodySimpleInteractionEngine::BodyBodyInteraction(const ParticleCpu* const pParticlesIn,
    ParticleCpu& particleOut, int numParticles) const
```

```
{
    float_3 pos(particleOut.pos);
    float_3 vel(particleOut.vel);
    float_3 acc(0.0f);

    std::for_each(pParticlesIn, pParticlesIn + numParticles, [=, &acc](const ParticleCpu& p)
    {
        const float_3 r = p.pos - pos;

        float distSqr = SqrLength(r) + m_softeningSquared;
        float invDist = 1.0f / sqrt(distSqr);
        float invDistCube =  invDist * invDist * invDist;
        float s = m_particleMass * invDistCube;

        acc = r * s;
    });

    vel += acc * m_deltaTime;
    vel *= m_dampingFactor;
    pos += vel * m_deltaTime;

    particleOut.pos = pos;
    particleOut.vel = vel;
}
```

This function, which you might recall is called once for each particle, loops through all the particles (so overall we are on the order of n^2 passes through the algorithm) and calculates the contribution of each of the other bodies to the acceleration of this body. There are two "fudge factors" in the equation: *softening*, which increases the effective distance between two particles and therefore decreases the inverse cube of that distance and the acceleration, and *dampening*, which could further decrease the calculated velocities but which is set to 1.0 in this sample.

Because *particleOut* is passed by reference, this function changes the particle values in *g_pParticlesNew*, which then gets swapped with *g_pParticlesOld* in *OnFrameMove()*.

NBodySimpleInteractionEngine::BodyBodyInteractionSSE() and *NBodySimpleInteractionEngine::BodyBodyInteractionSSE4()* make the same calculations but use SSE to speed up the process. Discussion of those details is out of scope for this chapter. They certainly serve as good examples of how using SSE hurts code readability, even when comments preceding most lines show the equivalent straight-ahead code.

C++ AMP Calculations

The *NBodyGravityAMP* project overlaps significantly with the *NBodyGravityCPU* project, so only those aspects that are different will be discussed here. These include a different set of classes that inherit from an *INBodyAmp* abstract class and the code to set up the accelerators and the C++ AMP-specific data structures.

The entry point is *wWinMain*, as it is in the NBodyGravityCPU project, and these two functions are identical except for the title of the application window: "C++ CPU N-Body Simulation Demo" or "C++ AMP N-Body Simulation Demo." *OnFrameMove* calls *Integrate()* and swaps old and new particle information, as in the CPU version. *OnD3D11CreateDevice* calls *CreateParticlePosBuffer()* instead of *CreateParticlePosVeloBuffers()*. One of the important differences between these two is that the C++ AMP version sets up the accelerators to be used:

```
    accelerator_view renderView =
        concurrency::direct3d::create_accelerator_view(reinterpret_cast<IUnknown*>(pd3dDevice));
    g_deviceData = CreateTasks(g_maxParticles, renderView);
```

The *CreateTasks()* function is discussed below.

Data Structures

The particle type *ParticlesCpu* is used in the C++ AMP-accelerated calculations. However, rather than an array of structs, a class holding individual arrays is used for better performance. Specifically, *LoadParticles()* in NBodyGravityAmp.cpp works with this local variable:

```
ParticlesCpu particles(g_maxParticles);
```

And ParticlesCpu is declared in NBodyAmp.h and defined like this:

```
class ParticlesCpu
{
public:
    std::vector<float_3> pos;
    std::vector<float_3> vel;

    ParticlesCpu(int size) : pos(size), vel(size) { }

    inline int size() const
    {
        assert(pos.size() == vel.size());
        return static_cast<int>(pos.size());
    }
};
```

A similar class holds the *concurrency::array* instances, *ParticlesAmp*:

```
class ParticlesAmp
{
public:
    array<float_3, 1>& pos;
    array<float_3, 1>& vel;

public:
    ParticlesAmp(array<float_3, 1>& pos, array<float_3, 1>& vel) : pos(pos), vel(vel) { }

    inline int size() const { return pos.get_extent().size(); }
};
```

The data on a device (such as a GPU) is represented by the *TaskData* struct:

```
struct TaskData
{
public:
    accelerator Accelerator;
    std::shared_ptr<ParticlesAmp> DataOld;
    std::shared_ptr<ParticlesAmp> DataNew;

private:
    array<float_3, 1> m_posOld;
    array<float_3, 1> m_posNew;
    array<float_3, 1> m_velOld;
    array<float_3, 1> m_velNew;

public:
    TaskData(int size, accelerator_view view, accelerator acc) :
        Accelerator(acc),
        m_posOld(size, view),
        m_velOld(size, view),
        m_posNew(size, view),
        m_velNew(size, view),
        DataOld(new ParticlesAmp(m_posOld, m_velOld)),
        DataNew(new ParticlesAmp(m_posNew, m_velNew))
    {
    }
};
```

The C++ AMP versions of *Integrate* work with one or more of these *TaskData* structs, as you'll see later in this chapter. In general, for GPU programming, structs of arrays are more efficient than arrays of structs.

CreateTasks

This function checks for the existence of accelerators and builds an array of nonemulated, non-CPU ones. It also sets up a vector of shared pointers to *TaskData* structs for the accelerators to work with.

```
std::vector<std::shared_ptr<TaskData>> CreateTasks(int numParticles,
    accelerator_view renderView)
{
    std::vector<accelerator> gpuAccelerators = AmpUtils::GetGpuAccelerators();
    std::vector<std::shared_ptr<TaskData>> tasks;
    tasks.reserve(gpuAccelerators.size());

    if (!gpuAccelerators.empty())
    {
        //  Create first accelerator attached to main view. This will attach the C++ AMP
        //  array<float_3> to the D3D buffer on the first GPU.
        tasks.push_back(std::make_shared<TaskData>(numParticles, renderView,
            gpuAccelerators[0]));

        //  All other GPUs are associated with their default view.
        std::for_each(gpuAccelerators.cbegin() + 1, gpuAccelerators.cend(),
            [=, &tasks](const accelerator& d)
```

```
        {
            tasks.push_back(std::make_shared<TaskData>(numParticles, d.default_view, d));
        });
    }

    if (tasks.empty())
    {
        OutputDebugStringW(L"WARNING: No C++ AMP capable accelerators available, using REF.");
        accelerator a = accelerator(accelerator::default_accelerator);
        tasks.push_back(std::make_shared<TaskData>(numParticles, renderView, a));
    }

    AmpUtils::DebugListAccelerators(gpuAccelerators);
    return tasks;
}
```

The helper function *GetGpuAccelerators()* just checks if the accelerator is a GPU device, as opposed to an emulator or a CPU-accelerator:

```
static inline std::vector<concurrency::accelerator> GetGpuAccelerators()
{
    return GetAccelerators(IsAmpAccelerator(false));
}
```

This code is in AmpUtilities.h, where you will also find *GetAccelerators()*:

```
template<typename Func>
static std::vector<concurrency::accelerator> GetAccelerators(Func filter)
{
    std::vector<accelerator> accls = accelerator::get_all();
    accls.erase(std::remove_if(accls.begin(), accls.end(), filter), accls.end());
    return accls;
}
```

This code gets all the accelerators and then removes those meeting the criteria passed in—for example, those that are not GPU accelerators. For completeness, here is *IsAmpAccelerator*:

```
class IsAmpAccelerator
{
private:
    bool m_includeWarp;

public:
    IsAmpAccelerator(bool includeWarp) : m_includeWarp(includeWarp) {}

    bool operator() (const concurrency::accelerator& a)
    {
        return (a.is_emulated ||
            ((a.device_path.compare(concurrency::accelerator::direct3d_warp) == 0)
             && !m_includeWarp));
    }
};
```

This little function object just wraps up the code to establish whether a particular accelerator is hardware or emulated, with the added complication that you might want a WARP accelerator to be considered as an AMP accelerator, or you might not.

The C++ AMP NBody Classes

Like the CPU hierarchy, there are three classes that derive from *INBodyAmp: NBodyAmpSimple*, *NBodyAmpTiled*, and *NBodyAmpMultiTiled*. The base class, INBodyAmp, is essentially INBodyCpu with an extra function to provide the tile size:

```
class INBodyAmp
{
public:
    virtual int TileSize() const = 0;
    virtual void Integrate(const std::vector<std::shared_ptr<TaskData>>& particleData,
        int numParticles) const = 0;
};
```

The classes for tiled calculations are actually templates that take the tile size as a parameter. Tiling will be discussed in a later chapter; this chapter will discuss only the simple algorithm.

NBodyAmpSimple::Integrate

The *Integrate()* implementation sets up data structures and a *parallel_for_each* to perform the same calculations as the CPU version of Integrate:

```
    void Integrate(const std::vector<std::shared_ptr<TaskData>>& particleData,
        int numParticles) const
    {
        assert(numParticles > 0);
        assert((numParticles % 4) == 0);

        ParticlesAmp particlesIn = *particleData[0]->DataOld;
        ParticlesAmp particlesOut = *particleData[0]->DataNew;

        extent<1> computeDomain(numParticles);
        const float softeningSquared = m_softeningSquared;
        const float dampingFactor = m_dampingFactor;
        const float deltaTime = m_deltaTime;
        const float particleMass = m_particleMass;

        parallel_for_each(computeDomain, [=] (index<1> idx) restrict(amp)
        {
            float_3 pos = particlesIn.pos[idx];
            float_3 vel = particlesIn.vel[idx];
            float_3 acc = 0.0f;

            // Update current Particle using all other particles
            for (int j = 0; j < numParticles; ++j)
```

```
            BodyBodyInteraction(acc, pos, particlesIn.pos[j], softeningSquared,
                particleMass);

            vel += acc * deltaTime;
            vel *= dampingFactor;
            pos += vel * deltaTime;

            particlesOut.pos[idx] = pos;
            particlesOut.vel[idx] = vel;
        });
    }
```

This implementation illustrates three important AMP concepts working together: *array*, *extent*, and *index*. All are one-dimensional in this sample. Here is how they relate:

- The particle information is on the accelerator in the *ParticlesAmp* instances, each of which holds *concurrency::array* objects—one of position and one of velocity. These were set up in *LoadParticles* to hold *numParticles* particles.
- To shape the *parallel_for_each*, use a one-dimensional *extent*, called *computeDomain*.
- The size of *computeDomain* is *numParticles*. If it were a two-dimensional extent, it would be declared as *extent<2>* and the constructor would be passed two integers representing the most significant and least significant dimensions, in that order.
- The *parallel_for_each* takes two parameters: the *extent* and a lambda. It's the lambda that represents the repetitive calculations to be done on the accelerator. The only parameter it takes is an *index*, which points to a specific place in the grid where the calculation is to be done.
- The lambda captures variables it needs from the stack by value and performs essentially the same calculation as *NBodySimpleInteractionEngine::BodyBodyInteraction()*. It is marked *restrict(amp)* because it is to be parallelized onto an accelerator, such as the GPU. The contents of the *for* loop, adding up the contributions of each other body to the acceleration of this body, are in a helper function.

BodyBodyInteraction

This function calculates the contribution of another particle to this particle's acceleration:

```
void BodyBodyInteraction(float_3& acc, const float_3 particlePosition,
    const float_3 otherParticlePosition,
    float softeningSquared, float particleMass) restrict(amp)
{
    float_3 r = otherParticlePosition - particlePosition;

    float distSqr = SqrLength(r) + softeningSquared;
    float invDist = concurrency::fast_math::rsqrt(distSqr);
    float invDistCube =  invDist * invDist * invDist;
```

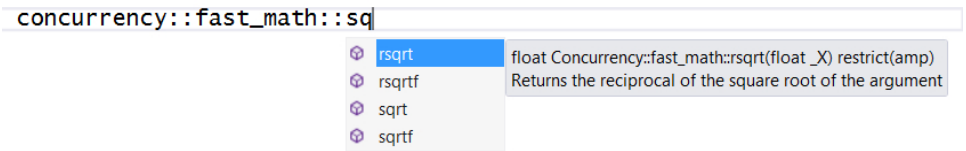
```
    float s = particleMass * invDistCube;

    acc += r * s;
}

inline const float SqrLength(const float_3& r) restrict(amp, cpu)
{
    return r.x * r.x + r.y * r.y + r.z * r.z;
}
```

Notice that this function and the *SqrLength()* function both have *restrict* clauses specifying that they run on an accelerator, as the lambda does. However, *NBodyAmpSimple::Integrate()* does not.

The actual implementation of the calculations reflects the GPU nature of the code. For example, it uses concurrency::*fast_math::rsqrt()* instead of 1.0/*sqrt()*. In many cases amp.h and its companion header amp_math.h provide GPU-friendly functions that you can use on the GPU. Typically, they are one-line wrappers, calling a function with a far more awkward name, such as *__dp_d3d_rsqrtf* in the case of *rsqrt*.

Discovering how to translate your calculations into GPU-appropriate equivalents is where the real work occurs when adapting a working calculation to use the power of C++ AMP. Using *sqrt()* here doesn't cause a compile error or a run-time error because there is a GPU-friendly version of that in amp.h also, wrapping a call to *__dp_math_sqrtf()*.

The name *fast_math* reminds you that this function is faster (but less precise) than the *rqsrt* in the *precise_math* namespace, which requires double-precision support. You can't count on double-precision on the GPU. More details about double-precision support are the in "Double-Precision Support" section of Chapter 12, "Tips, Tricks, and Best Practices." It's a very good idea to look through amp_math.h for these GPU versions of mathematical functions that you might be using in your algorithm. There are too many to list; you can find them in the namespace *fast_math*, and IntelliSense is a good way to learn about each of them:



Everything else in the NBodyGravityAMP project—building the user interface, drawing the particles on the screen, and so on—is the same as in the NBodyGravityCPU project. The tiling-related calculation will be discussed in a later chapter.

Summary

This sample shows the kinds of acceleration that C++ AMP can achieve in a real application. Some of the realistic features of this sample include:

- It has a graphical interface that exerts its own load on the GPU, unlike a console application that can have the GPU all to itself.
- The CPU cases used for comparison have been accelerated with SSE and SSE4 if available on the machine.
- The possibility of CPU acceleration with PPL is considered. On many developer machines with four or more cores and a low-end video card, multicore might beat manycore in this sample. The same might not hold true on your target machines, of course—and adding GPU power is cheaper than adding CPU power.

This example exhibits some interesting optimizations, most notably skipping the copy from GPU to CPU and back to GPU again in the single GPU case. You need to consider these sorts of things when you design your own speedups with C++ AMP.

The differences between the CPU code and the C++ AMP code are quite small. Obviously, the C++ AMP code includes a *parallel_for_each* that spreads the work of calculating each particle's new position and velocity across the many cores of the GPU (or other accelerator). The CPU calculation uses standard vectors, while the C++ AMP calculation uses *concurrency::array* objects. Some of the functions used in the CPU approach are replaced in the AMP approach with the Direct3D equivalents. These changes are not time-consuming to implement and produce a dramatic speedup—easily a 10× speedup from the simplest CPU approach and 25 percent faster than the highly tuned CPU approach—on an ordinary video card. In future chapters, you will see how to use tiling to achieve even larger speedups and how to take advantage of multiple GPUs as well.

CHAPTER 3

C++ AMP Fundamentals

In this chapter:

array<T,N> 45

accelerator **and** ***accelerator_view*** 48

index 50

extent 50

array_view<T,N>. 51

parallel_for_each 55

Functions Marked with ***restrict(amp)*** 57

Copying Between CPU and GPU 59

Math Library Functions. 61

Summary 62

In Chapter 1 of this book "Overview and C++ AMP Approach," you saw some very simple C++ AMP code to add and multiply matrices. In Chapter 2, "NBody Case Study," you saw a working case study of C++ AMP in action. Before drilling down further into details such as tiling, using multiple accelerators, or effective ways to "mix and match" CPU and GPU acceleration, it's a good idea to review the fundamentals of C++ AMP and ensure that you're clear on all the concepts. C++ AMP consists almost entirely of library code. The two language changes are the *restrict* keyword, discussed in this chapter, and the *tile_static* keyword, discussed in Chapter 4, "Tiling." Everything else is provided through the amp.h header, primarily in the form of templates.

array<T, N>

The first type that matters to you as a C++ AMP developer is the *array*. This template is in the concurrency namespace. It takes two template parameters: the type that is in the collection and the rank, or number of dimensions. Most work will involve one-dimensional, two-dimensional, or three-dimensional arrays, but C++ AMP doesn't restrict you to three; if you have a need for more dimensions, you can have up to 128!

Under normal use, an *array* means a collection of information, all of the same type, located on an accelerator, normally a GPU. It's actually located on an *accelerator_view*: every accelerator has at least one such view. Accelerators and their views are discussed in the next section. There is a default accelerator and every accelerator has a default view, so when you simply create an array, that's where it lives. For example:

```
array<int,1> a(5);
```

This line of code declares a one-dimensional array of *ints* five elements long. In general, you set the size of an array with an extent, which will be discussed shortly, but some convenience overloads are available for one, two, and three dimensions:

```
array<float, 2> b(4, 2);
array<int, 3> c(4, 3, 2);
```

None of the three arrays just declared contain any values; these constructors create empty arrays. You can write to them later. Or, if you prefer, you can create an array and copy data into it at the same time:

```
array<int, 1> a(5, v.begin(), v.end());
```

In the preceding example, *v* is some container that exposes iterators (a *std::vector* would be fine, but there are plenty of others, including raw C-style arrays if you use *begin(v)* and *end(v)* instead of the member functions *begin()* and *end()*). Constructing the array in this way copies the data synchronously from *v* into *a* and thus onto the accelerator. The third parameter is optional; you could also construct this array like this:

```
array<float, 1> a(5, v.begin());
```

An *array* has a guaranteed memory layout; all the elements are in sequence in a contiguous block of memory. Two elements whose least significant index differs by one lie next to each other in memory. The least significant dimension is the last one when declaring the array.

To get the data back from the *array*, you must explicitly copy it:

```
copy(a, v);
```

A number of overloads of the *copy()* method are in the concurrency namespace that copy to or from an *array*, such as *array_view* (described later in this chapter), and various containers, such as *std::vector*. All these are synchronous, although there are async versions you can use if you prefer. IntelliSense or amp.h are the best ways to discover all the options.

Arrays are bound to a particular accelerator view. If you have only one accelerator on your system, you don't have a lot of choices for which accelerator to use, but you might still want to use a specific view. If your system has more than one accelerator and you want to specify the accelerator on which your code will run, you can use an *accelerator_view* to specify where the array is to be created:

```
array<float, 1> m(n, v.begin(), av);
```

Note How to get an *accelerator_view* and how to declare and initialize the *av* variable used above is discussed in the next section.

Here's a summary of the various constructors for *array*. There are a great number of overloads, but all of them are variants on this example:

```
array<float, 1> m(e, v.begin(), v.end(), av);
```

This example takes an *extent* (*e*), an iterator into the source container from which to copy (*v.begin*), another iterator (*v.end*) that marks the end of the data to be copied, and an accelerator view (*av*). From here the following variations are possible. The *extent* type is discussed in a later section; here it is used to define the dimensions of the *array*. You can mix and match as you like:

- Every constructor that takes an *extent* as the first parameter has three equivalent constructors that take one, two, or three integers (to build an *extent* of the appropriate rank) followed by the same parameters as the constructor that takes an *extent*.
- Every constructor that takes a pair of iterators has an equivalent constructor that takes only one iterator and copies the number of elements specified by the *extent* (be careful: there is no check that enough elements are available from the source), and another that takes no iterators and does not copy data into the array.
- Every constructor that takes an *accelerator_view* has an equivalent constructor that does not take an *accelerator_view*; these use the default view.
- Every constructor that takes an *accelerator_view* has an equivalent constructor that takes another *accelerator_view*, which is used to create staging arrays. Staging arrays are covered in the "Using Staging Arrays" section of Chapter 7, "Optimization."

These combinations produce 48 constructors for *array*. In addition, there is a constructor to create an array on a specific *accelerator_view* from an *array_view* and to copy the contents. One variant of this constructor does not take an *accelerator_view* (it uses the default *accelerator_view*), and another takes two for a staging array. That brings the total to 51. There is also a copy constructor (which does a deep copy), a move constructor, and various copy and move assignment operators, but you can ignore those. IntelliSense in Microsoft Visual Studio is the simplest way to keep it all straight and to make sure you are using the appropriate parameters to construct and fill an *array*.

After you have created an *array*, you will want to perform some calculations with it, probably using a *parallel_for_each*. Before you can write that code, there are some other classes and concepts you should understand.

accelerator and *accelerator_view*

Most of the time, when you see "accelerator," you can think "GPU," but the GPU is not the only accelerator and not all video cards have a GPU that can be used as a C++ AMP accelerator (for example, the card might not support DirectX 11). The object in the concurrency namespace, *accelerator*, represents not only a GPU but also possibly a virtual accelerator such as the emulator installed with Visual Studio or WARP (a CPU-side accelerator implemented using multicore and SSE instructions). It has memory that can hold one or more arrays, it can perform calculations on those arrays, and it is optimized for data-parallel computing.

The *accelerator::get_all()* function will return a vector of accelerators at run time, so that you could take different code paths depending on the configuration of the machine on which your code is executing. You can check the properties of an accelerator, for example, to discover whether it is an emulator or the CPU so you can make a decision more refined than simply whether an accelerator is present or not. You can also query its capabilities, such as whether an accelerator supports double precision. Some useful constants are defined that you can pass to constructors or use in comparisons:

- *accelerator::default_accelerator*
- *accelerator::direct3d_warp*
- *accelerator::direct3d_ref*
- *accelerator::cpu_accelerator*

The default accelerator is the best one available, chosen at run time. If you have one hardware accelerator, say a GPU, and the reference accelerator, the default will be the GPU. On the other hand, on a machine without an appropriate video card, the default might be the reference accelerator (which is very slow and will not actually accelerate your application but can still be useful for debugging purposes). On a Microsoft Windows 8 machine, you won't have to resort to the reference accelerator because you will have WARP, which can actually accelerate the application.

An accelerator is usually a physical device. It's possible to have several logical views of this device. Those views are isolated from one another. An accelerator is an isolated resource and execution context. You can choose to have threads share a view, or you can use separate views on the same accelerator to eliminate sharing concerns. This is why, for example, there are *array* constructors that take an *accelerator_view*. Every accelerator has a default view, so if all you are thinking is, "I want this array on that particular accelerator," you can pass the accelerator's default view to the constructor:

```
accelerator device(accelerator::default_accelerator);
accelerator_view av = device.default_view;
array<float, 1> C(n, av);
```

Of course, those three lines of code accomplish the same thing as this single line:

```
array<float, 1> C(n);
```

That code will create the array on the default view of the default accelerator. To keep the code samples shorter and more readable, most of the sample code in this book uses this approach. Production code will often create and use an *accelerator_view* to gain more control over execution and error-handling. If you want your application to gracefully handle exceptions related to Time-Out Detection and Recovery (TDR), then you must always create a unique *accelerator_view* and execute your code on this logical view. This makes it possible to handle TDR-related exceptions and to rerun your kernel on a fresh *accelerator_view.* It is not possible to recover from a TDR error on the default *accelerator_view* without restarting the application. More details on this are provided in the "Time-Out Detection and Recovery" section of Chapter 12, "Tips, Tricks, and Best Practices."

When you create accelerator views of your own, to achieve isolation between threads on different accelerator views you should choose a queuing mode. This mode can be *immediate*, meaning that commands involving the accelerator view (a copy or a *parallel_for_each* involving an array on that view, for example) are sent to the device as they are processed on the CPU, or it can be *automatic*, meaning that such commands are accumulated in a queue and sent to the device when you flush the queue for that accelerator view. The default is automatic.

Sending a command to an accelerator involves building a DMA buffer, a set of commands, and references to GPU memory. It is quicker to build one DMA buffer used for several commands, as in the automatic queuing mode, than to build a DMA buffer for each command, as is done in immediate mode. (More information on queuing modes is in the "Queuing Modes" section of Chapter 7).

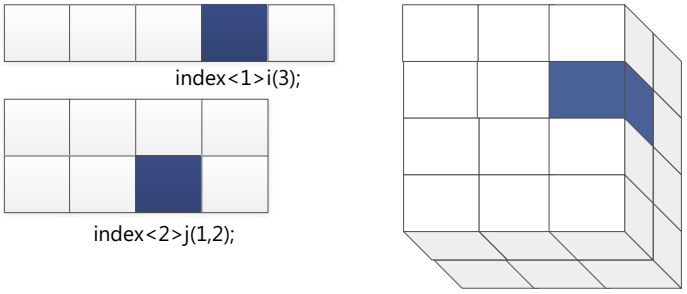
You must keep in mind that Windows will not allow a process to occupy the GPU for too long, so a batch that takes more than two seconds to complete will be cancelled (and the results will be lost). If your work has some long-running commands, immediate mode is safer than automatic mode. Chances are that under those circumstances, the overhead of building the DMA buffer will be far smaller than the work being done by the command anyway, so you are unlikely to be penalized for using immediate mode in those cases. More on the timeout for long calculations, and how to use an accelerator view to minimize errors due to long calculations, is in the "Time-Out Detection and Recovery" section of Chapter 12.

The accelerator object has a number of useful properties, including a description. Knowing that, you could now write your own version of the *ShowAmpDevices* utility introduced in Chapter 2, "NBody Case Study."

Note When you use an accelerator other than the default, you will have to modify your code to enable debugging on the REF accelerator unless you have a driver that enables hardware debugging on that accelerator. Chapter 6, "Debugging," shows how to use pre-processor directives that ensure debug builds always run on the default accelerator.

index<N>

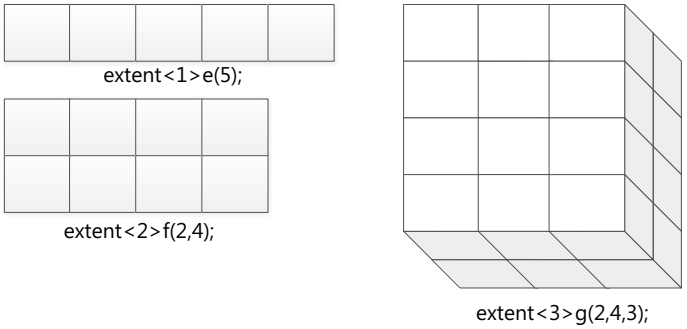
Each element in an *array* or an *array_view* resides at a position represented with an index. The number of integers in the index corresponds to the dimension of the array, and they are listed from most to least significant. The following diagram shows examples of one-dimensional, two-dimensional, and three-dimensional *index* declarations.



In a one-dimensional array, the elements indexed by (3) and (4) are next to each other in memory. In a two-dimensional array, the elements indexed by (1, 2) and (1, 3) are next to each other in memory. In a three-dimensional array, the elements indexed by (0, 1, 1) and (0, 1, 2) are next to each other in memory.

extent<N>

Just as a template class is used to identify the "address" of an element within an *array* or *array_view*, so a template class is used to described the size of an *array*, *array_view*, or section of an *array*. As always, the dimensions are most-to-least significant. Each dimension is the count of the number of elements, independent of the starting point or origin.



Extents are used to describe the size of *array* and *array_view* instances. The constructors shown for arrays that pass in one, two, or three integers for the size are convenience overloads, offered because one-dimensional, two-dimensional, and three-dimensional arrays are the most commonly used. You can pass in an extent explicitly if you prefer. The array has an *extent* property, much like the *size()* method of a standard collection class, that returns an extent object describing the array.

array_view<T, N>

An *array* represents data on an accelerator. You can construct it and fill it with data in a single step or construct it and fill it with data later. In either case, after some calculations have been performed on it, you will almost certainly copy the results from an array back to the CPU so that you can use them in some other part of your application.

Note Some applications, such as the NBody case study you saw in the previous chapter, don't copy the data back. They leave it on the GPU so that the representation of the moving particle clouds can be rendered. But it's more typical to return values that are then used in another part of the application or shown to the user.

You can certainly write useful applications using only arrays, but C++ AMP also offers the *array_view*, which supports features that often make it more convenient than working directly with arrays. An *array_view* looks like an *array* to the accelerator, but it saves you the trouble of arranging to copy the data to and from the accelerator.

The relationship between an *array_view* and an *array* is somewhat (but not precisely) like that between a reference and the object it refers to. Like a reference, array views must be initialized when they are created. Also as with a reference, changing the *array_view* changes (eventually) the data it was created from. However, the reverse is not true: changing the data from which the *array_view* was created might not automatically change the *array_view*, so you should approach such operations with care.

You can create an *array_view* both from an *array* on an accelerator and from some collection of data (such as a *std::vector*) on the CPU. After you have constructed an *array_view*, the data will be copied around as needed. For example, when a *parallel_for_each* on the accelerator starts to use the values in that *array_view*, those values will be copied to the accelerator. After parallel processing, when you are finished with that *array_view*, you can synchronize the new values in the *array_view* on the accelerator back to the vector.

The following code demonstrates this approach. The vector, *v*, is created in the CPU memory and initialized to contain the numbers *{0, 1, 2, 3, 4}*. Next, an *array_view* is initialized with the data stored in *v*. Not until the *parallel_for_each* causes code to execute on the GPU does the data get copied to the GPU's memory. The lambda executes on the GPU and doubles the value of each element in the array. The program must call *synchronize()* on the *array_view* before trying to access the changed values on the CPU.

```
std::vector<int> v(5);
std::iota(v.begin(), v.end(), 0);
array_view<int, 1> av(5,v);
parallel_for_each(av.grid, [=](index<1> idx) restrict(amp)
{
    av(idx) = av(idx) * 2;
});
av.synchronize();
```

It's possible that in the future or on certain accelerators there will be optimizations such as only copying back changed elements, though you can't count on that. If the run time knows that an *array_view* wasn't changed, the call to *synchronize()* will do nothing.

You can wrap an entire array in an *array_view* by calling the *view_as()* method of the *array*, which returns an *array_view*. Alternatively, you can use the *section()* method of the *array* class, passing in an origin and an extent, to wrap just part of an array in an *array_view*. There are also times when you want to look at the data differently; for example, to look at a three-dimensional array as though it were a one-dimensional array. The *reinterpret_as()* method of the array will give you such a view. Like a *reinterpret_cast* in standard C++, the name alerts anyone reading your code that you are making an important change in the way you access the data. The method is used only to reduce arrays to a rank of one—in other words, you can't use it to look at a three-dimensional array as a two-dimensional array. The layout of the elements is the same as it is in memory, with elements that differ by one in their least significant index next to each other.

Once you wrap CPU-bound data, such as a *std::vector*, in an *array_view*, it's best not to change the original data on the CPU. Because of the caching, your changes might not be reflected in the *array_view* that is used on the accelerator, and they might be overwritten if the *array_view* data is copied back into the source vector. If you *must* make changes to the source data directly, call *refresh()* on the *array_view* afterward to update the *array_view* with your changes. You will have to do some synchronization work to make sure the *array_view* doesn't change while you change the source vector.

Lambdas in C++11

The latest C++ standard, known as C++11, introduced several new language features, including lambda expressions, also known just as "lambdas." At first, lambdas appear to solve only a very minor annoyance: they free you from having to name small functions before passing them to standard algorithms, such as *std::sort()*, or from needing to build functors or function objects to pass around the idea of "doing something" from one place in code to another. Although it might seem that lambdas thus offer only "syntactic sugar," the convenience, readability, and simplicity of lambdas is remarkable and they make many idioms usable and comfortable. The "entry point" to C++ AMP, the *parallel_for_each* call, takes a lambda as a parameter. Here is a quick review of lambda syntax for those who might not have seen it before.

A lambda expression can go wherever any expression might go, most commonly on the right-hand side of an assignment or as a parameter to a function call. For example, consider this code to echo all the elements of some vector *v* onto standard out:

```
void print(int i)
{
    std::wcout << i << " ";
}

// . . .

std::vector<int> v(5, 0);
// . . .
std::for_each(v.begin(), v.end(), print);
```

Although they're adjacent here, in most cases the *print()* function is a very long way from the code that uses it, so readability suffers. In large code bases, coming up with unique names for these tiny helper functions is a challenge. Worse, the actual contents of the function can change over time without the function being renamed. Irritations like this lead many developers to just use a language *for* instead, which is a shame, because the *for_each* algorithm is actually more expressive than a language *for*, which could be doing something more complex.

Using a lambda expression as the last parameter to the *for_each* call lets you include the code right where you're using it so you know what it does. It also saves you the trouble of naming a one-line function and keeping that name and the function's contents accurate over time. Here's the lambda equivalent of the *for_each* call above:

```
std::for_each(v.begin(), v.end(), [](int i) { std::wcout << i << " ";});
```

All lambda expressions start with [], although the square brackets are not always empty, as they are in this example. This is called the "capture clause." Next, in parentheses, come the arguments to the lambda. The *for_each* algorithm will call the lambda once for each element in the vector, passing the element to the lambda. Finally, wrapped in braces comes the body of the lambda. It need not be a single line, and there are no restrictions on the kind of code you write in a lambda. It can even contain one or more *return* statements, which return from the lambda itself. When lambdas do not contain *return* statements, or when the entire lambda is a single *return* statement, the compiler can deduce the type automatically. In all other cases you must specify it, like this:

```
std::vector<int> v;
// . . .
std::vector<double> dv;
transform(v.begin(), v.end(), back_inserter(dv), [](int n) -> double
{
    if (n % 2 == 0) {
        return n * n * n;
    } else {
        return n / 2.0;
    }
});
```

The lambdas used in C++ AMP's *parallel_for_each* don't return values, so you won't be seeing that notation much, but it's included here for completeness.

Lambdas take arguments provided to them by the calling code, such as the *for_each()* algorithm. They can also have access to values that are in scope at the point where the lambda is created. When you create a lambda, the compiler actually generates an anonymous function object; values from the scope where the lambda is created can be kept in member variables of the function object at your direction. You may name specific values you wish to capture by value by including them in the capture clause:

```
int x, y;
// . . .
std::for_each(v.begin(), v.end(), [x, y](int n)
{
    if (n >= x && n <= y)
        std::wcout << n << " ";
});
```

You may also name specific values you wish to capture by reference:

```
int x, y;
// . . . std::for_each(v.begin(), v.end(), [&x, &y](int& r)
{
    const int old = r;
    r *= 2;
    x = y;
    y = old;
});
```

Or you can instruct the compiler to capture everything you use in the lambda body by value with a capture clause of [=] or capture everything you use by reference with a capture clause of [&]. You can even combine the two:

```
process(0, numItems, [=, &y](int i)
{
    //use various values from calling scope
    // any changes to y in the lambda will be reflected in calling scope
});
```

More Info For more on lambdas, this recorded session from the PDC is well worth watching: *http://channel9.msdn.com/events/PDC/PDC10/FT13*.

It is certainly not necessary to know all about how lambdas work to use them in your C++ AMP code. To get started, it will be enough to recognize the pattern [] () {}. That is the capture clause, lambda argument, and lambda body.

parallel_for_each

At the heart of C++ AMP is the *parallel_for_each* function. This is what parallelizes the work. You set up an array and put values in it, or you set up an *array_view* around some values you have in a CPU-bound data structure such as a *std::vector*. Then you use a *parallel_for_each* to do something to each element in that array or *array_view*, or a piece of it. A *parallel_for_each* operates over an extent—the shape of the extent is what controls the number of threads that do the work.

Note A variant of *parallel_for_each* exists that operates over a tiled extent and behaves slightly differently. That variant is discussed in Chapter 4, "Tiling," and in Chapter 5, "Tiled NBody Case Study."

Here is the sample from Chapter 1 again:

```
#include <amp.h>
using namespace concurrency;

void AddArrays(int n, const int* const pA, const int* const pB, int* const pC)
{
    array_view<int,1> a(n, pA);
    array_view<int,1> b(n, pB);
    array_view<int,1> c(n, pC);
```

```
    parallel_for_each(c.extent, [=](index<1> idx) restrict(amp)
    {
        c[idx] = a[idx] + b[idx];
    });
}
```

The preceding code wraps an *array_view* around each of the raw pointers (presumably to a C-style array) representing the one-dimensional matrices to be added. Each *array_view* is one-dimensional, of extent (*n*), and holds integers. The first parameter to the *parallel_for_each* is the extent of *c*, the *array_view* that will hold the result. In the particular case of adding one-dimensional matrices, the size (product of each dimension of the extent) is the same for all three array views, but in many other algorithms, you will need to put some thought into choosing the extent to shape the threads. The second parameter is the lambda that will actually do the calculation. In this example it captures everything it uses by value. Only arrays can be captured by reference (and they must always be captured by reference)—all other types, including *array_view* instances, are captured by value.

When an array is passed to a *parallel_for_each*, it runs on the *accelerator_view* associated with the array (which might be the default view on the default accelerator.) If none can be determined (for example, in this case where only *array_view* instances are captured by the lambda), the *parallel_for_each* will run on the default accelerator, whatever that might be on the system where this runs. (There are overloads of *parallel_for_each* that take an *accelerator_view* if you want to specify where it runs.) The lambda is decorated with the new *restrict* keyword, which directs the compiler to ensure that nothing in the lambda is inappropriate for running on the accelerator. The actual restrictions this imposes will be discussed shortly. You could use a functor instead of a lambda if you happened to have written one or if the calculation needs to be used in multiple places in your code. In these samples, however, lambdas are a natural choice and make the code more readable. An example of using a functor with a *parallel_for_each* is in Chapter 12.

The accelerator launches one thread for each element in the extent. In the code example above, *c* is a one-dimensional *array_view* of extent (*n*), so *n* threads will run on the accelerator, each calculating one element of *c*. If *c* had the extent *(2, 3, 4)*, then 24 ($2 \times 3 \times 4$) threads would run on the accelerator. Of course, you're unlikely to realize a performance benefit from moving such a small grid to the accelerator because the cost of copying the data back and forth outweighs the savings. Problem spaces that involve millions of calculations, such as a three-dimensional grid of extent (200, 200, 200), that do more with the data than simply add it are more likely to see a performance benefit. For the *parallel_for_each* to calculate the correct answer, the threads must all be independent. There are no guarantees about the order of execution, for example, and it's not possible to have a thread wait until some other thread has calculated a number or to communicate between the threads.

The "kernel function"—the lambda or functor to be executed by each thread—has these characteristics:

- It returns *void*. Whatever calculations it does should be stored in an element of one or more of the arrays or array views being worked on.

- It takes only an index (of the dimensionality of the extent) in the nontiled (default) case. This index is used to determine inputs to the calculation and where to store the result of the calculation.
- It is marked with *restrict(amp)* or *restrict(amp, cpu)*.
- It may call only functions marked with *restrict(amp)* or *restrict(amp, cpu)* that are visible at code generation time. This might mean a function that is implemented in the same .cpp file or is inline in a header file included by this .cpp file, or if link time code generation (*/ltcg*) is used, it could be in a different .cpp file that is linked in. A function imported from a separate executable, or from a .lib or .obj that didn't use */ltcg*, is not visible at code generation time. Of course, the same restriction applies to functions called from a function that is called from the kernel function. The compiler must be able to inline the entire call graph.
- It must capture nothing (except array instances) by reference. Capturing by value is allowed.
- It must capture only types that are compatible with the restrictions for the accelerator, which are discussed in the next section.

Note In some applications, you might have some code that you wish to use both as the kernel of a *parallel_for_each* and for some other purpose. If you find yourself in that situation, you might be interested in some slight relaxations of these rules as outlined in the official specification for C++ AMP. For example, the function is allowed to return a value, but the *parallel_for_each* will ignore it. It's allowed to take parameters in addition to the index as long as they are optional parameters with default values. These and other technicalities are safe to ignore when you are writing the kernel function only for use with a *parallel_for_each*, as you typically are when you get started with C++ AMP.

As discussed in the NBody case study, the "heavy lifting" of writing these kernel functions lies in complying with the rule that the lambda call only functions with the same restriction that are visible at code generation time. Relatively simple mathematical calculations might not be implemented in functions marked *restrict(amp)* or might be in separate .lib or .obj files. For this reason, a variety of functions are available in the *concurrency::fast_math* and *concurrency::precise_math* namespaces, all marked *restrict(amp)* and implemented appropriately. Those are discussed later in this chapter.

Functions Marked with *restrict(amp)*

A function must follow a number of rules to successfully compile with *restrict(amp)*. The first, as mentioned in the *parallel_for_each()* section, involves functions that it calls. Those must be visible at code generation time and must also be marked with *restrict(amp)*. If you are not using link time code generation, this essentially means they must be in the same .cpp file by compile time, possibly from a header file included in that .cpp file. If you are using */ltcg* when compiling both .cpp files (the one

that calls the function and the one that implements it) as well as when linking, then you can keep the calling and called functions in separate files.

A C++ AMP-compatible function or lambda can only use C++ AMP-compatible types, which include the following:

- *int*
- *unsigned int*
- *float*
- *double*
- C-style arrays of *int*, *unsigned int*, *float*, or *double*
- *concurrency::array_view* or references to concurrency::array
- structs containing only C++ AMP-compatible types

This means that some data types are forbidden:

- *bool* (can be used for local variables in the lambda)
- *char*
- *short*
- *long long*
- unsigned versions of the above

References and pointers (to a compatible type) may be used locally but cannot be captured by a lambda. Function pointers, pointer-to-pointer, and the like are not allowed; neither are static or global variables.

Classes must meet more rules if you wish to use instances of them. They must have no virtual functions or virtual inheritance. Constructors, destructors, and other nonvirtual functions are allowed. The member variables must all be of compatible types, which could of course include instances of other classes as long as those classes meet the same rules.

The actual code in your amp-compatible function is not running on a CPU and therefore can't do certain things that you might be used to doing:

- recursion
- pointer casting
- use of virtual functions
- *new* or *delete*
- RTTI or dynamic casting

- *goto*
- *throw*, *try*, or *catch*
- access to globals or statics
- inline assembler

It might be helpful to see the error messages the compiler produces if you break these rules. The following code meets all the rules and compiles without error:

```
std::vector<int> v(5);
std::iota(v.begin(), v.end(), 0);
array<int, 1> a(5, v.begin(), v.end());
parallel_for_each(a.extent, [&](index<1> idx) restrict(amp)
{
    a[idx] = a[idx] * 2;
});
```

Now replace the line in the lambda with a call to a function that is not marked *restrict(amp)* but is in the same .cpp file, and you will see this (the name DoubleIt is the function that was called):

```
error C3930: 'DoubleIt' : no overloaded function has restriction specifiers that are compatible
with the ambient context
```

You'll also see that message if the function has the appropriate restriction but is not visible at code generation time.

If you use a type, such as *short*, that is not allowed in *restrict(amp)* code, you see an error message like this:

```
error C3581: 'short': unsupported type in amp restricted code
```

That's the same message you see if you try to declare a pointer-to-pointer or other unsupported type.

Copying between CPU and GPU

Data is copied between the CPU and the accelerator (usually a GPU) either automatically or with one of the many overloads of the *copy()* method provided in amp.h. For example, you can construct an *array* on the default accelerator and copy data to it in a single call:

```
array<int, 1> a(5, v.begin(), v.end());
```

Alternatively, you can construct an empty array and later use the *copy()* function to fill it with data. An *array_view* associated with a container on the CPU will automatically copy to the accelerator when the processing of that *array_view* on the accelerator begins, and it can synchronize changed data back for use on the CPU, as shown in the *array_view* section earlier in this chapter.

These blocks of code are equivalent:

```
std::vector<int> v(5);
std::iota(v.begin(), v.end(), 0);
array<int,1> a(5,v.begin(),v.end());
parallel_for_each(a.extent, [&](index<1> idx)
    restrict(amp)
{
    a[idx] = a[idx] * 2;
});
copy(a,v);
```

```
std::vector<int> v(5);
std::iota(v.begin(), v.end(), 0);
array_view<int,1> av(5,v);
parallel_for_each(av.extent, [=](index<1> idx)
    restrict(amp)
{
    av[idx] = av[idx] * 2;
});
av.synchronize();
```

When you access the *array_view* on the CPU after the *parallel_for_each*, it will synchronize automatically, so you can omit the call. This is a major advantage of using an *array_view*.

Of course, automatic copies that are not really needed are bad for performance. C++ AMP gives you control over this automatic copying. When declaring an *array_view*, you can give the compiler a hint that the data will be sent to the accelerator but not changed there:

```
array_view<const int, 1> a(5, v);
```

Doing this prevents any automatic copying back from the accelerator. It's a clever reuse of a keyword and concept already well known to C++ developers. You can also provide a hint that the starting values don't need to be copied to the accelerator because the kernel function will overwrite them, for example:

```
array_view<int, 1> out(5, v2);
out.discard_data();
```

There is no *writeonly* keyword (or *anti-const*) in C++, so you use this function call instead.

There is another way to get data from an *array_view* back to the CPU-located collection it wraps, and that is for the *array_view* to be destructed. For example, this version of the code causes the results of the calculation to appear in the *std::vector v2* even though it never calls *synchronize*:

```
std::vector<int> v(5), v2(5, 0);
std::iota(v.begin(), v.end(), 0);
// braces for scope only
{
    array_view<const int, 1> a(5, v);
    array_view<int, 1> out(5, v2);
    out.discard_data();
    parallel_for_each(a.extent, [=](index<1> idx) restrict(amp)
    {
        out[idx] = a[idx] * 2;
    });
}
```

This automatic copying at end-of-scope for the array views is why the *const* hint is important—there's no need to copy unchanged values from *a* back to *v*, and the hint prevents it.

At any time, you can explicitly copy from one array to another, from one *array_view* to another, from an array to a CPU-located collection such as a *std::vector*, and so on. The two parameters to copy are the source and the destination.

Note Depending on how much exposure you have to Standard Library methods, which typically take source as the first parameter, and C-style functions, which typically take destination as the first parameter, you might find it a challenge to remember the parameter order. However, an order had to be picked, and the one chosen is more like Standard Library than like C. The C++ AMP designers follow C++ conventions more often than C conventions. IntelliSense will remind you if you need a hint.

Any two containers between which you copy data must match on both element type and number of elements. If both containers have rank (*array* to *array*, *array* to *array_view*, *array_view* to *array_view* or *array_view* to *array*), then the rank must also match. You can use all the standard containers that support iterators.

Math Library Functions

As mentioned earlier, you can't call just any function from the "kernel function" in your *parallel_for_each*. The functions you call should be visible at code generation time and marked with *restrict(amp)*. If you are converting some existing code and you wrote it all, you can probably make the necessary changes to your code. But chances are that some of your code calls library functions such as *sqrt()* or *sin()*. Therefore, you'll need to replace those calls with accelerator-compatible versions. The good news is that hundreds of such functions are defined for you in amp_math.h in the namespace *concurrency::fast_math*. They span over 4,000 lines in that file, so listing all of them here is impractical. A quick summary of the categories might, however, be useful.

- Trigonometry: *cos*, *sin*, *tan*, and *arccos*, *arcsin*, and *arctan*, plus the six hyperbolic trig functions
- Roots and powers: *sqrt*, *cbrt*, *pow*
- Simple manipulations: *ceil*, floor, *round*, *trunc*, *copysign*, *abs*, *mod*, *max*, *min*, and so on
- Exponents: *exp* (e to the x), *expm1* (e to the x, minus 1), *exp2* (2 to the x), *exp10* (10 to the x), and so on
- Logs: *log* (base e), *log10*, *log2*, *log1p* (log base e of (x+1)), and so on
- Compound operations such as *fdim* (x-y if it's positive, otherwise 0) *fma* (x*y+z), or *hypot* (square root of the sum of the squares)

The upshot is that if you've been using a mathematical function from the Standard Library, there's a good chance a *restrict(amp)* version has been defined for you in amp_math.h. There is also a *concurrency::precise_math* namespace with double-precision versions of the functions, but you can use those only on an accelerator that supports double precision. The section "Double-Precision Support" in Chapter 12 has more details on double-precision support.

Summary

This chapter covered the building blocks from which C++ AMP applications are constructed. Writing a C++ AMP application means, first and foremost, writing a C++ application. You use familiar-looking constructs such as templates to represent data on the accelerator, and you can take advantage of the many overloads that have been written to make copying data to and from an accelerator simple.

The *array* and *array_view* templates introduced in this chapter serve a similar purpose—they represent data on an accelerator. Both have an extent that can be used to shape the threads of a *parallel_for_each*. The difference is that an *array_view* is a wrapper around data that can copy (or elide copying) data back and forth between the CPU and an accelerator, while an *array* is on an accelerator, requiring developers to write code to copy the data back and forth as needed.

CHAPTER 4

Tiling

In this chapter:

Purpose and Benefit of Tiling 64
***tile_static* memory** 65
tiled_extent 66
tiled_index 67
Modifying a Simple Algorithm into a Tiled One 68
Effects of Tile Size 77
Choosing Tile Size 79
Summary 81

The time comparisons of the case study in Chapter 2, "NBody Case Study," showed that C++ AMP can dramatically outperform single-threaded simple calculations on the CPU. If your only two choices when designing your application were simple C++ AMP and single-CPU, then the choice would be simple: you'd use C++ AMP. However, you have a number of other choices that can make your CPU solutions faster. One of these is clever caching of intermediate results to reduce the number of times each calculation must be performed. Another, done automatically for you on the CPU, is caching variables you're working with in memory locations that are dramatically faster to access. You can modify your algorithm to allow the CPU to do this more efficiently.

Note The NBody sample includes a number of CPU optimizations not discussed in this book that you can use to speed up CPU calculations by helping the caching process.

The GPU (and other accelerators) can offer similar enhancements that will make your C++ AMP code even faster. In this chapter you will see how tiling—grouping threads into tiles that can share access to programmable caches—can produce the most dramatic speedups, especially for algorithms that use each piece of data more than once.

Purpose and Benefit of Tiling

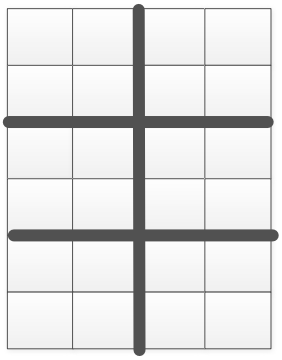
Accelerators such as a typical GPU have a relatively small programmable cache that can be accessed far more quickly than the global accelerator memory used for an *array* or *array_view*. How much more quickly? On the order of a hundred times faster! Of course, there's more to your algorithm than accessing memory, but taking advantage of that fast memory can produce a significant gain that can have a big impact on your application's overall performance. It's a bit complicated to implement tiling in your application, but keep this benefit in mind and you might find the additional effort worth your while.

The GPU has a programmable cache, unlike the automatic caches common on CPUs, because of the hardware differences between the GPU and the CPU. You have to choose to use the GPU's programmable cache and you have to manage it yourself. There are two key reasons for this. The first is the drive to keep GPUs simple. The hardware caches on a CPU consume overhead, transistors, and die space. On a GPU, the space that could be used for managing hardware caches is instead used for more GPU cores. The second reason is to keep the developer in control. On a CPU, several levels of cache—each larger and slower than the one before—combine with some heuristics about what to keep in the cache and what to drop to add something new, that have been proven to improve performance for most applications. Generally a cache miss is satisfied by the next level of cache, minimizing the cost of a miss. On the GPU there is just one cache (so misses are expensive), the cache is quite small (so contention will happen), and the best cache policy depends strongly on the algorithm you are executing. Having the developer manage the cache means more work but also better results. As is so often the case for C++ developers, we give up the simplicity of having something managed for us in exchange for maximum flexibility while working with limited resources.

To perform a tiled calculation with C++ AMP, you will define a *tile*, a group of threads within the overall thread group that is executing your *parallel_for_each*, the compute domain. The tile has the same rank as the extent you are using to shape the *parallel_for_each*. (Tiling is available only for extents of one, two, or three dimensions.)

To clarify, some diagrams of very small tiles (far smaller than are realistic) may help. Here, a one-dimensional array of extent 4 is divided into two tiles, each of extent 2:

This is a two-dimensional array of extent 6×4 divided into 2×2 tiles:



Tiling your algorithm means making two adjustments. First, you would use a tiled index instead of the simple index used in the nontiled case (this is fairly mechanical). Second, you would use the programmable GPU cache for the quickest possible calculations. Examples of each of these changes to the kernel follow later in this chapter. Reworking your kernel to use tiling can make your algorithm harder to read, but for many algorithms the results are worth it.

tile_static Memory

Each thread in a given tile has access to a tile-specific area of memory, the programmable cache, sometimes called a scratchpad or local memory. When you declare *tile_static* variables, they are in this cache. For example, a C++ array to hold some intermediate data could be declared like this:

```
tile_static float sA[16][16];
```

If two threads are in the same tile, they access the same memory when they refer to a *tile_static* variable like sA. If two threads are in different tiles, they appear to access different memory—it's as though their instances are in different caches.

The restrictions on declaring *tile_static* are the same as any local variable declared in an amp-restricted function. If you use an object here rather than a fundamental type, array of fundamental type, and so on, the class should be bitwise-copyable. It can have a constructor or destructor (if you wrote them for other reasons), but they won't be called on the accelerator. The lifetime of *tile_static* data begins when the first thread in the tile reaches the line that declares the *tile_static* variable and ends when the last thread in the tile returns from the kernel function.

The *tile_static* storage class is one of the two language changes required for C++ AMP (the other being the use of the *restrict* keyword), and you can use it only within a kernel function for a tiled *parallel_for_each* algorithm. It may not appear in a function marked *restrict(cpu)* or *restrict(cpu, amp)*.

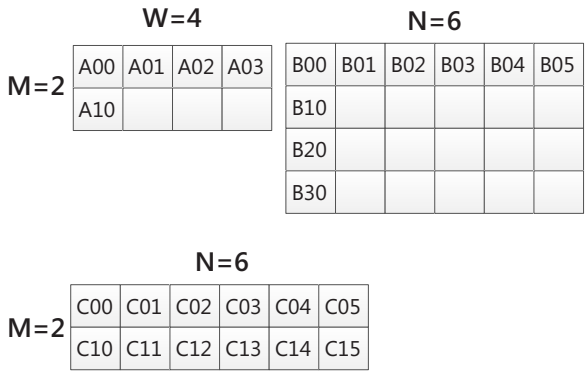
tiled_extent

The *parallel_for_each* introduced in Chapter 3, "C++ Fundamentals," works with an extent, often using a property of an *array* or *array_view*. For example, you can implement a simple matrix multiplication of two-dimensional arrays like this:

```
void MatrixMultiply(std::vector<float>& vC,
    const std::vector<float>& vA,
    const std::vector<float>& vB, int M, int N, int W)
{
    array_view<const float,2> a(M, W, vA);
    array_view<const float,2> b(W, N, vB);
    array_view<float,2> c(M, N, vC);
    c.discard_data();
    parallel_for_each(c.extent, [=](index<2> idx) restrict(amp)
    {
        int row = idx[0];
        int col = idx[1];
        float sum = 0.0f;
        for(int i = 0; i < W; i++)
            sum += a(row, i) * b(i, col);
        c[idx] = sum;
    });
    c.synchronize();
}
```

It's worth noting that this algorithm uses the *const* hint for the input array_views, *a* and *b*, and calls *discard_data()* for *c*, the *array_view* that wraps the result data. This further improves performance by eliminating copies of *a* and *b* from the accelerator (because the kernel leaves them unchanged) and the initial copy of whatever data is in *c* to the accelerator (because it is not used by the kernel, only written to).

These are two-dimensional array views, and *c* is an $M \times N$ array view. Because dimensions in C++ AMP arrays and array views are listed from most-to-least significant, here is a set of small arrays labeled with their coordinates as a reminder of the notation. To fit on the diagram, *A[0, 0]* is labeled *A00*, and so on.



Using this notation, the value of *C00* (to take one example) is

```
A00 * B00 + A01 * B10 + A02 * B20 + A03 * B30
```

In the code presented earlier, the shape of the *parallel_for_each* was determined by the *extent* associated with *c*, the result of the matrix multiplication. Because *c* is an $M \times N$ array, there will be $M \times N$ threads—12 in the example used for the diagram, although of course a real example would have thousands or millions of threads on the accelerator to manipulate much larger arrays. This version of *parallel_for_each* does not tile the threads.

There is another overload of *parallel_for_each* that does a tiled computation. This overload uses a *tiled_extent* instead of an *extent*. Getting a *tiled_extent* is easy—just choose a tile size and call the *tile()* method of an *extent* you already have. It's a templated function and expects you to specify the tile size. Tile size is expressed as a rank that matches the *extent* you're working with and must be a compile-time constant. So, for example, to change the matrix multiplication example to use 16×16 tiles (256 threads per tile), the *parallel_for_each* call would start like this:

```
parallel_for_each(c.extent.tile<16, 16>(), // ...
```

In practice, it's common to put the tile size into a *const* variable or template argument so it can be used for loop bounds and other calculations. For example:

```
static const int TileSize = 16;
```

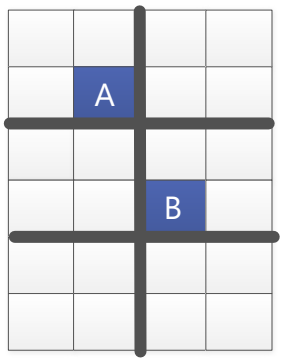
Choosing the tile size is not arbitrary. Each dimension of the extent must be a multiple of the tile size in that dimension; in other words, the extent must be divided into an integer number of tiles. In addition, the product of the dimensions of the tile size can't exceed 1,024. In other words, a one-dimensional tile can have up to 1,024 for tile size, a two-dimensional tile can have up to 32×32 (or other combinations that multiply to 1,024 or less) and a three-dimensional tile can have $8 \times 8 \times 16$ (or again, other combinations that multiply to 1,024 or less.) Optimal tile size is discussed later in this chapter in the sections "Effects of Tile Size" and "Choosing Tile Size."

tiled_index<N1, N2, N3>

The *parallel_for_each* overload that doesn't use tiling takes a kernel—a function object or lambda—that in turn takes an *index* whose rank matches the *extent*. Similarly, the tiled *parallel_for_each* also takes a kernel that takes a *tiled_index* whose rank matches the *tiled_extent*. But whereas an *index* just represents a point and carries one integer for each dimension, a *tiled_index* carries more indices. A tiled index represents a point within a tile and has four properties representing position:

- **global** the overall position within the extent
- **local** the position within the tile
- **tile_origin** the origin of the tile within the extent
- **tile** the overall "tile index"

These are best illustrated with a diagram.



In this 6×4 extent, the index of the point labelled *A* is *(1, 1)*. The index of the point labelled *B* is *(3, 2)*. If the extent is divided into 2×2 tiles, as indicated by the heavier lines, then when each of these points is presented as a *tiled_index*, it will have the following properties:

```
A.global: (1,1)          B.global: (3,2)
A.local: (1,1)           B.local: (1,0)
A.tile_origin: (0,0)     B.tile_origin: (2,2)
A.tile: (0,0)            B.tile(1,1)
```

It's important to be able to translate back and forth between these in order to read and write tiled algorithms correctly. The relationships are generally easy to spot when using these small numbers; for example, the tile origin plus the local index will always be the global index. The *tile* property times the tile size will always be the tile origin.

The kernel executed by the *parallel_for_each* receives a tiled index, and then your calculation may use some combination of these four properties in the calculations to be performed by each thread. This obviously makes your kernel more complicated, so let's start with a simple kernel and transform it into a tiled version.

Modifying a Simple Algorithm into a Tiled One

The first step in writing a tiled algorithm is to write a simple (nontiled) algorithm that gives correct answers. This will ensure that you have declared any needed *array* or *array_view* instances properly and that your calculations are correct. With that complete, you can apply a methodical transformation on your code to make it into a tiled algorithm. The samples in this book often present both simple and tiled versions of the same application. This is because the tiled code is harder to read. In your own applications, you would probably not keep the simple code once the tiled code was written and tested. Here are the steps to follow in order to change a simple algorithm into a tiled one:

1. Choose a tile size and declare it as a compile-time constant.
2. Change the *parallel_for_each* to use a *tiled_extent* instead of an extent.

3. Change the kernel to take a *tiled_index* instead of an index.

4. Change the kernel body to use the global property of the tiled index where it once just used the index.

You might not have chosen an optimal tile size, and you won't be taking advantage of any *tile_static* memory, but this will get the process underway. Transforming the matrix multiplication code like this produces the following result:

```
static const int TileSize = 16;

void MatrixMultiply(std::vector<float>& vC,
    const std::vector<float>& vA,
    const std::vector<float>& vB,
    int M, int N, int W)
{
    array_view<const float,2> a(M, W, vA);
    array_view<const float,2> b(W, N, vB);
    array_view<float,2> c(M, N, vC);
    c.discard_data();
    parallel_for_each(c.extent.tile<TileSize, TileSize>(),
        [=](tiled_index<TileSize, TileSize> tidx) restrict(amp)
    {
        int row = tidx.global[0];
        int col = tidx.global[1];
        float sum = 0.0f;
        for(int i = 0; i < W; i++)
            sum += a(row, i) * b(i, col);
        c[tidx] = sum;
    });
    c.synchronize();
}
```

The new code just calculates the row and column of each element in *c* by using the global property of the *tiled_index*, where the simple algorithm used the untiled index itself. For this particular algorithm, this is the only change required and the lines that use *row* and *col* don't need to change at all.

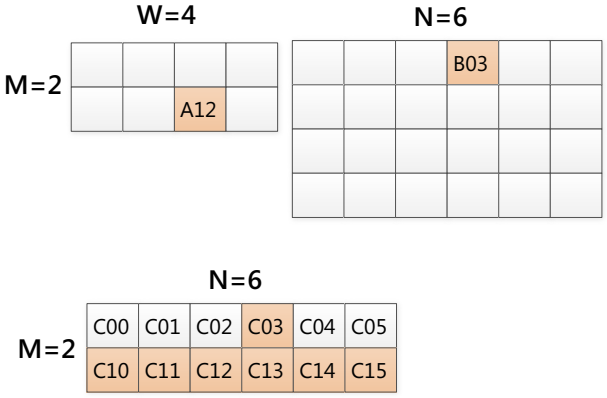
You can build and run this code to confirm that the matrix multiplication still produces the same result. (Make sure you set up matrices of the right extents—the extent of the result matrix, *C*, must be a multiple of the tile size.) You won't see any noticeable performance benefit, even though technically this is now a tiled algorithm. In fact, when you write simple code like the first matrix multiplication code shown in this chapter, it's actually tiled for you under the covers simply because of the way a GPU accelerator works. The implicit tiling, just as in this example, doesn't use any *tile_static* memory, which is where the benefit of tiling arises. Directing a *parallel_for_each* to use a *tiled_index* and a *tiled_extent* is not what produces performance improvements; it's merely a requirement to enable you to work with *tile_static* memory. You might see a slight performance benefit because the implicitly tiled code has some *if* statements to make sure your tile size and result extents are compatible. When you tile the algorithm yourself, these tests are removed, which might improve performance slightly.

Using *tile_static* memory

When you adapt a CPU algorithm to work with C++ AMP, your solution will involve copying data to the GPU global memory, working with it there, and copying it back to host memory. In the same way, tiling and using *tile_static* memory typically involves copying data from an *array* or *array_view* in global memory to the *tile_static* memory, or vice versa. If you simply copy the data once and then use it once, you gain nothing over using it directly from the array. To get performance improvements, you need an algorithm that would use a particular location in the GPU global memory (whether that is input, an intermediate result, or output) multiple times, since it can save time whenever it uses the *tile_static* value you have cached.

Array multiplication is just such an algorithm. When multiplying, for example, a 2×2 matrix and a 2×2 matrix, the point at *(0, 0)* in the first matrix will be used to calculate all of the points in row 0 of the result. The point at *(0, 0)* in the second matrix will be used to calculate all of the points in column 0 of the result. In general, the point at *(i, j)* in the first matrix will be used to calculate all of the points in row i of the result matrix and the point at *(i, j)* in the second matrix will be used to calculate all the points in column j of the result matrix.

Revisiting the matrices used in the diagrams earlier, look at which elements in the result matrix are affected by two particular cells in the input matrices. The value in *A[1, 2]* is involved in the calculation of all of row 1 of the result, *C*. The value in *B[0, 3]* is used to calculate all of column 3 of *C*.



This property of matrix multiplication—that a given point in the source matrix is used multiple times—is what makes it appealing to use *tile_static* memory and makes the tiling effort worthwhile. If the problem were matrix addition, in which each point is used only once, there would be no gain. If elements of *A* were cached in *tile_static* memory, the access to that value would be much quicker than retrieving it from global memory, providing a significant performance gain.

For example, consider the matrix multiplication example again. When running the thread that calculates the value at *(0, 0)* in the result matrix, you will need all of row 0 from matrix *A* and all of column 0 from matrix *B*. If using a 2×2 tile, one other thread in your tile (the one calculating *C(1, 0)*) will also need all of row 0 in matrix *A*, and one other thread (the one calculating *C(1, 0)*) will need all of column 0 in matrix *B*. In a larger tile, more threads will need each element; for matrix multiplication, a given column or row will be used as many times as one side of the tile. You could set up *tile_static* arrays to hold this information and gain some benefit.

A natural first thought when trying to write code that uses *tile_static* memory is to design it like this:

- First, copy the data to be used in calculations for all threads in this tile to *tile_static* memory.
- Next, use that data.

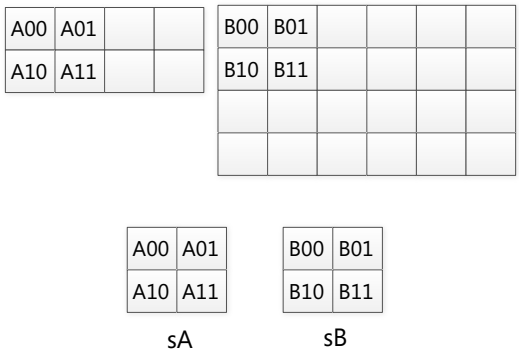
Pseudocode like this carries an implied "just once" or "just on one thread" for the "copy to *tile_static* memory" steps and an implied "for every thread in the tile" for the "use static memory" steps. But that is not how a *parallel_for_each* works. There is one thread for each element in the tiled extent, and you have no idea of in which order the threads will run. You can't spawn an extra set of threads, one for each tile, to get things set up for the "real" threads in each tile. Instead, those threads have to cooperate to get their calculations set up, just as they cooperate to get the overall calculation done. In the same way that each thread calculates one value of the result, for example, each thread can copy one element of the required information into *tile_static* memory before the calculation can begin.

So far, in the discussion of the matrix multiplication, the size of the matrices has not been specified. Whatever *M*, *N*, and *W* are, they are probably not the same as the tile size—after all, tile size is limited to a total product of 1,024, and C++ AMP provides speedups when dealing with arrays of thousands or millions of elements. This means that copying "one row" or "one column" to *tile_static* memory would involve copying a lot more elements than there are threads in the tile. It might mean even copying more elements than can fit into the small programmable cache. In most cases you gain the most benefit by adapting your algorithm to work with pieces of the problem so that the number of elements in each *tile_static* variable depends on the number of threads in each tile. The precise approach will be different for every algorithm.

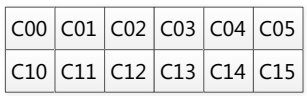
Here's how it works for matrix multiplication with 2×2 tiles. If you imagine your threads run sequentially (even though they don't) and that you can somehow do certain work "per tile" instead of per thread, you could write some pseudocode to define your algorithm. (After the algorithm is explained, you'll see an explanation of the cooperation to get the values copied from global memory to *tile_static* memory.) For the tile whose four threads calculate the values in the top left tile of the result matrix, the process would look like this:

- Once per tile, copy the 2×2 square at the top left of matrix *A* into a *tile_static* container, *sA*, for quick access.

- Once per tile, copy the 2×2 square at the top left of matrix *B* into a different *tile_static* container, *sB*, for quick access.

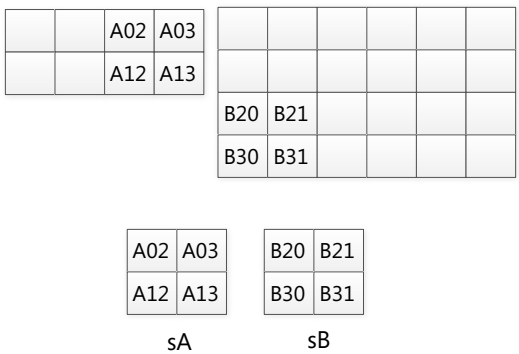


- For each thread in the tile, use two of the four values in *sA* and two of the four values in *sB* to calculate four partial values toward the final values in the top left of the result matrix, then add these to a running total for each element in the result tile.




```
C00_partial = A00 * B00 + A01 * B10
C10_partial = A10 * B00 + A11 * B10
C01_partial = A00 * B01 + A01 * B11
C11_partial = A10 * B01 + A11 * B11
```

- Move to the "next" 2×2 square—moving across matrix *A* and down matrix *B*—and repeat, accumulating more values toward the results in the top left 2×2 tile of the result matrix.



The running values for the result elements are now:

```
C00_partial = C00_partial + A02 * B20 + A03 * B30
C10_partial = C10_partial + A12 * B20 + A13 * B30
C01_partial = C01_partial + A02 * B21 + A03 * B31
C11_partial = C11_partial + A12 * B21 + A13 * B21
```

At this point the first tile in the result has been completely calculated: *C00_partial* is in fact the final result for *C00*, and so on. If the matrices were larger, it would take more "move to the next" tile steps to complete the calculation, but the algorithm would continue to work. This works for any tile with suitable offsets in it to ensure that the correct rows and columns of *A* and *B* are used in the calculation of the temporary "contributions," tile by tile, to the final accumulated result.

Here's how that might look in (incomplete) code:

```
static const int TileSize = 2;

void MatrixMultiplyTiled(std::vector<float>& vC,
    const std::vector<float>& vA,
    const std::vector<float>& vB,
    int M, int N, int W)
{
    array_view<const float,2> a(M, W, vA);
    array_view<const float,2> b(W, N, vB);
    array_view<float,2> c(M, N, vC);
    c.discard_data();

    parallel_for_each(c.extent.tile<TileSize, TileSize>(),
        [=](tiled_index<TileSize, TileSize> tidx) restrict(amp)
    {
        int row = tidx.local[0];
        int col = tidx.local[1];
        float sum = 0.0f;
        for (int i = 0; i < W; i += TileSize)
        {
            tile_static float sA[TileSize][TileSize];
            tile_static float sB[TileSize][TileSize];

            // TODO for each tile, copy elements into sA and sB
            // before other threads in the tile work with them

            for (int k = 0; k < TileSize; k++)
                sum += sA[row][k] * sB[k][col];
        }
        c[tidx.global] = sum;
    });
    c.synchronize();
}
```

With this tile size of 4 (2×2), each value is copied once to a *tile_static* area and then used four times. For matrix multiplication, the number of times you will use each element of the *tile_static* memory is the number of threads in the tile. (For another problem it might always be 1, or the number of threads squared, or something completely different—the answer depends entirely on your algorithm.) Because accessing *tile_static* memory takes only about one percent of the time it takes to access the source *array* or *array_view* in global memory, the more times each piece is used, the more time you are saving by using *tile_static*. You incur the cost of accessing global memory once and then incur only four small costs to access *tile_static*, instead of four accesses to global memory.

Tile Barriers and Synchronization

Designing a tiled algorithm that copies as efficiently as possible is simpler when the size of the *tile_static* arrays coincides with the number of threads in the tile. In the 2×2 tile matrix multiplication example above, there are four threads in the tile and four values to be copied into each of *sA* and *sB*. The pseudocode is not realizable in code as written because of the phrases like "once per tile" that simply can't be coded. But because there is one element to be copied for each thread in the tile, you can rewrite it like this:

- For each thread in the tile, copy one element of the appropriate 2×2 square in the first matrix to *sA*.
- For each thread in the tile, copy one element of the appropriate 2×2 square in the second matrix to *sB*.
- For each thread in the tile, use an entire row (two elements) of *sA* and an entire column (two elements) of *sB* to calculate a partial sum toward the result in this thread's element of the result array.
- Move to the "next" tile-sized square and repeat.

There are four threads per tile and four elements to be copied into *sA* and *sB*. This is how you can achieve the copying just once—by spreading the work of the copy among the threads within the tile. The code (still incomplete) might look like this:

```
static const int TileSize = 2;

void MatrixMultiplyTiled(std::vector<float>& vC,
    const std::vector<float>& vA,
    const std::vector<float>& vB,
    int M, int N, int W)
{
    array_view<const float,2> a(M, W, vA);
    array_view<const float,2> b(W, N, vB);
    array_view<float,2> c(M, N, vC);
    c.discard_data();

    parallel_for_each(c.extent.tile<TileSize, TileSize>(),
        [=] (tiled_index<TileSize, TileSize> tidx) restrict(amp)
    {
        int row = tidx.local[0];
        int col = tidx.local[1];
        float sum = 0.0f;
        for (int i = 0; i < W; i += TS)
        {
            tile_static float sA[TileSize][TileSize];
            tile_static float sB[TileSize][TileSize];
            sA[row][col] = a(tidx.global[0], col + i);
            sB[row][col] = b(row + i, tidx.global[1]);

            for (int k = 0; k < TS; k++)
                sum += sA[row][k] * sB[k][col];
        }
```

```
        c[tidx.global] = sum;
    });
    c.synchronize();
}
```

There is a big problem in this code—a race condition. What if thread 3 finishes copying and is ready to start calculating while thread 1 has not yet finished copying? It will use invalid values and calculate the wrong result. To overcome this, you use a *tile_barrier*. The tile barrier is available as a property of the tiled index passed in to the kernel, and it has a method, *wait()*, that causes this thread to wait until all the threads in this tile have reached this point. After the barrier, all the threads have finished copying values to *tile_static* memory. The value written by one thread is visible to (and therefore can be used by) other threads in the same tile.

Given a tiled index *tidx*, waiting for all the threads in this tile to reach the same line is a simple call:

```
tidx.barrier.wait();
```

You must structure your kernel carefully when using tile barriers. If you have a lot of branching, it must not be possible for some branches to bypass a *wait*. Nonuniform branching, in general, hurts performance on accelerators—they are set up for high performance when doing the same operation on a large number of threads. But branching past a *wait* is actually forbidden. Consider this example (which is not allowed):

```
for (int i = 0; i < W; i += TileSize)
{
   // . . .
   if (somecondition)
   {
       // . . .
       tidx.barrier.wait();
   }
   else
   {
       // . . .
   }
}
```

To resolve this issue, move the *wait()* out of the *if* and wait regardless of whether the *if* body or the *else* body was executed. Similarly, you can't return early from your kernel if there is a *wait()* later in the code, and you can't set up a loop with a *wait()* inside it if not all threads enter the loop. More details on barriers and on best practices for using *wait()* can be found in the Barriers section of Chapter 7, "Performance."

Adding tile barriers, the matrix multiplication algorithm above becomes (for each thread):

- Copy one element to *sA*.
- Copy one element to *sB*.
- Wait to make sure all threads in this tile have finished copying, meaning *sA* and *sB* are complete.

- Calculate a partial sum toward the result in this thread's element of the *C* array.
- Wait to make sure all threads in this tile have finished calculating, meaning *sA* and *sB* are safe to overwrite.
- Move to the "next" tile-sized square and repeat.

The storage specifier *tile_static* should remind you that *sA* and *sB* have a scope and lifetime that matches the tile—it is larger than the *for* loop in which you have seen these instances declared. Each pass through that loop (each move to the next tile-sized square) is working with the very same *sA* and *sB* as the previous iteration. Moving to the next square too soon would overwrite *sA* and *sB* before they had been used to update the partial sum completely.

Completing the Modification of Simple into Tiled

Here is the actual code for tiled matrix multiplication. It uses a tile size of 16, which gives a much bigger benefit (each thread uses 16 values from *sA* and 16 from *sB*) than a tile size of two. Two *wait()* calls are required to implement the algorithm described above: one to ensure that all threads have finished copying and a second one to ensure that all threads have finished calculating before moving on to the next tile.

```
static const int TileSize = 16;

void MatrixMultiplyTiled(std::vector<float>& vC,
    const std::vector<float>& vA,
    const std::vector<float>& vB,
    int M, int N, int W)
{
    array_view<const float,2> a(M, W, vA);
    array_view<const float,2> b(W, N, vB);
    array_view<float,2> c(M, N, vC);
    c.discard_data();

    parallel_for_each(c.extent.tile< TileSize, TileSize >(),
        [=] (tiled_index< TileSize, TileSize> tidx) restrict(amp)
    {
        int row = tidx.local[0];
        int col = tidx.local[1];
        float sum = 0.0f;
        for (int i = 0; i < W; i += TileSize)
        {
            tile_static float sA[TileSize][TileSize];
            tile_static float sB[TileSize][TileSize];
            sA[row][col] = a(tidx.global[0], col + i);
            sB[row][col] = b(row + i, tidx.global[1]);

            tidx.barrier.wait();

            for (int k = 0; k < TileSize; k++)
                sum += sA[row][k] * sB[k][col];

            tidx.barrier.wait();
        }
```

```
        c[tidx.global] = sum;
    });
    c.synchronize();
}
```

Because the tile size is kept in a constant, you could try running this repeatedly on large matrices to see the effect of changing tile size.

Effects of Tile Size

For the matrix multiplication example used in this chapter, the larger the tiles, the more times each value in the *tile_static* memory gets used. So the obvious question is: should you automatically use the largest possible tile size?

To test tile size, here is the structure of a simple application that initializes arrays with random float values and runs the same calculation multiple ways:

```
int main()
{
    const int M = 64;
    const int N = 512;
    const int W = 256;

    std::vector<float> vA(M * W);
    std::vector<float> vB(W * N);
    std::vector<float> vC(M * N);
    std::vector<float> vRef(M * N);

    std::random_device rd;
    std::default_random_engine engine(rd());
    std::uniform_real_distribution<float> rand(0.0f, 1.0f);

    std::generate(vA.begin(), vA.end(), [&rand, &engine](){ return rand(engine); });
    std::generate(vB.begin(), vB.end(), [&rand, &engine](){ return rand(engine); });

    // Calculate a reference result on the CPU for comparison.
    for (int row = 0; row < M; ++row)
    {
        for (int col = 0; col < N; ++col)
        {
            float result = 0.0f;
            for (int i = 0; i < W; ++i)
            {
                int idxA = row * W + i;
                int idxB = i * N + col;
                result += vA[idxA] * vB[idxB];
            }
            vRef[row * N + col] = result;
        }
    }

    MatrixMultiply(vC, vA, vB, M, N, W);
```

```
    MatrixMultiplyTiled(vC, vA, vB, M, N, W);

    return 0;
}
```

(For space reasons, the code to query performance counters and report results was elided in this example, along with code to check that the results are the same in all three calculations.) This code was run for a variety of matrix sizes and tile sizes. The largest possible tile size here is 32 because $32 \times 32 = 1{,}024$, which is the limit for the product of the tile dimensions. On a system with a fairly ordinary video card, these times were recorded (in milliseconds):

	C++ AMP, simple	Tiled, TileSize=4	Tiled, TileSize=8	Tiled, TileSize=16	Tiled, TileSize=32
M=64, N=4096, W=64	17	39	14	13	13
M=128, N=4096, W=128	33	135	30	25	26
M=256, N=4096, W=256	90	522	96	73	80
M=512, N=4096, W=512	307	2015	330	235	266

Here are a few notes on timing, which will be addressed more thoroughly in Chapter 7. First, any time over two seconds is not meaningful; by default, the process is stopped after two seconds of execution on the GPU. (You can learn more about this time-out in the Time-Out Detection and Recovery section of Chapter 12, "Tips, Tricks, and Best Practices." Second, when running executables to time them, be sure to run them several times in quick succession. The first run is usually much slower than subsequent ones. Ignore any timing results from your first runs. Chapter 7, "Performance," and Chapter 8, "Performance Case Study," have more details on how to time your code correctly.

The CPU implementation here is simple and doesn't use memory very efficiently, so no CPU results are included. With these numbers, you can compare simple (nontiled) C++ AMP results with those for various tile sizes.

The first conclusion is that tile sizes that are too small can produce results that are worse than the simple case. Tiles of 4×4 do not reuse the *tile_static* data enough to make up for the cost of copying, and they make inefficient use of the GPU architecture by not occupying a full multiprocessor. For the arrays used in these sample runs, tiles of 8×8 manage to just break even, although because the code is more complex, it doesn't seem worth the effort. Tiles of 16×16 look like the sweet spot from this set of runs, with a definite improvement over the simple solution. It seems—although more runs would be required to be sure—that 32 is consistently a little worse than 16 for array sizes in this range. For a different problem or even a different set of array sizes, you'll see different optimal tile sizes. It's even possible that you might find different optimal tile sizes on different video cards, which can make choosing tile size rather challenging.

What won't vary from problem to problem is the poor performance that results from choosing a tile size of 4×4 or 8×8. To understand why, consider two aspects of GPU and C++ AMP architecture that were mentioned in Chapter 1, "Overview and C++ AMP Approach."

A GPU typically queues thousands of threads for work and does so in clumps or bundles. NVIDIA calls these a *warp*, for example, and they are 16 or 32 threads. AMD calls them a *wavefront*. The word "warp" will be used here.

Elements of an array that differ only in their least significant index are next to each other in memory (for example, *A00* and *A01* in the diagrams in this chapter).

The fastest and most efficient arrangement for a GPU is when all the threads in the warp are accessing consecutive memory and performing the same operations on that memory. If the number of threads in the tile is smaller than the warp size, then different threads within the warp are in different tiles and are accessing entirely different areas of memory; the threads have diverged. This is inefficient. Different hardware devices have different warp sizes—typically 32 and 64—but in the future this may change and 64 will probably be more common. As a result, you will not see top performance when you choose a tile size that makes the number of threads per tile (the product of the dimensions) smaller than 32 or 64. The same effect can happen with tiles where the number of threads is not a multiple of 32 or 64 because they will involve "remainders" that take up only part of a vector-wide warp. For example, 40 threads in a tile might mean 32 threads in one warp and 8 in another. That causes some of the same issues as a tile size of four.

Further, if the tile size in the least significant dimension is at least the size of the warp (mostly 32 today and 64 in the future), all the threads in the warp will be accessing consecutive memory locations, and that's the sweet spot for GPU work. In the examples shown here, both the 16×16 and 32×32 tile-size runs gain this benefit, which you can see is significant. When you use the right tile size, your tiled calculations will be much faster than your nontiled ones, and that gives you the maximum performance gain from using C++ AMP.

Choosing Tile Size

Always choose your tile size so that the number of threads in the least significant dimension is at least 16, and use 32 or even 64 if you can. This maximizes the access to consecutive memory locations and provides a significant performance benefit. For the specific case of matrix multiplication, a similar logic leads to setting the most significant tile dimension to 16 or 32. This is because of the way the algorithm accesses the elements moving down a column as well as across a row. That would not apply to all algorithms. Because 32×32 is the maximum possible tile size (1,024 total dimension product) you have really only four choices when handling matrix multiplication: 16×16, 32×16, 16×32, or 32×32. For other algorithms you might have more options. To choose among them, do test runs on the hardware you're most likely to be targeting, using a realistic workload, and test whether one choice outperforms the others.

One constraint to keep in mind is that the *arrays* and *array_views* you are using on the accelerator must have extents that are multiples of the tile size. If you can choose the size of your datasets then this isn't an issue, but in most cases you can't. The larger the tile size, the more likely there is to be a mismatch between tile size and array extent. Choosing an invalid tile size will not cause a compile error even when the sizes of your arrays are known at compile time, but it will cause a run-time

error. It is up to you to make sure that you construct *arrays* (and *array_views*) whose dimensions are multiples of the tile size.

If you've chosen your tile size, it's a constant known at compile time, and your application sets the array size at run time (by asking the user for number of points to use, by reading in data to be processed from a file or device, by reacting to events it is monitoring, and so on), then how can you prevent a mismatch? You can use several techniques:

- Design your system so that the size of the array will be a multiple of the tile size. For example, when generating or sampling data and allowing the user to choose the sample size, offer only choices that are a multiple of the tile size. If you have 16×16 tiles, for example, provide a slider with scaled-down values and multiply the value selected on the slider by 256 to produce the total size of your dataset.
- Calculate extents that are larger than your sample set and are multiples of the tile size. For example, if you have a total tile thread count of 256 and a device has captured 200 points, set up an array of 256. If 4,000 points have been captured, set up an array of 4,096. Adapt your kernel to handle the possibility that some points in the array are outside the problem space.
- Calculate extents that are smaller than your sample set and are multiples of the tile size. For example, if a device has captured 300 points, set up an array of 256. Handle the "leftover" points with a simple (nontiled) *parallel_for_each* or a CPU-based calculation. This doesn't require any signal values but might require you to retouch (on the CPU) every point in your results to account for values beyond those used on the accelerator.

Obviously, the first approach is the simplest; use it whenever you can. The second and third techniques are illustrated in Chapter 12.

There is one more aspect to choosing a tile size that makes the decision even more complicated. Sometimes called occupancy, this has to do with what fraction of resources a tile of thread consumes. Consider *tile_static* memory, for example. If a tile needs 60 percent of the GPU cache for *tile_static* memory, then two tiles can't run at once. If you have eight tiles, the total time will be eight times the time taken for one tile. But if you make the tiles half the size so that you have 16 of them and they need 30 percent of the resources, now three tiles can run at once, meaning it will take about five times as long as it did for one tile. This is a large performance gain from making a tile smaller. It's almost impossible to tell by just looking at your code if this is a possibility for your particular application or not, and it could apply to just one of the possible hardware configurations you're targeting. Profiling, as covered later in this book, can help, as can performance testing on a variety of hardware. In the end, you are likely to choose a tile size that you believe is probably the best, but absolute certainty is hard to come by. See "Occupancy and Registers" in Chapter 7 for further discussion of occupancy.

Summary

Tiling is a powerful technique that can improve the performance of your application by another factor of two or more above the gains you got from simply moving to an accelerator with *parallel_for_each*. By accessing the programmable GPU cache, you can save the time spent on fetching data from memory dramatically. There are a few caveats, of course. Tiling applies only to grids of one, two, or three dimensions. If you're using arrays of higher rank, you can't tile your calculations. More important, just switching to a tiled grid and a tiled index doesn't affect performance; you need to rewrite your algorithm to work with a small local programmable cache. This will make your code harder to write, to read, and to debug. It can be difficult to visualize how a tiled calculation is working or to adapt your calculations to be broken apart into tiles in this way.

Exactly what gain in performance you can expect for your effort is also difficult to determine. It might vary depending on the hardware that ends up running your application. Small differences in one area, such as the size of that programmable cache, might make a big difference in overall execution time. It's even possible, especially for a poorly chosen tile size, to make your performance worse by used a tiled approach, although that's not the norm when tile sizes are chosen with the hardware in mind. Always test both simple and tiled algorithms to be confident you are making the right decision.

CHAPTER 5

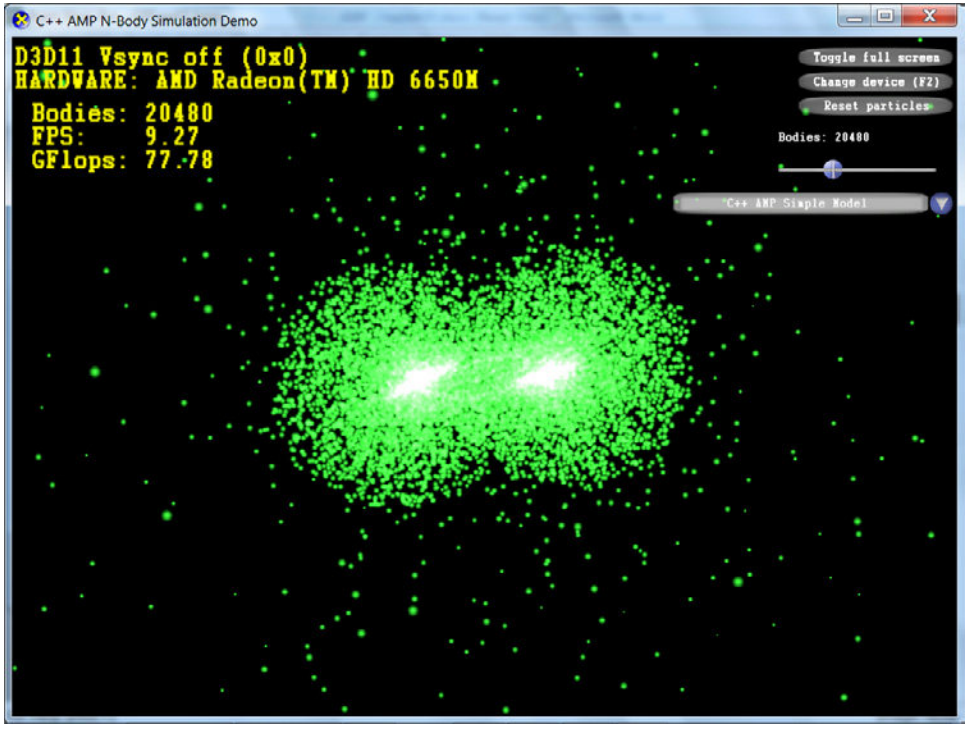
Tiled NBody Case Study

In this chapter:

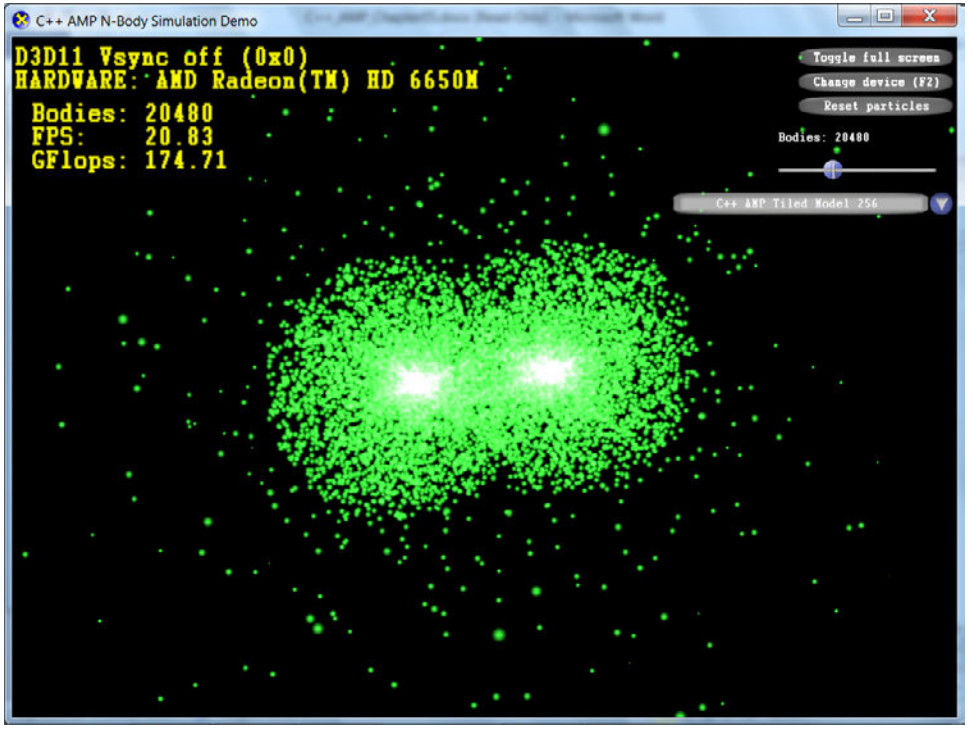
How Much Does Tiling Boost Performance for NBody? 83
Tiling the n-body Algorithm 85
Using the Concurrency Visualizer 90
Choosing Tile Size 95
Summary 99

How Much Does Tiling Boost Performance for NBody?

The NBody sample was described in some detail in Chapter 2, "NBody Case Study," so that background will not be repeated here. But the tiled versions were not covered. These are available in the drop-down list below the slider to set particle size. Here's a comparison of two runs, both for the same number of particles. First, the C++ AMP simple model:



Second, the C++ AMP tiled model:



The numbers in the drop-down list refer to the tile size. The differences between the tile sizes are explored later in this chapter; they are dwarfed by the difference between the simple option and any of the tiled options. The speed is more than doubled, and you might see a more dramatic difference if you drag the slider to work with more and more particles. For example, tests on a variety of hardware with 58,368 particles (the highest value on the slider) resulted in as much as five times the GFlops for the tiled 256 case compared to the simple algorithm. Clearly, a speedup of that order is worth a little complexity in your code.

Tiling the n-body Algorithm

Both the simple and tiled algorithms are invoked by the *OnFrameMove()* method in NBodyGravityAmp.cpp. This code calls *Integrate()* on an instance of the *INBodyAmp* class, or more specifically an instance of a class derived from *INBodyAmp*:

```
void CALLBACK OnFrameMove(double fTime, float fElapsedTime, void* pUserContext)
{
    g_pNBody->Integrate(g_deviceData, g_numParticles);
    std::for_each(g_deviceData.begin(), g_deviceData.end(), [] (std::shared_ptr<TaskData>& t)
    {
        std::swap(t->DataOld, t->DataNew);
    });
    std::swap(g_pParticlePosOld, g_pParticlePosNew);
    std::swap(g_pParticlePosRvOld, g_pParticlePosRvNew);
    std::swap(g_pParticlePosUavOld, g_pParticlePosUavNew);
    g_camera.FrameMove(fElapsedTime);
}
```

The variable *g_pNBody* holds a pointer to the *INBodyAmp*-derived class that does the actual work, in the *Integrate()* function, of determining the new acceleration, velocity, and position of each element in the large array of particles. This pointer is set by a call to *NBodyFactory()* in *OnD3D11CreateDevice()* and again whenever the user makes a choice in the drop-down. *NBodyFactory()* is discussed in more detail later in this chapter.

For a single-accelerator system, the *INBodyAmp*-derived class will be *NBodyAmpTiled*. There is a similar class called *NBodyAmpMultiTiled* that will be used only on systems with multiple accelerators; for example, a system with two video cards. That class is not discussed in this chapter, and all the multiaccelerator code is not discussed here either. When the compute type is not multitile, this code swaps the old and new particle values to get ready for another iteration.

The NBodyAmpTiled Class

Chapter 2 described the *INBodyAmp* base class and the classes that inherit from it. The implementations of the NBody classes in the two projects, NBodyGravityCPU and NBodyGravityAMP, are different. Two classes in NBodyGravityAMP that have not yet been described are *NBodyAmpTiled* and *NBodyAmpMultiTiled*. These both inherit from *INBodyAmp* and are both implemented as template

classes, taking the tile size as an integer template parameter. The class definition for *NBodyAmpTiled* starts like this:

```
template <int TSize>
class NBodyAmpTiled : public INBodyAmp
{
private:
    float m_softeningSquared;
    float m_dampingFactor;
    float m_deltaTime;
    float m_particleMass;
    static const int m_tileSize = TSize;

    // .. rest of class elided
};
```

The class is implemented as a template in order to minimize the amount of code to read and maintain while allowing the user to choose the tile size at run time. The tile size must be a compile-time constant, and with this approach, it is. Although you could just copy some member functions such as *TiledBodyInteraction()* and call different versions of the function for different sizes, this would lead to maintenance problems, such as making the same change in all the copies of the function whenever changes are needed. Using templates here reduces the amount of code to write, read, and maintain. The advantages of this approach are discussed more completely in the section of this chapter entitled "Choosing Tile Size."

NBodyAmpTiled::Integrate

The *Integrate()* method in *NBodyAmpTiled* simply calls *TiledBodyBodyInteraction()*, which does what the *Integrate()* method in *NBodyAmpSimple* does: it sets up the data structures and performs the calculations using a *parallel_for_each*. It has been pulled into a separate function in order to simplify the multiaccelerator code. *TiledBodyBodyInteraction()* is quite general, taking an offset into the particle arrays to start at, the number of particles to use starting from that offset, and the number of particles overall:

```
void TiledBodyBodyInteraction(const ParticlesAmp& particlesIn, ParticlesAmp& particlesOut,
    int rangeStart, int rangeSize, int numParticles) const
{
    // ...
}
```

TiledBodyBodyInteraction() calculates new positions, velocities, and accelerations of the particles at index *offset* to *offset* + *rangeSize* in *particlesIn* and stores the results in *particlesOut*.

In the single-accelerator case presented here, *Integrate()* calls *TiledBodyBodyInteraction()* like this:

```
TiledBodyBodyInteraction(*particleData[0]->DataOld, *particleData[0]->DataNew,
                         0, numParticles, numParticles);
```

In other words, for the single-accelerator case, inside *TiledBodyBodyInteraction() offset* will be zero and both *rangeSize* and the local parameter *numParticles* will be the same *numParticles* that was

passed to *Integrate()*, which in turn is *g_numParticles*, the number that the user selects with the slider. (The difference between *rangeSize* and *numParticles* in the multiaccelerator case is that *rangeSize* refers to the number of particles whose new positions are being calculated on this accelerator, whereas *numParticles* refers to the number of particles whose contributions to that new position must be considered. That's why they are the same in the single-accelerator case.)

There are three significant differences between the simple code and the tiled code:

- The *parallel_for_each* is tiled.
- Particle positions are cached in *tile_static* memory, and the calculations are adjusted to reuse the *tile_static* values as much as possible.
- The innermost loop is unrolled.

Each of these differences will be discussed in turn.

As mentioned in Chapter 4, "Tiling," a tiled *parallel_for_each* does not provide significant performance benefit on its own; it only provides benefit because it allows the use of *tile_static* memory, a small programmable cache on the GPU with 100× faster access than the global memory of the GPU where *array* and *array_view* data is kept. You can't use *tile_static* memory in a simple *parallel_for_each* or anywhere else; you can only use it in a tiled *parallel_for_each*.

A tiled *parallel_for_each* takes a *tiled_extent* instead of an ordinary *extent*. The *tile()* function of the *extent* class gets you a tile of the same rank as the extent. The relevant lines in *TiledBodyBodyInteraction()* are the following:

```
extent<1> computeDomain(rangeSize);
// . . .
parallel_for_each(computeDomain.tile<m_tileSize>(), [=] (tiled_index<m_tileSize> ti)
    restrict(amp)
{
    // . . .
}
```

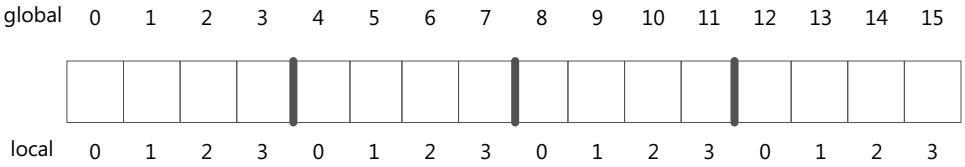
This sets up the *parallel_for_each* using the tile size kept in a member variable, which in turn came from the template parameter. Because the particles are kept in one-dimensional arrays, the extent and the tile are also one-dimensional.

This now enables the use of *tile_static* memory and the corresponding performance improvements. One approach might be to load all the particles into *tile_static* memory because each particle's calculation needs all the other particles' information. However, the number of particles is too large for this to be feasible. Some smaller number of values will need to be copied into *tile_static* memory. If that number of values is not a multiple of the tile size, some threads in each tile will not copy as many values as others, which will waste some capacity. If the number is a multiple of the tile size, it must go evenly into the total number of particles, which might add some complexity to your code. (This application limits the number of bodies that can be set with the slider to specific values that are a whole number of tiles, but that doesn't guarantee that any multiple of the tile size will also go evenly into the total number of particles.) Copying the same number of values as the tile size ensures that

the number of values to be copied will go evenly into the total number of particles, and to use more *tile_static* memory, you need only increase the tile size.

What should be copied into *tile_static* memory? The formula for the body-body interaction involves only the positions of the other particles; their velocity and acceleration do not affect the other particles. Only particle positions need to be cached into *tile_static* memory.

The algorithm needs to be adapted a little to use the local and global indexes into the particle *array*. If you imagine a much smaller particle *array* of only 16 particles, divided into tiles of 4, the indices would look like this:



The calculations in *TiledBodyBodyInteraction()* use *tile_static* memory repeatedly. Each thread is calculating the acceleration, velocity, and position of one particle at position *idxGlobal* in the particle array. To do so, it iterates through the tiles as a way to iterate through all the other particles that affect the particular particle this thread is considering. The pattern is much like the one in the matrix multiplication example; each thread in the tile copies one value to *tile_static* memory. Then, after it is used, the calculation moves over one tile and repeats. Consider a thread that is calculating the acceleration, velocity, and position of particle 5 in the diagram above. Its local index is 1. This thread will do the following work:

1. First, work on tile 0.
2. Copy particle 1's position to *tile_static* memory.
3. Wait until all the other threads in the tile have done their copies and the positions of particles 0 through 3 are in *tile_static* memory.
4. Use all four of the *tile_static* values, determining their contributions to the acceleration of particle 5.
5. Wait until all the threads in the tile have used particles 0 through 3.
6. Move to tile 1.
7. Copy particle 5's position to the *tile_static* memory.
8. Wait.
9. Use the copied positions of particles 4 through 7, still determining their contributions to the acceleration of particle 5.
10. Wait.
11. Move to tile 2.

12. Copy particle 9's position.

13. Wait.

14. Use the copied positions of particles 8 through 11.

15. Wait again.

This pattern continues until this thread has used the position of every particle in the array to determine the acceleration of particle 5.

The code in *TiledBodyBodyInteraction()* inside the *parallel_for_each* is as follows:

```
tile_static float_3 tilePosMemory[m_tileSize];

const int idxLocal = ti.local[0];
int idxGlobal = ti.global[0] + rangeStart;

float_3 pos = particlesIn.pos[idxGlobal];
float_3 vel = particlesIn.vel[idxGlobal];
float_3 acc = 0.0f;

// Update current Particle using all other particles
int particleIdx = idxLocal;
for (int tile = 0; tile < numTiles; tile++, particleIdx += m_tileSize)
{
    // Cache current particle into shared memory to increase IO efficiency
    tilePosMemory[idxLocal] = particlesIn.pos[particleIdx];
    // Wait for caching on all threads in the tile to complete before using the data.
    ti.barrier.wait();

    for (int j = 0; j < m_tileSize; )
    {
        BodyBodyInteraction(acc, pos, tilePosMemory[j++], softeningSquared, particleMass);
    }

    // Wait for all threads to finish calculating before a new tile starts.
    ti.barrier.wait();
}

vel += acc * deltaTime;
vel *= dampingFactor;
pos += vel * deltaTime;

particlesOut.pos[idxGlobal] = pos;
particlesOut.vel[idxGlobal] = vel;
```

This is a very common pattern when writing C++ AMP code that takes advantage of tile static memory. The first half of the tiled *parallel_for_each* uses each thread in the tile to load data into tile static memory. A barrier is then used to make every thread in the tile wait for all copies to complete before moving on to the calculation that uses all the tile static data. Finally, a second barrier prevents some threads from starting the next loop through the *parallel_for_each* and overwriting data still being used by other threads performing the calculation. You will spot this pattern in most C++ AMP applications that use tile static memory, including the ones in this book.

This change improves the performance significantly. For a further boost, you can experiment with unrolling the loop through each tile. There is a performance cost for the *if* statement that compares *j* with *m_tileSize*. For example, if you are sure that *m_tileSize* is even, you could write the loop like this:

```
for (int j = 0; j < m_tileSize; )
{
   BodyBodyInteraction(acc, pos, tilePosMemory[j++], softeningSquared, particleMass);
   BodyBodyInteraction(acc, pos, tilePosMemory[j++], softeningSquared, particleMass);
}
```

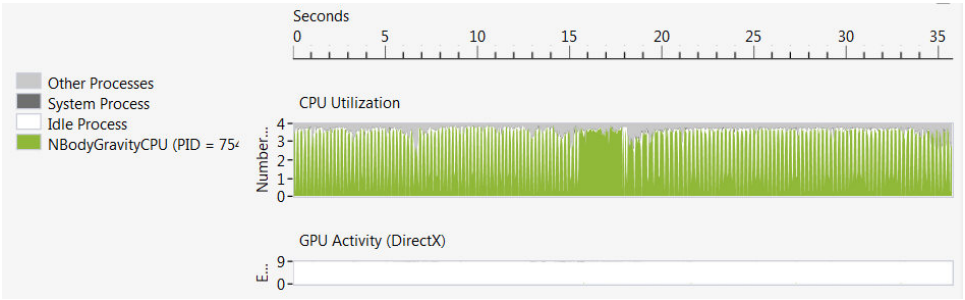
This would cut the number of tests of *j* against *m_tileSize* in half for a free performance boost. Some experimentation (which you can reproduce yourself using the sample code) shows that having four identical statements in the if body gives performance improvements over two, but that further increases to eight and above do not improve performance any further. These experiments can also serve to demonstrate an important principle: do not rely on C++ AMP alone to increase the speed of your calculations. Hand optimizations like this, which draw on your knowledge of the properties of the application (in this case that tile size is always even and will typically be a multiple of 8 or 16), allow improvements in performance and should not be neglected.

Using the Concurrency Visualizer

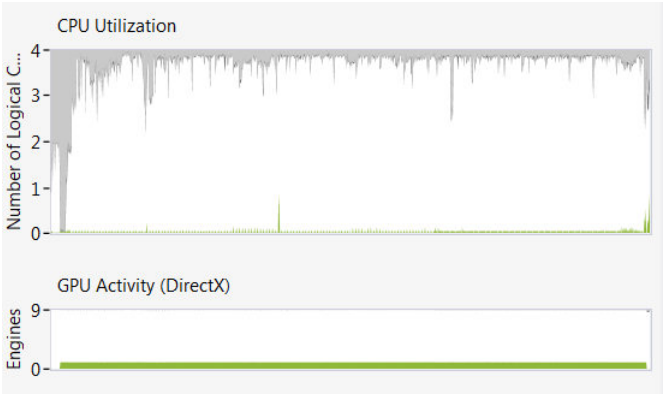
Microsoft Visual Studio 2012 includes changes to both the debugger and the Concurrency Visualizer to work with C++ AMP applications and provide insight into what is happening on the accelerator as well as on the CPU. The Concurrency Visualizer can show you where the bottlenecks in a given algorithm are happening or let you compare different aspects, such as the time spent copying data to an accelerator and the time spent executing a calculation.

To use the visualizer, you need to start by collecting some data. Use a release build of the application (you don't need to know about bottlenecks in a debug version, and the performance of a debug build is often radically different from that of a release build). You can launch it from within Visual Studio using Analyze | Concurrency Visualizer | Start With Current Project. If you want to omit the startup activities from the trace, you can run the application and attach to it instead. Run the application, select AMP Simple in the drop-down box, increase the number of particles until the movement noticeably slows, and then let the application run for a few seconds. In Visual Studio, on the Analyze menu choose Concurrency Visualizer, then Attach To Process. In the dialog box, choose NBodyGravityAMP and watch it run for a minute or so. Make a note of the frame rate and the GFlops. Reset the particles if you like. Then use the drop-down list to change to a tiled option; make a note of the tile size. Wait until it settles to a steady speed and then make a note of the frame rate and GFlops again. Switch focus back to Visual Studio and click on the Stop Collection link in the collection pane. You may now close the application.

Repeat this process for NBodyGravityCPU for extra insight. Here is a short (just over 30 seconds) trace of NBodyGravityCPU shown in the Utilization View. This run used the CPU Advanced option, which leverages PPL to use all the CPU cores.



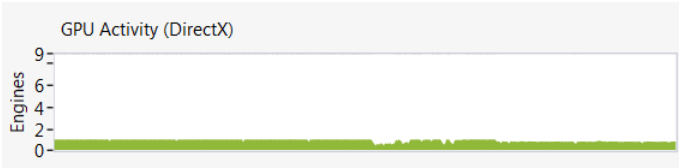
For comparison, here is a trace of NBodyGravityAMP over roughly a minute—30 seconds or so of simple C++ AMP and 30 seconds or so of tiled:



Without knowing anything at all about the Concurrency Visualizer, you can instantly conclude that these are very different traces. The first one has a large green band in the upper graph, which is labeled CPU Utilization. The CPU is almost entirely utilized for most of the run. On the second trace, the CPU is being used far less, and the chart labeled GPU Activity has a fairly solid green band. The scale on the GPU activity is zero to nine engines in these images—the scale might differ somewhat on other hardware depending on the card or cards in use. It's not feasible to keep all your GPU engines occupied at once because they serve different purposes. For example, C++ AMP uses the 3D engines but not the video engines, which are fixed-function. You can't control which engines are used by the C++ AMP parts of your application. Looking at this graph, it's clear that one engine is being kept busy and that the load on the CPU has been dramatically reduced.

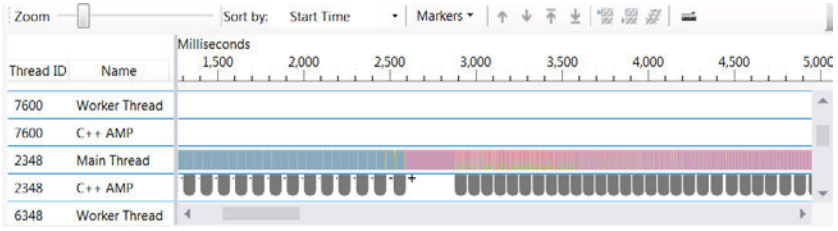
What else can this view of the Concurrency Visualizer tell you about your application? If you do several runs and glance at a stopwatch or some other clock that shows you elapsed seconds while you interact with the application, you'll be able to match shapes on the traces with actions you took at a particular time when you were running the application. For example, the gray spikes coming down from the top in the trace correspond to a time when you use the UI of the NBody application—to reset the particles, to make a choice from the drop-down list, to change the number of particles with the slider, and so on. To compare the simple and tiled performance then, zoom in on the area containing the gray spike that marks the switch from Simple to Tiled calculations.

Click before the spike right in the body of the graph and drag to the right, selecting a portion of the graph. Release when you are past the spike. Close any Visual Studio windows that are taking up vertical real estate, such as an output view at the bottom of the main pane, to ensure that as much space as possible is given to the GPU Activity graph. You are likely to see a change in the green bar across the bottom. Zoom in a little closer on that area and you might see something like this:

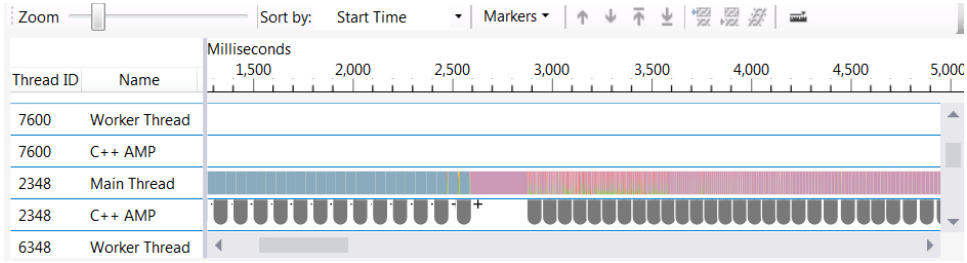


You can see that there is a qualitative difference between the two ends of this graph. On the left, activity is pretty much pegged at one engine for a long stretch of time. On the right, there are more frequent occurrences of a drop in GPU activity.

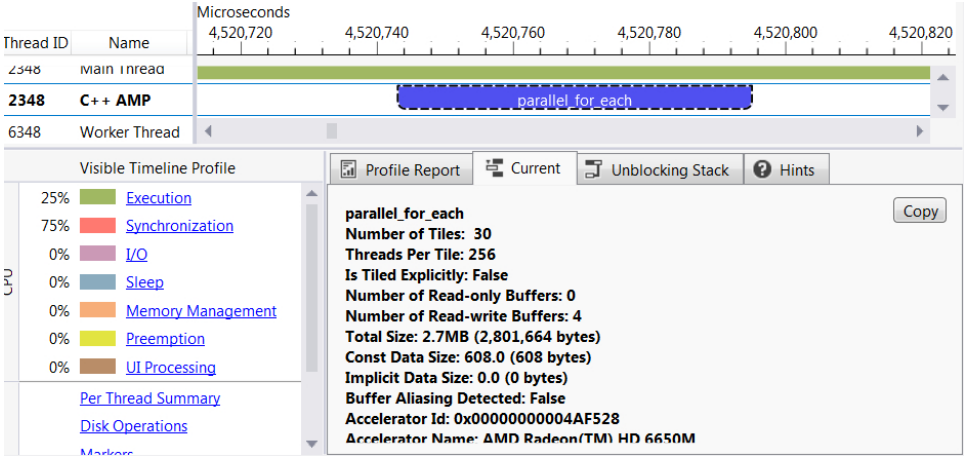
Can you conclude that having the gaps closer together means the GPU is getting its work (calculating the new acceleration, velocity, and position of all the particles) done more quickly in the tiling case? Yes. For real insight into how the application is behaving, it's best to use another view of the Concurrency Visualizer. The images presented so far are of the Utilization view; there is also a Threads view and a Cores view. Zoom out to the whole trace using the slider above the graphs and then click on the Threads button just above the slider. This produces a view which, at first glance, is dense with information and in dire need of the Demystify button you see in the upper right. But don't click that button just yet. Here is the threads view of the same trace as the Utilization view images above:



Look at the lower lane labeled C++ AMP. The symbols in this lane are called spans. They have rounded ends and indicate a length of time. Sometimes they are quite long and other times, as in this image, they are drawn with no middle—just the two rounded ends—to indicate that the span is too short to draw to scale. Gray spans indicate the presence of several spans that are too close together to draw separately at this scale. Blue spans are single spans. You can clearly see that the spans are closer together in the second half of the trace than they are in the first half. The tiling part of the run is getting its work done more quickly than the simple part of the run at the beginning. These results make a lot more sense if you zoom in. Click and drag to zoom into an area that contains about three of these spans early in the run. You will see a few gray spans and some blue ones with "..." on them. Click on one of the gray ones to fill in the pane below with more details—for example, that there are three spans in this region. Click on one of the blue ones and you'll see it describes a *parallel_for_each* or other C++ AMP span. Here's an example with a blue span from the simple (nontiled) portion of the run:



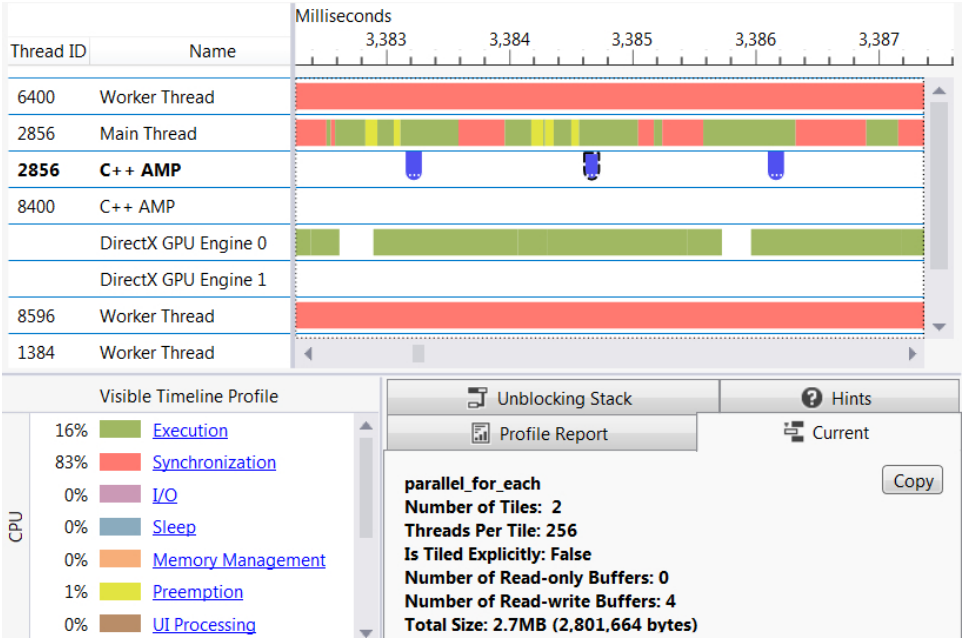
The information in the lower-right pane identifies the number of tiles (even when you do not tile your calculations, they are tiled implicitly but will not take advantage of tile static memory) and the threads per tile, along with other useful information, including the accelerator details. Things really get interesting when you zoom even further: click and drag a range that includes just one of the blue spans. Keep zooming until the "…" is replaced with information about the span. Eventually, you should see something like this:



Now you can see that the length of the blue bar represents the time the *parallel_for_each* took on the CPU. It's not necessarily the time it took to execute—the work is scheduled onto the accelerator and then the CPU work carries on with whatever comes after the *parallel_for_each*. The various colors in the main thread represent some of the other work that needs to be done, including copying values back from the accelerator to use them (synchronization). The bottom-right pane has information about the area you have selected in the top half of the view. At the very bottom of this pane is a small scrollable area that includes information like the length of the *parallel_for_each*.

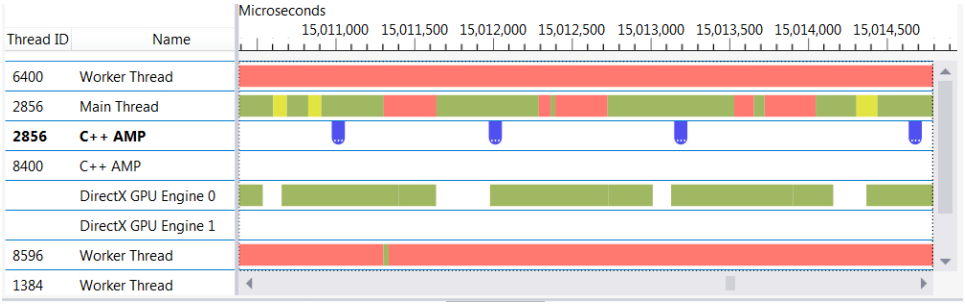
You will gain a better understanding of the execution of your application if you remove extraneous information from this view. Click on one of the worker threads below C++ AMP in the diagram

that has a colored bar in it. In the bottom right pane, on the Current tab, if you see lines that start kernel32.dll and d3d11ref.dll, these threads are not running your code. (There might be threads running video drivers as well; look for dynamic-link library [DLL] names associated with your video card driver, such as atidxx32 or other DLLs that start ati for AMD cards and DLLs that start nv for NVIDIA cards.) To remove this lane, right-click the thread in the list at the top left and choose Hide Selection. Do the same for any Worker Thread lanes with no colored bars or spans in them over the entire run. When you have hidden enough lanes, all the lanes can be seen at once, including those for the GPU engines. Now zoom in on a few of the spans from the simple section of the run:



The green bars on the GPU Engine 0 lane indicate execution, and you can click on any bar for more details. One of the first things you'll notice is that even though there is a regular pattern of green GPU execution bars, they don't start at the same time as the blue spans that represent a *parallel_for_each*. That's because the commands are queued on the GPU, and the *parallel_for_each* doesn't wait until the command finishes or even until it starts. Once the command is sent, then as far as the *parallel_for_each* is concerned, that is all. The main thread moves on to something else. In many applications, this would immediately be followed by a pink synchronization event while the main thread waited for the answers to come back from the accelerator. But as you might recall from Chapter 2, for the NBody simulation the particle information is left on the GPU, where it can be used in the rendering without copying it back to the CPU just to copy it to the GPU for rendering purposes.

If you now scroll in this threads view using the scroll bar in the middle of the screen, watch the overview (called the Utilization Navigator) at the top to see when you've reached the part of the run that involved a tiled calculation. Don't change your zoom level. You'll immediately see that the *parallel_for_each* spans are much closer together:



You can also see that the regular pattern of the green bars and blue *parallel_for_each* spans is less predictable in this part of the run. This is again because of the queuing of commands to the accelerator. You could suppress this by using an *accelerator_view* with a queuing mode of immediate, but as discussed in Chapter 3, "C++ AMP Fundamentals," this might hurt your performance. You should know that under these circumstances the length of time reported for the *parallel_for_each* is not really relevant, since it returns immediately and the work continues on the GPU after the *parallel_for_each* appears to have finished. The lengths of the green GPU execution bars are more significant. Notice they are about half the length here in the tiled section of the run as they were in the simple section of the run; this matches up well with the doubled frame rate and GFlops for the tiled section at this number of particles.

There are other ways you can use the Concurrency Visualizer to understand the performance of your application. For example, if this algorithm copied the results back from the accelerator to the CPU and you saw that the copy time was 8 or 10 times the length of the green execution bar, you would know that efforts to further reduce the GPU execution time (by loop unrolling, using more *tile_static* memory, or something similar) would be unlikely to have a significant performance benefit. In a way, this is Amdahl's Law again—the "sequential part" is now the copy between the CPU and the GPU, and it can dominate the total time of execution on the accelerator as you shrink the time spent actually calculating the results.

Further use of the Concurrency Visualizer and ways other than tiling to improve the performance of your application are the subjects of Chapter 7, "Optimization."

Choosing Tile Size

The sample used in this chapter has a drop-down list to let you experiment with different tile sizes. On a single machine with a fairly ordinary video card, here is a summary of the GFlops achieved for various numbers of particles:

# particles	Simple	TS=64	TS=128	TS=256	TS=512
20,480	73	154	157	154	150
35,328	81	171	174	176	163
55,808	82	177	180	180	171

If you want to reproduce these results, be sure to let the application run without doing anything else on the machine. You should also ensure that the application has focus because this will affect performance. You will see the frame rate and GFlops change quite a lot (for example, GFlops for the simple case, 20,480 particles varies from 56 to 85) but eventually settle at a steady number for many seconds at a time. The number reported here is neither an average nor what GFlops peaks at, but rather this steady number to which it regularly returns. On a machine with a different video card you will see different numbers, but you should see a similar pattern.

What can you conclude from these results? Primarily, that the n-body algorithm is not particularly sensitive to tile size as long as you choose a reasonable one; for example, not 4 or 8. The noise in the GFlops and frame rate numbers exceeds the differences between the columns for different tile sizes.

When choosing a tile size, one thing to remember is the requirement that the extent of the array or *array_view* you're using should be a multiple of the extent of the tile. Whenever possible, you can ensure this by taking more control over the size of the array or *array_view*. For example, this NBody sample ensures that the number of particles is a multiple of 512. In the *OnGUIEvent()* method handler in NBodyGravityAmp.cpp, the case statement that handles the slider being moved to a new value reads as follows:

```
case IDC_NBODIES_SLIDER:
    {
        CDXUTSlider* pSlider  = static_cast<CDXUTSlider*>(pControl);
        g_numParticles = pSlider->GetValue() * g_particleNumStepSize;

        CorrectNumberOfParticles();
        SetBodyText();
        g_FpsStatistics.clear();
    }
    break;
```

Because *g_particleNumStepSize* is a *const int* with the value of 512, there's no need to check for the number of particles not matching the tile size because you can choose only 64, 128, 256, and 512 with the slider. (The function *CorrectNumberOfParticles()* ensures that there are enough particles in the case of multiple accelerators.)

If you would like to experiment with some more tile sizes, look at the *NBodyFactory()* method in NBodyGravityAmp.cpp; it creates instances of the *NBodyAmpTiled* class with various tile sizes passed as the template parameter. It looks like this:

```
std::shared_ptr<NBodyBase> NBodyFactory(ComputeType type)
{
    switch (type)
    {
    case kSingleSimple:
        return std::make_shared<NBodyAmpSimple>(g_softeningSquared, g_dampingFactor,
            g_deltaTime, g_particleMass);
        break;
    case kSingleTile64:
        return std::make_shared<NBodyAmpTiled<64>>(g_softeningSquared, g_dampingFactor,
            g_deltaTime, g_particleMass);
```

```
        break;
    case kSingleTile128:
        return std::make_shared<NBodyAmpTiled<128>>(g_softeningSquared, g_dampingFactor,
            g_deltaTime, g_particleMass);
        break;
    case kSingleTile256:
        return std::make_shared<NBodyAmpTiled<256>>(g_softeningSquared, g_dampingFactor,
            g_deltaTime, g_particleMass);
        break;
    case kSingleTile512:
        return std::make_shared<NBodyAmpTiled<512>>(g_softeningSquared, g_dampingFactor,
            g_deltaTime, g_particleMass);
        break;
    // ... multi GPU cases elided
    default:
        assert(false);
        return nullptr;
        break;
    }
}
```

One easy tweak is to leave the drop-down box unchanged and just have the "AMP Tiled 64" choice set the tile size to 32 or the "AMP Tiled 512" choice set the tile size to 1,024. Run the app, record some new numbers, and then put the sample back to normal. Of course, you could add more options to the drop-down box and the *ComputeType* enum to permanently expand the possible types of computation.

This code defines the *ComputeType* enum:

```
enum ComputeType
{
    kSingleSimple = 0,

    kSingleTile64,
    kSingleTile128,
    kSingleTile256,
    kSingleTile512,

    kMultiTile = 5,
    kMultiTile64 = 5,
    kMultiTile128,
    kMultiTile256,
    kMultiTile512
};
```

If you add more single-tile values, be sure to increase the value of the *kMultiTile* marker because there is code that compares the compute type to *kMultiTile*.

This code in *InitApp()* sets up the drop-down box:

```
    CDXUTComboBox* pComboBox = nullptr;
    g_HUD.AddComboBox( IDC_COMPUTETYPECOMBO, -133, y += 34, 300, 26, L'G', false,
                       &pComboBox );
    std::wstring processorNames[] =
    {
```

```
        std::wstring(L"C++ AMP Simple Model "),                // kCpuSingle
        std::wstring(L"C++ AMP Tiled Model 64 "),
        std::wstring(L"C++ AMP Tiled Model 128 "),
        std::wstring(L"C++ AMP Tiled Model 256 "),
        std::wstring(L"C++ AMP Tiled Model 512 "),             // kSingleTile512
        // ... multi GPU cases elided
    };

    // ...
    std::wstring path = accelerator(accelerator::default_accelerator).device_path;

    //  If there is a GPU accelerator then use it.
    //  Otherwise add a REF accelerator and display warning.

    for (int i = kSingleSimple; i <= kSingleTile512; ++i)
        pComboBox->AddItem(processorNames[i].c_str(), nullptr
```

If you add more items to the *ComputeType* enum, you will need to add code here to add items to the drop-down box and in NBodyFactory to create an appropriate instance of *NBodyAmpTiled*.

As shipped, the sample uses the same color to draw the particles for all the single-accelerator C++ AMP compute types and one other color for all the multiaccelerator C++ AMP compute types. If you would like to use different colors for the particles for different tile sizes, you can do so by editing the code that fills the *g_particleColors* array. Even if you want to use all the same color, if you add entries to the enum, you should make the corresponding entries in this array. The relevant code is a little further down in *InitApp()* from the code that put the choices in the drop-down list, and it looks like this:

```
g_particleColors.resize(kMultiTile512 + 1);
g_particleColors[kSingleSimple] = D3DXCOLOR( 0.05f, 1.0f, 0.05f, 1.0f );
g_particleColors[kSingleTile64] = D3DXCOLOR( 0.05f, 1.0f, 0.05f, 1.0f );
g_particleColors[kSingleTile128] = D3DXCOLOR( 0.05f, 1.0f, 0.05f, 1.0f );
g_particleColors[kSingleTile256] = D3DXCOLOR( 0.05f, 1.0f, 0.05f, 1.0f );
g_particleColors[kSingleTile512] = D3DXCOLOR( 0.05f, 1.0f, 0.05f, 1.0f );
// ... multi GPU code elided
```

If you add compute types, add another line here for each of them even if you use the exact same color as for other single-accelerator tiled compute types.

On the machine where the performance numbers above were gathered, a tile size of 32 performed about the same as the simple case, and a tile size of 1,024 was about halfway between the simple case and the very similar numbers for 64, 128, and 256. The "useful range" of tile sizes is the range used in the drop-down box in the sample as shipped.

The template approach used in this sample makes it really easy to compare tile sizes. You do not have to hand-edit a constant and rebuild and run the application. The user can change the tile size at run time even though it must be a compile-time constant. A drop-down box for tile size is more likely to be a feature of a demo program than a production application, but it's not inconceivable that an application could be deployed on a variety of hardware that had different optimal tile sizes for different configurations. You could adapt this technique, including the *NBodyFactory()* method that provides the tile size as a template parameter, to enable the application to set the tile size at run

time based on a configuration setting or run-time detection of hardware. You might even write your application to run some tests with different tile sizes and choose the best one on the user's behalf. This would ensure that you got the maximum performance from the application wherever it was installed.

Summary

This sample responds dramatically well to being tiled, with a doubling of the frame rate and the GFlops. The tiling changes to the algorithm are relatively simple, with a tile's worth of data copied to *tile_static* memory and then used by all the threads in the tile. The improvement as a result of tiling is easy to see in the Concurrency Visualizer. Because the tiling algorithm has been written as a template, with tile size as a parameter, you can experiment with tile size yourself to see what the useful range of tile sizes is for a variety of particle counts on a variety of hardware. You can do this temporarily by just having one of the drop-down box choices set a different tile size, or more permanently by adding to the enum of compute types and making a few small adjustments to the rest of the code. The template approach used in this sample can be adapted to any situation where it's difficult to know what the optimal tile size will be.

CHAPTER 6

Debugging

In this chapter:

First Steps 101
GPU Debugging Basics 108
Seeing Threads 112
Taking More Control 121
Summary 125

First Steps

Debugging on the GPU feels a lot like debugging on the CPU, even though the underlying mechanics are quite different. Microsoft Visual Studio 2012 makes it simple to step through code that is inside a *parallel_for_each* or called from one. Even though your code is translated into HLSL and shipped to the GPU to execute there, Visual Studio gives you the illusion of a call stack and a familiar debugging environment.

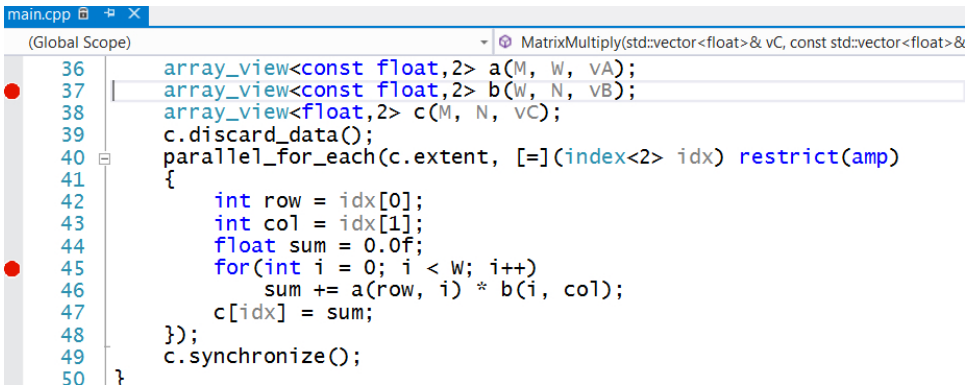
In general, you debug on an emulated accelerator called the reference accelerator. It's very slow—slower than running the code on the CPU without C++ AMP—but it's accurate and complete, and after all, when you're debugging, it's not about speed, it's about correctness. Depending on your video card, you might be able to get a driver that allows hardware debugging. The process will be the same as shown in this chapter but will involve more simultaneous threads. At the time of writing, debugging on the reference accelerator was available only on Windows 8. You can also remote debug with Visual Studio 2012 on Windows 7 and the application on Windows 8, although the setup is more complicated than it is for local debugging. This is covered in Chapter 12, "Tips, Tricks, and Best Practices."

For a given debugging run, you can debug either CPU code or GPU code; you can't mix and match. Either all your CPU breakpoints will be ignored or all your GPU breakpoints will be ignored. Once you get past that possible issue, debugging your C++ AMP code will feel just like debugging your C++ code that runs on the CPU. There are also some specific concurrency-related debugger features that are very useful in gaining an understanding of how your code works. Why not open a project and

follow along throughout this chapter? The screenshots and examples will primarily use the matrix multiplication example from Chapter 4, "Tiling."

Choosing GPU or CPU Debugging

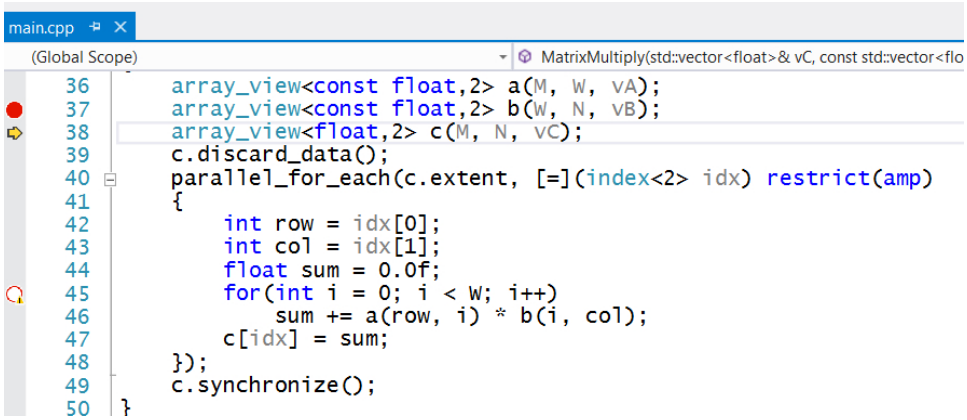
To test debugging, open a project that includes at least one *parallel_for_each* and set two or more breakpoints. Make sure you have at least one in code that will run on the GPU: inside a *parallel_for_each* or an amp-restricted function that is called from a *parallel_for_each* and one in code that will run on the CPU (neither in a *parallel_for_each* nor a function called from one). For simplicity, place them so you can see both breakpoints at once. It doesn't matter if the CPU breakpoints are C++ AMP related, such as declaring an *array* or *array_view*, as long as they are not in a *parallel_for_each* or an amp-restricted function called from one. While you are writing your code, in the IDE the two kinds of breakpoints will look just the same: a solid red dot.



There are four ways to invoke the Start Debugging command in Visual Studio:

- Press F5.
- Click the Start Debugging toolbar button.
- Right-click the project in Solution Explorer and choose Debug, Start New Instance.
- On the Debug Menu, choose Start Debugging.

Assuming you've made no project changes, all of these will launch the debugger to debug CPU code only. The CPU breakpoints will be bound (eligible to be hit if control passes through them), and the GPU ones will be unbound (skipped even if control passes through them.) The IDE will show this to you by drawing the GPU breakpoints as hollow red dots with tiny yellow warning triangles.



You can hover over these margin glyphs for more details. In this image, control has moved forward from an original breakpoint by using the Step Over command, and the three visible glyphs are the following:

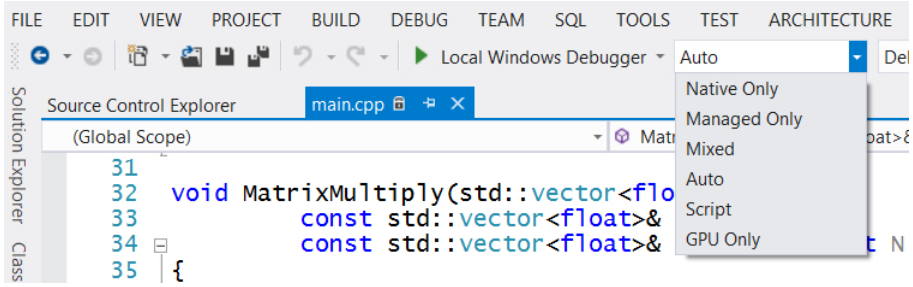
- Red dot—breakpoint that will be hit (line 37 in image)
- Yellow arrow—line that is about to execute (line 38 in image)
- Hollow red dot—breakpoint that will not be hit (line 45 in image)

The text on the breakpoint in the *parallel_for_each* is:

```
The breakpoint will not currently be hit. No executable code of the debugger's target type
is associated with this line. Possible causes include: conditional compilation, compiler
optimizations, or the target architecture of this line is not supported by the current debugger
code type.
```

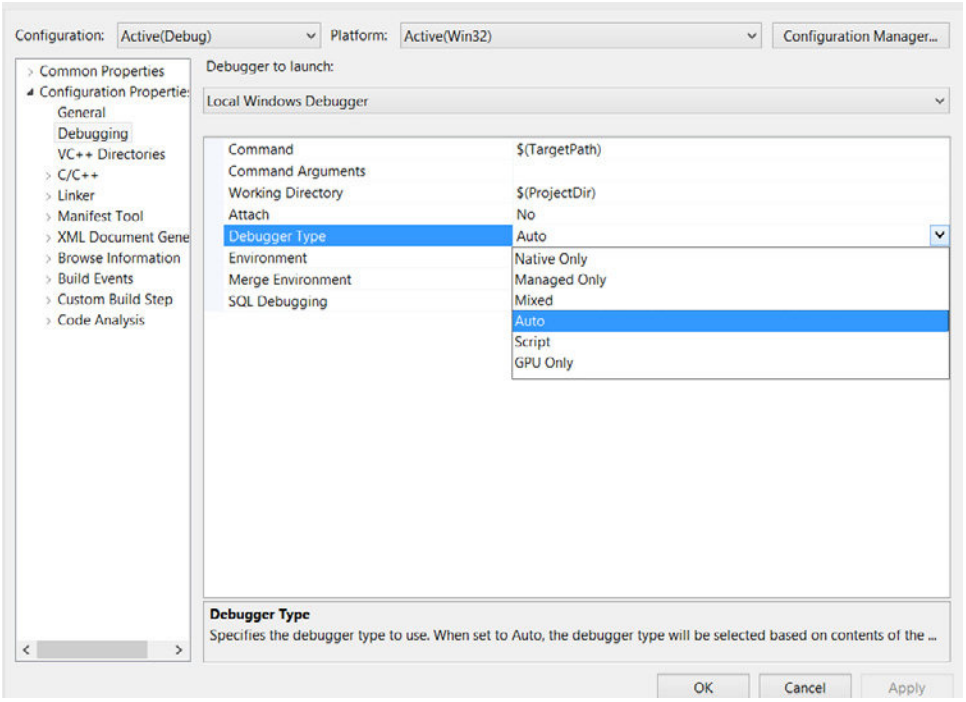
This is a very general message and not unique to C++ AMP debugging. It just describes any breakpoint that is not bound for a variety of reasons. You can confirm that it won't be hit by continuing to debug and establish that control passes over the *parallel_for_each* without pausing at the breakpoint. Then stop debugging or, if you're running a console application, run it through to completion.

There are several ways to turn on GPU debugging and enable GPU breakpoints (that is, those inside a *parallel_for_each*) to be hit. Turning on GPU debugging will disable CPU breakpoints. If you're using C++ settings, there is a drop-down box next to the Start Debugging button on the toolbar:

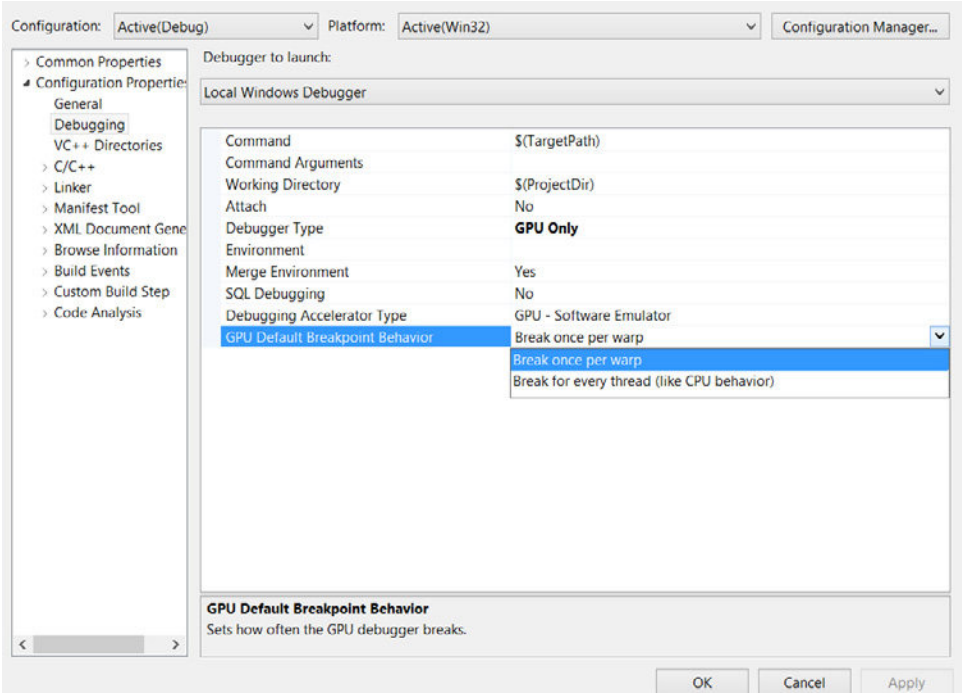


Choose GPU Only. (The Auto setting restricts you to CPU-only in this version of Visual Studio.)

If you don't have this drop-down box, you can access the setting by right-clicking the project in Solution Explorer and choosing Properties, then selecting Configuration Properties, and under that selecting Debugging. In that dialog box, the Debugger Type drop-down box has the same choices, and you can choose GPU Only:

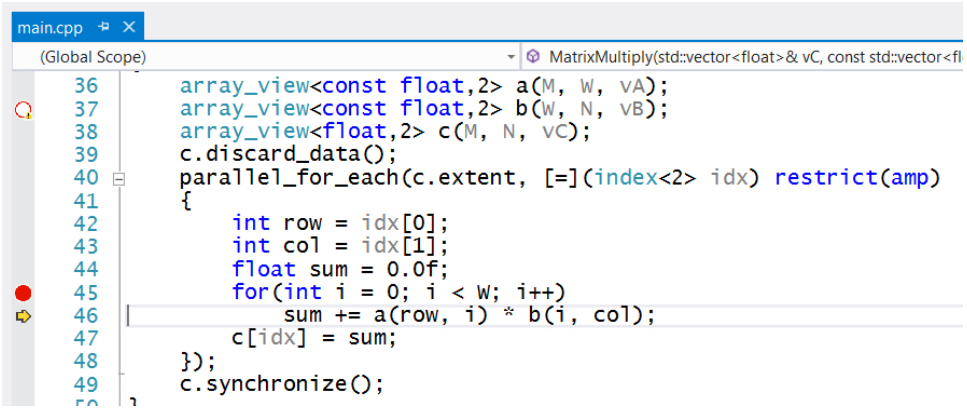


After you've chosen GPU Only in this dialog box, two more options appear: Debugging Accelerator Type and GPU Default Breakpoint Behavior. Debugging Accelerator always has the GPU—Software Emulator choice available. If you have a suitable combination of hardware and drivers to allow hardware debugging, you would see another choice in this drop-down box as well. You can also control whether breakpoints break execution once for each warp (a set of four threads when using the software emulator, or reference accelerator) or once for each thread.



On the GPU, sets of threads (warps) are executed in lock step—that is, all threads in the warp execute the same instruction at the same time on different data. When a breakpoint is placed on an instruction, all these threads "hit" the breakpoint simultaneously. Under the default setting (once per warp), the breakpoint will be reported only once—that is, the debugger will enter break state only once, even though many threads might have hit it. This setting makes stepping behave as expected; each F10 or "step over" will advance the entire warp one line. This is not what happens when debugging on the CPU, where each thread stops at each breakpoint. If you want the CPU behavior, change this setting to once per thread. You might find it produces a frustrating GPU stepping experience; each step will now require stepping over once for each thread in the warp in order to advance execution. Once-per-thread behavior might be useful when making use of conditional breakpoints. Then a breakpoint notification will be generated for each thread that satisfied the condition, even though the thread might be contained in the same warp as a previous notification. Stick with the default, once per warp, until you have a good reason to break more often.

One way or another, set your project debugging to GPU only and start debugging again. The debugger should now stop on a breakpoint in GPU code (within the *parallel_for_each* or a function marked with *restrict(amp)* that is called from a *parallel_for_each*), and the breakpoint in CPU code should be shown as unbound (hollow.)



The Reference Accelerator

When you debug a GPU application, if you don't have the drivers for hardware debugging, you'll use the Software Emulator, also known as the reference accelerator or reference rasterizer. This might cause a problem if your code explicitly chooses an accelerator to run on. Such code will generally not choose the reference accelerator, but unless you have hardware debugging available, the reference accelerator is the only accelerator that can be used while debugging. Code that does this might not be debuggable until you change it.

As introduced in Chapter 3, "C++ AMP Fundamentals," there are three ways to set the accelerator on which a *parallel_for_each* is run:

- When you construct an array, you can associate it with a particular accelerator view. If there are any array instances in the *parallel_for_each*, it will run on the *accelerator_view* associated with the array (which could be the default view on the default accelerator).
- There is an overload of *parallel_for_each* that takes an *accelerator_view*; if that overload is used, it runs on that *accelerator_view*.
- If there are only *array_views* and the *accelerator_view* wasn't passed to *parallel_for_each*, it runs on the default view of the default accelerator.

If the accelerator is set in multiple incompatible ways (for example, two array instances are passed in and each is associated with a different accelerator), an exception will be thrown.

In some applications, the default view of the default accelerator will always be used. However, you might have some code that selects a specific accelerator. For example, the NBody case study of Chapter 2, "NBody Case Study," and Chapter 5, "Case Study—Tiled NBody," includes a drop-down box to let users choose which accelerator to use when it is run on a multi-GPU system. You can also specify a specific view to change the queuing mode or to enable you to recover from timeouts, as discussed in the "Time Out Detection and Recovery" section in Chapter 12. You should be aware that specifying the accelerator to use can cause problems when debugging your application and adjust your code to use the software emulator when debugging.

To illustrate the problem, here is a seemingly pointless change you could make to the array multiplication example in use for this chapter. At the top of the *main()* function, before creating and filling the vectors of source data, add these lines:

```
accelerator defaultAcc(accelerator::default_accelerator);
accelerator_view defaultView = defaultAcc.default_view;
std::vector<accelerator> accls = accelerator::get_all();
accls.erase(std::remove_if(accls.begin(),accls.end(),
    [](const accelerator& acc) { return acc.is_emulated; } ), accls.end());
if (!accls.empty())
    defaultView = accls[0].default_view;
```

This code sets the default accelerator as a fallback value but then discovers any other accelerator on the system that is not an emulator (the reference accelerator is emulated) and does not use the CPU and chooses the first of those to explicitly use as the *accelerator_view*.

With this code in place, find one of the two *parallel_for_each* statements in the matrix multiplication example and change it, passing the *accelerator_view* as the first parameter. For example, change this line:

```
parallel_for_each(c.extent), [=](index<2> idx) restrict(amp)
```

to read like this:

```
parallel_for_each(defaultView, c.extent, [=](index<2> idx) restrict(amp)
```

Do not change the other *parallel_for_each* statement. Set a breakpoint inside each of the *parallel_for_each* calls. With GPU Only debugging selected, start debugging. Execution will stop only at the breakpoint in the *parallel_for_each* that did not specify the *accelerator_view*. To confirm what's happening, change both *parallel_for_each* statements so that the one that didn't take an *accelerator_view* now does and the one that did now doesn't. With the breakpoints still set and GPU Only debugging still selected, start debugging again. Execution will now stop at the breakpoint that was skipped before.

Knowing about the reference accelerator can solve an apparent mystery that developers sometimes face when first debugging C++ AMP code: breakpoints at which execution doesn't stop. If you set a breakpoint in a *parallel_for_each* and execution doesn't stop at it, check these three things:

- Make sure your debugging is set to GPU Only. If you're not using C++ settings, you may choose to customize your toolbars so this drop-down box is always visible to simplify checking your Debug Type.
- Make sure the code does not explicitly specify the accelerator (when constructing arrays or by passing an *accelerator_view* to a *parallel_for_each*), or if it does, make sure you specify the reference accelerator when debugging, as shown below.
- Consider lowering the workload or reducing the size of the data set you are passing to the accelerator. Because the reference accelerator is so much slower than a real accelerator, you might think the application has gotten stuck. Put your breakpoints as early in the application

as possible while you make sure debugging is properly hooked up and get a feel for how the application performs on the reference accelerator.

One technique to ensure that your code can specify an accelerator when you run a release build but still be debuggable is to omit the *accelerator_view* setting code when building a debug version. For example:

```
   accelerator defaultAcc (accelerator::default_accelerator);
   accelerator_view defaultView = defaultAcc.default_view;

#ifndef _DEBUG
    std::vector<accelerator> accls = accelerator::get_all();
    accls.erase(std::remove_if(accls.begin(),accls.end(),
        [](const accelerator& acc) { return acc.is_emulated; } ), accls.end());
    if (!accls.empty())
        defaultView = accls[0].default_view;
#endif
```

Now when a debug build is underway, *defaultView* will always be the default view on the default accelerator and any *parallel_for_each* that passes in *defaultView* will be debuggable. As always, any *parallel_for_each* that doesn't pass in an *accelerator_view* or use an *array* associated with a specific view that is not a view on the reference accelerator will be debuggable. In release builds, the code to set the *accelerator_view* will be included and the appropriate view will be used.

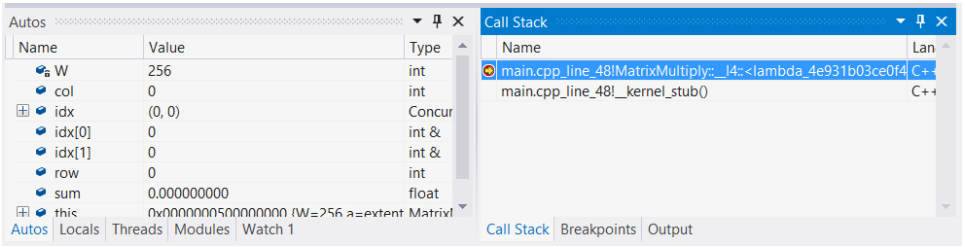
You can test this code by leaving the two *parallel_for_each* statements in the matrix multiplication sample untouched: one will take an *accelerator_view* and one will not. Add the *_DEBUG* test around the code that sets *defaultView* and start debugging. You should find that both breakpoints are hit. Tip: if you are stopped at a breakpoint in a *parallel_for_each* and click Continue, you will stop at the same breakpoint repeatedly as different threads or groups of threads reach the same statement. To go quickly to the breakpoint in the other *parallel_for_each*, clear this breakpoint after execution stops at it and then click Continue.

GPU Debugging Basics

Once you're successfully stopped at a breakpoint inside a *parallel_for_each*, take a moment and look at the information Visual Studio has to offer you.

Familiar Windows and Tips

By default, you should see an Autos window and a Call Stack window side by side below your code. Stacked with the Autos window are windows for Locals, Threads, and Modules, along with a Watch window. Stacked with the Call Stack window are the Breakpoints and Output windows. The *__kernel_stub()* referred to in the call stack is what calls your user code, your lambda.



In addition to looking at values in the Autos and Locals windows, you can hover over a variable to see a data tip that shows its value. Experiment with mouse placement a little and you'll discover you can be quite precise:

```
41      {
42          int row = idx[0];
43          int col = idx[1];
44          float sum = 0.0f;
45          for(int i = 0; i < W; i++)
46              sum += a(row, i) * b(i, col);
47          c[idx] = sum;
48      });   c[idx] 0.116226152
49      c.synchronize();
50  }
```

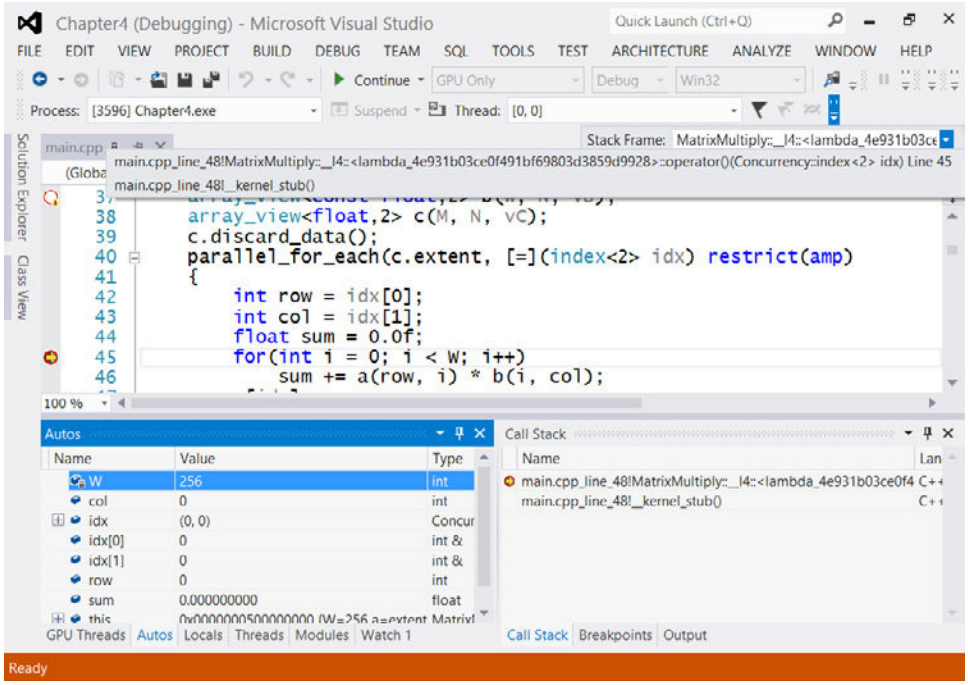
Just as when debugging on the CPU, you can pin these data tips in place and watch the values change as you step through the code. In fact, all the techniques you know from CPU debugging are available to you, including adding a watch or using Quick Watch.

You can even edit values in the Locals or Watch windows by right-clicking and choosing Edit Value, and when execution continues, the new value will be used on that thread. This is an interesting approach when you discover a logic error while you're running and realize that if a particular variable had the right value, everything would be fine. To confirm your theory, you change the value and let the run continue. (You can also use it to test your error-handling code by deliberately editing a correct value to an incorrect value and watching your code react.) None of this is unique to GPU debugging. All the techniques, tips, windows, views, and capabilities that you're used to when debugging a CPU-only application are just as applicable when debugging your accelerated code. The only difference is that you need to be aware that the same variable might have different values on different threads in the accelerator.

The Debug Location Toolbar

Some things are different when debugging concurrent code. There is an extra toolbar called the Debug Location toolbar that helps you to understand where your code has reached.

This toolbar lets you see information at a glance that would otherwise be spread over several other windows. For example, the Stack Frame shows the same information as the call stack window, and if you click the drop-down box, you'll see one entry for each line in the call stack window.



The other information available on the Debug Location toolbar will be discussed in the sections that follow.

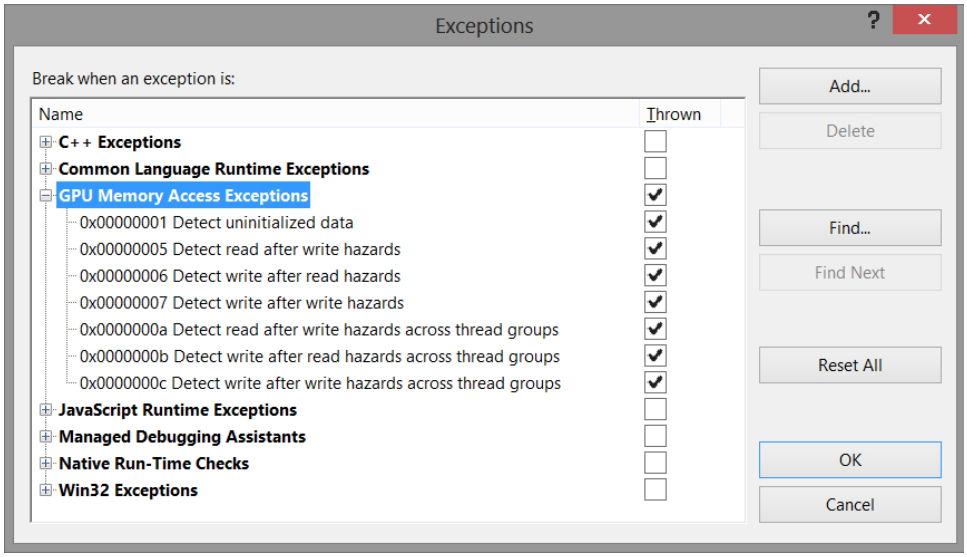
Detecting Race Conditions

One of the problems GPU Debugging can prevent is forgetting a tile barrier. If you omit a call to *tile_barrier::wait()* or another use of a barrier in a tiled *parallel_for_each*, the *tile_static* memory can be used too soon (before all the threads have finished their share of the copying) or can be overwritten too soon (before all the threads have finished using the values). The Visual Studio debugger will detect this problem. Consider this code:

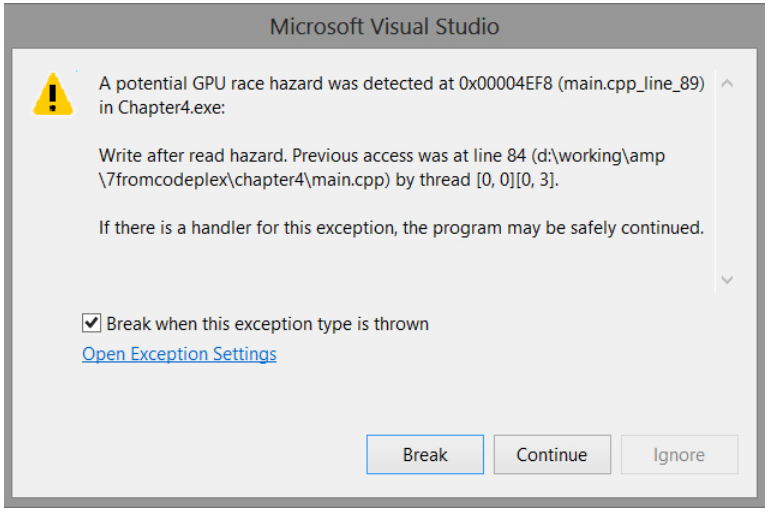
```
parallel_for_each(c.extent.tile<TileSize, TileSize>(),
    [=](tiled_index<TileSize, TileSize> tidx) restrict(amp)
{
    int row = tidx.local[0];
    int col = tidx.local[1];
    float sum = 0.0f;
    for (int i = 0; i < W; i += TileSize)
    {
        tile_static float sA[TileSize][TileSize], sB[TileSize][TileSize];
        sA[row][col] = a(tidx.global[0], col + i);
        sB[row][col] = b(row + i, tidx.global[1]);
        tidx.barrier.wait();
        for (int k = 0; k < TileSize; k++)
            sum += sA[row][k] * sB[k][col];
        //tidx.barrier.wait();
    }
```

```
    c[tidx.global] = sum;
});
c.synchronize();
```

There is a deliberate error here: the second *wait()* call has been commented out. By default, the debugger won't catch this for you, but you can ask it to. While not debugging, choose Debug, Exceptions. Expand the GPU Memory Access Exceptions node. Check the box at the end of the line for this node; this will check all the boxes for the various kinds of GPU Memory Access Exceptions.



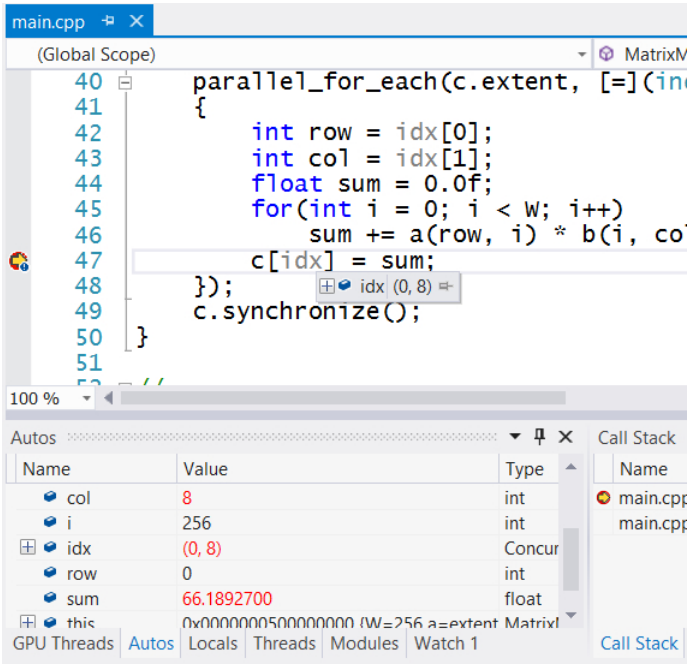
Click OK to save your settings. Now run some code that is missing a call to *wait()*, as in the code above. You don't need any breakpoints—just let it run. You will get a dialog box like this:



This sort of diagnostic can make writing correct concurrent code much easier. It is not on by default because it slows the debugging emulator (which is already very slow) and uses more memory. It's a very good idea to run your application at least once with this option on in order to gain some reassurance about race conditions. One thing to note: the race condition detector uses an exception dialog box, which is why you see the wording about safely continuing if there is a handler for this exception. A race hazard is not an exception, and you will never have a handler for it. All that you can do here is recognize that a possible race condition exists and look at the lines displayed in the dialog box to spot errors in your code (such as missing the use of a tile barrier).

Seeing Threads

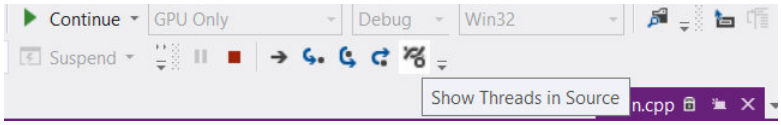
When your application is stopped at a breakpoint inside a *parallel_for_each* and you click Continue, execution often stops again at the very same breakpoint. Of course, you are now debugging a different thread. This might not be immediately apparent, but once you know where to look, Visual Studio makes that information available to you.



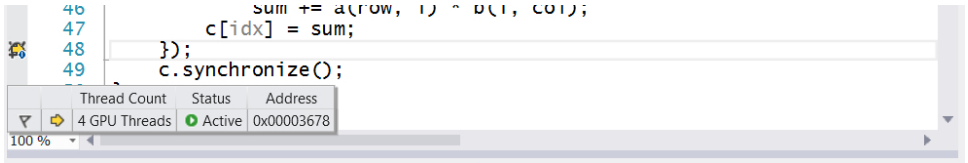
This image shows that the value of the two-dimensional index is (0, 8)—this is in the Autos window, and the data tip pops up when you hover the mouse over the *idx* variable. It's also shown to you in the Debug Location toolbar just above the code; it's labeled "Thread." This toolbar is always available when you're debugging your GPU code. It provides a convenient way to see at a glance which thread is stopped at the breakpoint.

Thread Markers

You can turn on more thread-related information as well. For example, the Debug Toolbar contains a button with the tooltip Show Threads In Source:



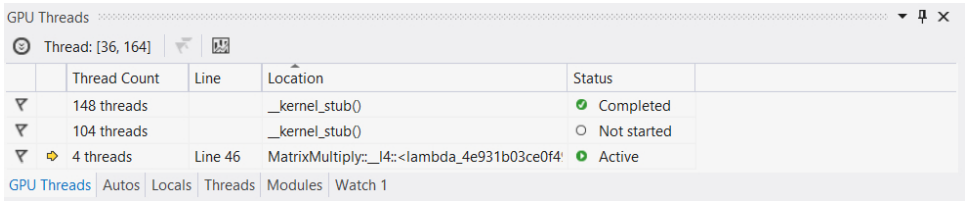
When you click this button, wavy lines appear in the gutter next to the code. They might be hard to see because they can be obscured by the yellow arrow that indicates the currently executing line, but hover over them anyway. As long as you are not at a breakpoint, you will see extra information specifically related to threads:



The pop-up window tells you how many threads are executing this particular line in your code, as well as some additional information, such as thread status. When you use the reference accelerator, which is a software emulator for debugging, four active GPU threads execute at once, running each statement simultaneously and then moving together to the next statement. If you are debugging on actual hardware, you will see a different number of threads executing simultaneously.

GPU Threads Window

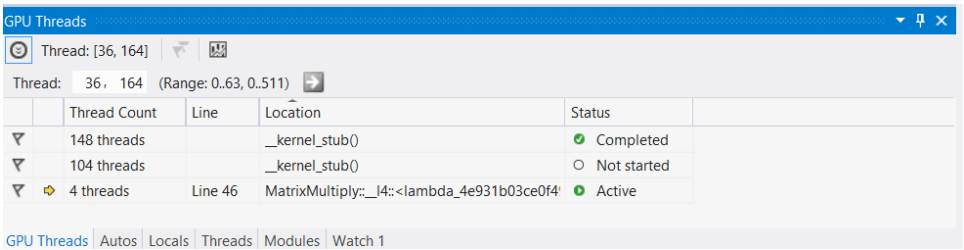
Looking at one thread at a time interactively can give you some insight into how a calculation is progressing. For a bigger picture—and to simplify switching from thread to thread—the GPU Threads Window can show you what the various threads in your application are doing. You can display the GPU Threads Window using the Debug, Windows menu, but a quicker way is to click the Thread drop-down box in the Thread Location Toolbar: click once to bring up a drop-down menu with only one choice (Open GPU Threads Window) and then move onto that item and select it.



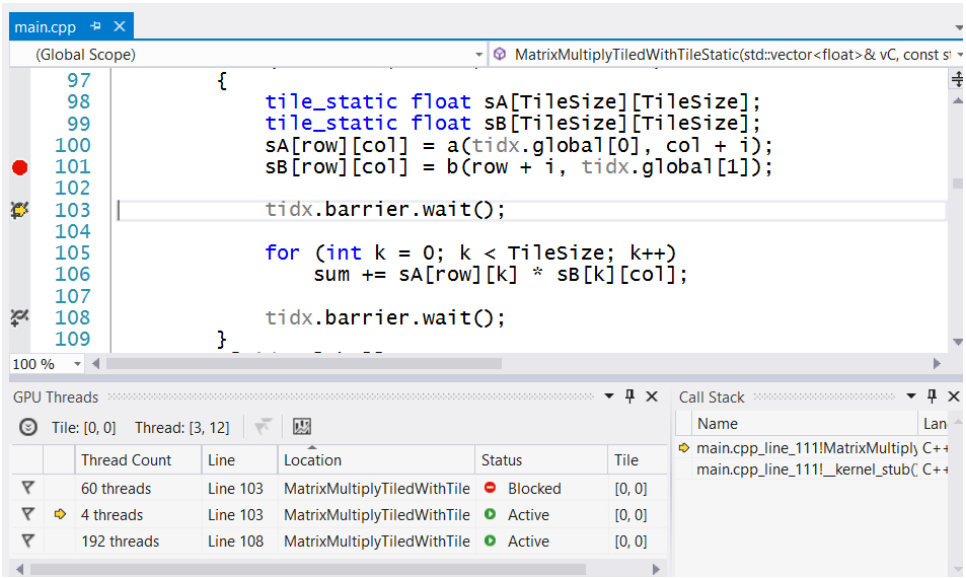
Some of the information in the GPU Threads window is the same as that provided by the tooltip on the thread marker: a number of active threads are at a particular point in the parallel_for_each where execution is paused. This window also includes information about the other threads in the computation, which have not yet started. You can use this window to switch from thread to thread, as well

as to look at information about your threads. The toolbar across the top of the window includes a control called the Thread Switcher. Click it to expand it and you can type a new thread index. (If you type an invalid index, you'll get a prompt telling you the valid range.) Click the Switch Thread arrow to switch to that thread. If the thread you choose is not active, you'll see a No Source Available window because the thread in question is not yet in the *parallel_for_each* or is no longer executing it.

As you continue to step and run through the application, you can watch the progress of rows of threads in the GPU Threads window. For example, after some time has passed, you might see a window like this:



Some threads have the status *Completed*, some are *Active*, and some are *Not Started*. When you debug a tiled algorithm, threads can be at various places within your *parallel_for_each*, such as waiting at a tile barrier. This gives you a more interesting result both in the GPU Threads window and in the thread markers in the gutter next to your code.



In this screen capture, the gutter shows a breakpoint on line 101, an unobstructed thread marker on line 108, and a thread marker with the "currently executing line" marker obscuring most of it on line 103. In the GPU Threads window, you can see one row for each thread marker. The line numbers in the window can help you map these threads to specific markers. In addition, a yellow "currently

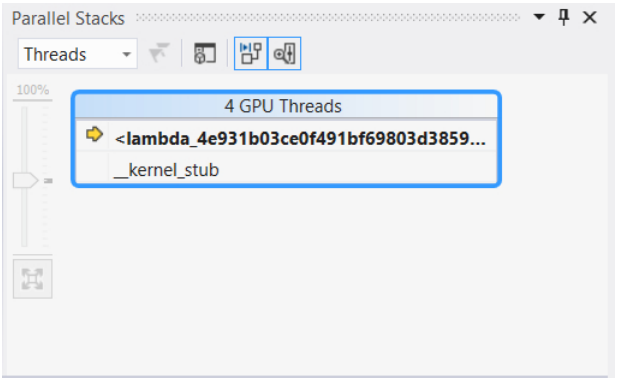
executing line" marker appears in the GPU Threads window on the row representing the threads that are at the line where that marker appears in the code. Hovering over any of the thread markers will give you some more information for each group of threads.

Try double-clicking a row in the GPU Threads window. When you do that, the yellow currently executing line marker in the code window will move to the line in the code where that row of threads is executing, and the debugger will pick one of the threads from the row and set it as the current thread. In addition, the Thread Location will update to show the thread index. (When you're debugging a tiled *parallel_for_each*, the Thread Location shows both the tile and the thread index. You will see the tile and thread index at the top of the GPU Threads window also.

The tabular presentation of the GPU Threads window provides a quick summary of what's happening with a particular *parallel_for_each*. For many applications you want more of an overview, and at the same time you want a way to get more details. The Parallel Stacks window provides both.

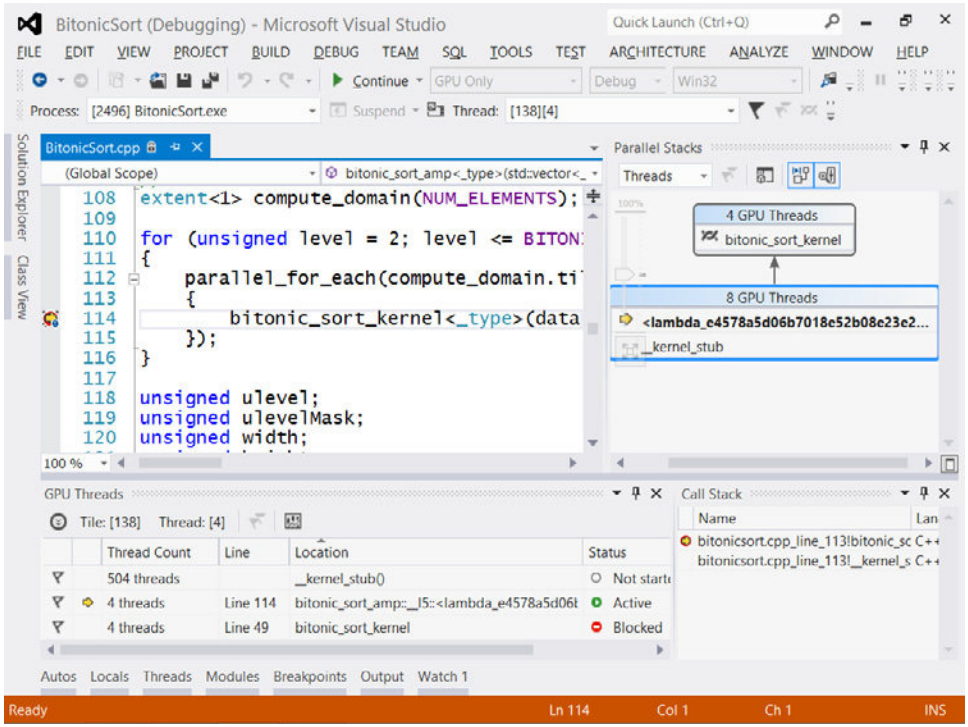
Parallel Stacks Window

If your *parallel_for_each* calls other methods, and especially if you have branching so that some threads make a particular method call and other threads do not, the Parallel Stacks window can show you what the threads are doing, filtering out any threads that are not executing. This can make it easier to see what is happening without a lot of rows referring to threads that are *Completed* or *Not Started*. You can display it using the Debug | Windows | Parallel Stacks menu. In the example used here, all the threads are in the lambda, so it doesn't provide much extra information.



If you don't see a view like this in your Parallel Stacks window, make sure the Show Stacks drop-down box at the left of the toolbar is set to Threads, not Tasks. Tasks are more appropriate in CPU parallelism with the PPL or TPL.

A good source of C++ AMP samples, suitable for general exploration as well as debugging practice, is the team blog Samples Collection at *http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx*. From that page you can download the Bitonic Sort sample. With breakpoints set in both the *parallel_for_each* and the function called from it and with execution paused appropriately, you will see the GPU Threads and Parallel Stacks windows looking like this:



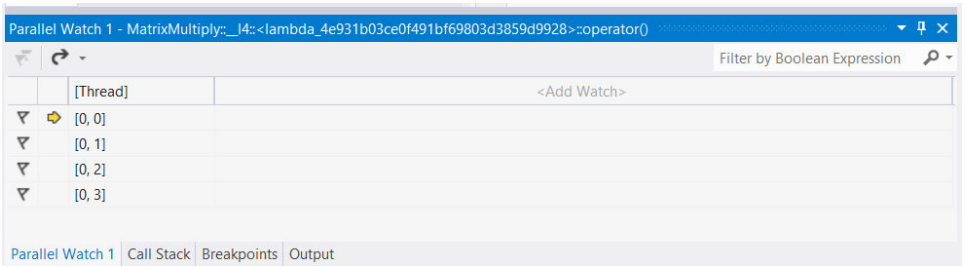
You can see that eight threads are at line 114 in BitonicSort.cpp where the "currently executing line" marker appears, and 504 other threads have not started yet, for a total of 512 threads in this tile. Of the eight threads, four are at line 49 of the *bitonic_sort_kernel()* function called from this line and four have not yet reached line 49. Some people find this information easier to see in the Parallel Stacks window than in the GPU Threads window. Others prefer to see the line numbers and other information in GPU Threads. Of course, the choice is yours.

With the various GPU debugging windows in place, you can move from thread to thread and gain more insight into the way your results are calculated. For example, when you double-click a row in the GPU Threads window, the code window shows you the line that the threads in that row are about to execute. If you hover over a variable in the code, you can see the value of that variable for one of the threads in the row. The thread index is in the Thread Location box above the code and also at the top of the GPU Threads window. You can change the thread index using the toolbar at the top of the GPU Threads window and then hover over the same variable in the code and see that it now has a new value. If you prefer, instead of double-clicking a row in the GPU Threads window, you can hover over a set of threads in the Parallel Stacks window and then double-click one of the rows that appears; this will also make the code window show you the line that those threads are about to execute.

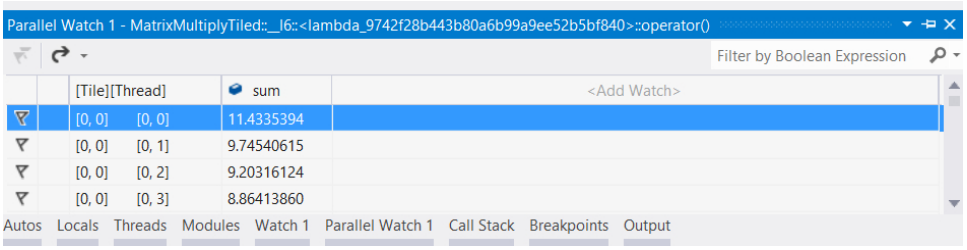
As you move from line to line by using Step Over, Step Into, or Continue, and as you switch from thread to thread, you are switching the current context and the stack frame. Switching the current context causes all other debugger windows to update.

Parallel Watch Window

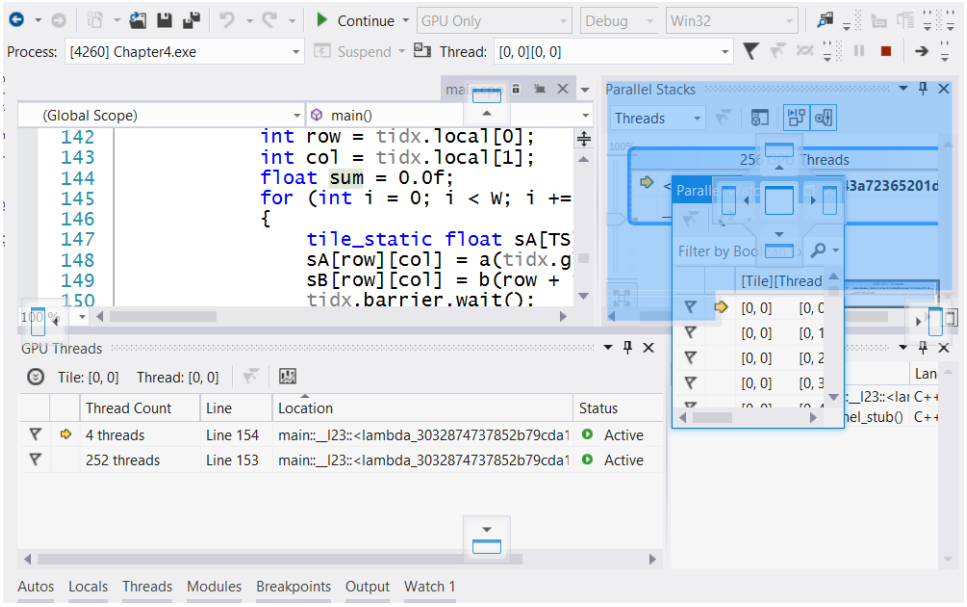
Before you get out paper and pen to note down the value of a particular variable across a number of threads, you should discover the Parallel Watch window. It lets you watch a value across multiple threads. In fact, you can have up to four Parallel Watch windows, all available from the Debug menu under Windows. When a Parallel Watch window first opens, it shows only the thread indices of the threads that are executing the current method:



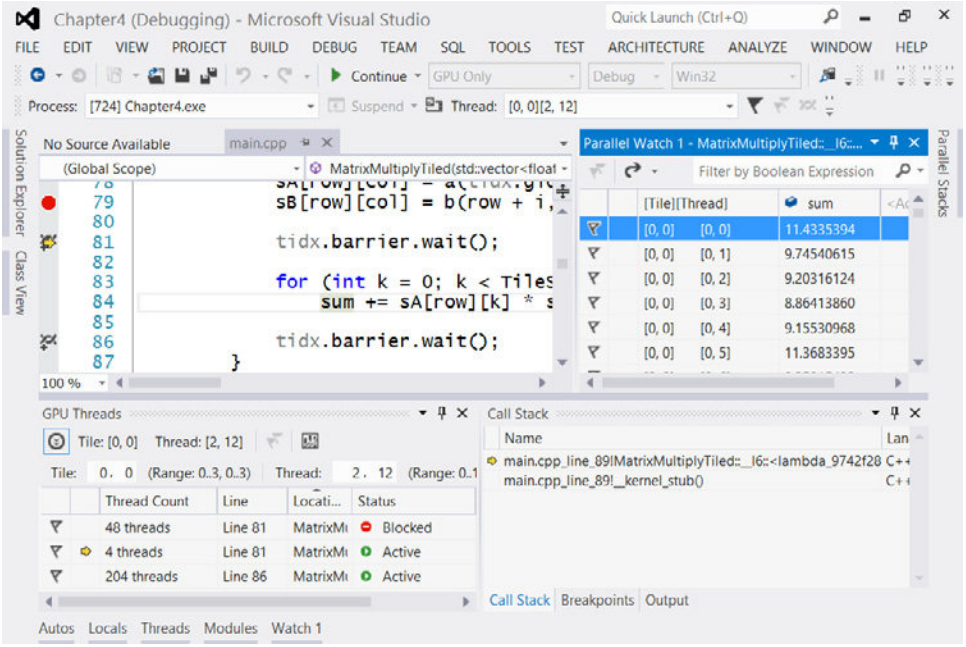
The name of the method is in the title bar of the Parallel Watch window. The names of lambdas can be hard to recognize. To see the value of a particular variable across many threads, right-click the variable in code and choose Add Parallel Watch or highlight an expression and then drag it to the window and drop it. This adds a column showing the value of that variable in each thread:



Alternatively, you can click the <Add Watch> heading in the Parallel Watch window and type or paste an expression. If you have a lot of threads, seeing just four or so in this small window might be inconvenient. If you have a large screen, you can resize the bottom zones so that more lines of information can fit in these windows. Alternatively, you can drag the window to a second monitor and enlarge it or redock the window somewhere larger, such as next to your code. Click the tab at the bottom of the window and drag it until blue sizing reminders appear and then drop it. (If you click the top of the window and drag, all the tabbed windows from this location will be dragged to the new location.)



When you have a large number of threads in flight, seeing a lot of them at once can really show how a particular value was calculated or what is wrong with it.

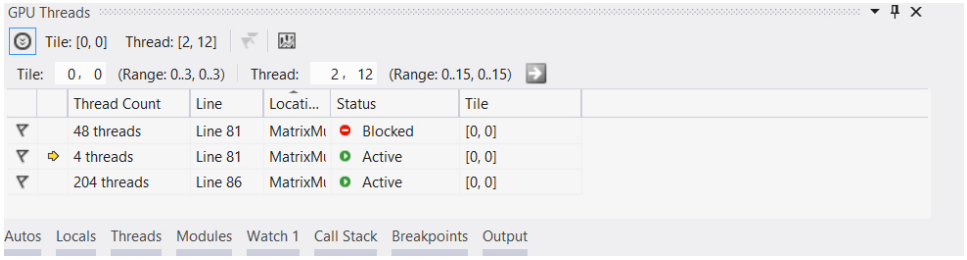


If you don't want to watch a particular value anymore, right-click in the Parallel Watch window and choose Delete Watch, or choose Clear All Watches to delete them all.

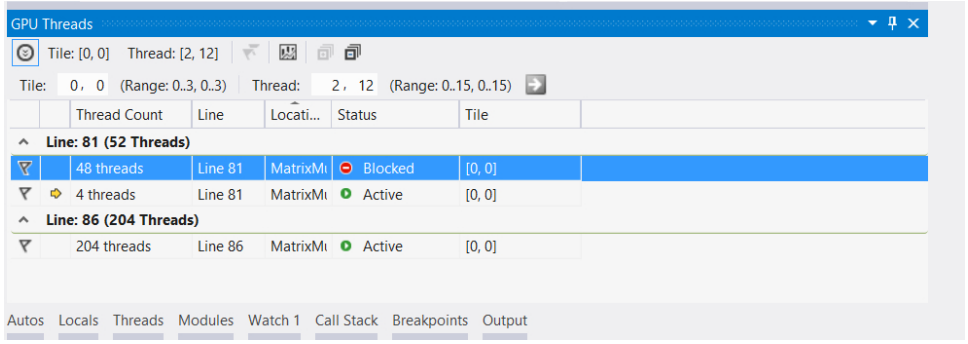
Flagging, Grouping, and Filtering Threads

Another approach would be to focus on a smaller number of threads that you know are interesting or that you are following. Throughout these windows, you will see small flag icons at the left of many rows. When you click one, you flag the thread or row of threads. You will also see an icon in most toolbars showing two small flag icons with the tooltip Show Only Flagged. Flag a few threads and then toggle this toolbar button in the GPU Threads window, the Parallel Stacks window, the Parallel Watch window, or the Debug Location toolbar. It's easy to flag and unflag entire rows of threads in the Parallel Stacks and GPU Threads windows or individual threads in the Parallel Watch window. You can also flag and unflag the current thread in the Debug Location toolbar. This capability is especially valuable in hardware debugging when many more threads at once are executing simultaneously.

Often, threads that are at the same line are represented on separate rows in the GPU Threads window because they have a different status. For example, these threads have two rows that are at line 81:

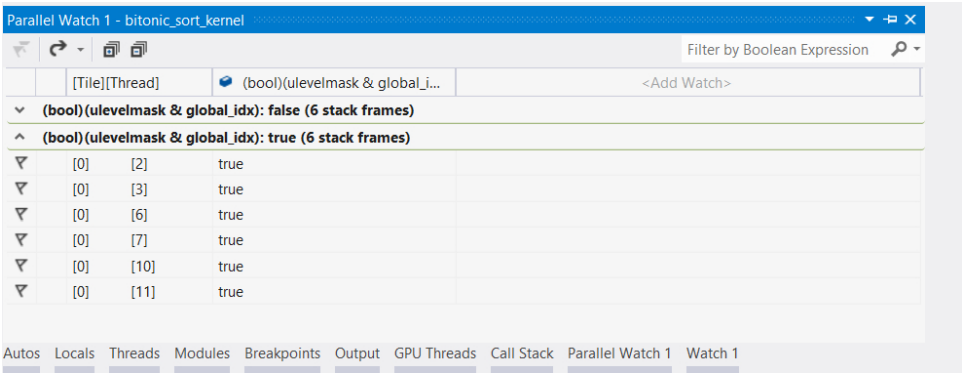


If you right-click any row in the GPU Threads window and choose Group By and then choose Line, this will be made more obvious:

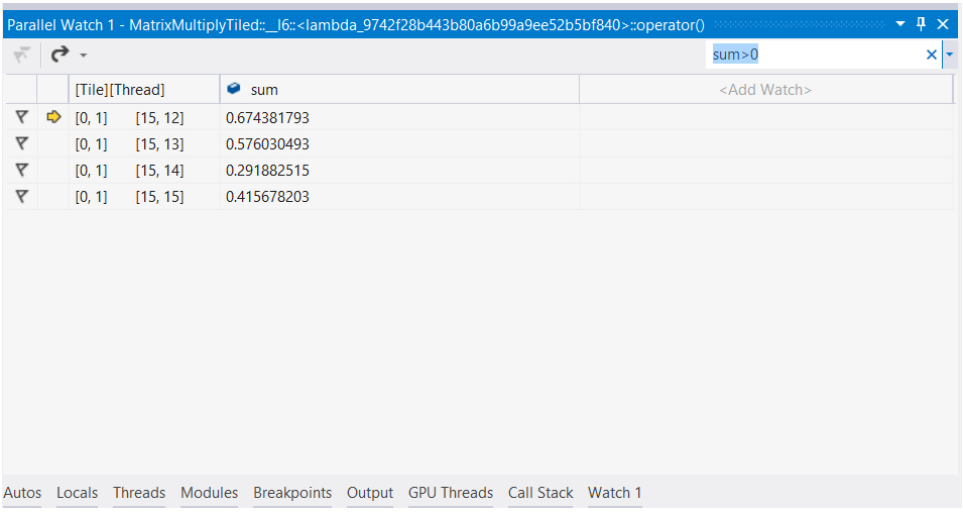


You can group rows by whether they are flagged or not, the number of threads in the row, the line being executed, the address being executed, the location (typically the function name), the status, or the tile to which they belong. Any of these options might help you to see the relationships or to focus only on the threads that matter. In the Parallel Watch window you can group watch lines by the value of a watched variable or by whether or not they are flagged.

For example, in this image taken while debugging the Bitonic sort sample, a watch has been added for the expression *(bool)(ulevelmask & global_idx),* which is used in an if-condition on line 54 of BitonicSort.cpp. The rows are grouped by the value of that expression and the first group is collapsed. You can immediately see that half the threads have the value *true* for this expression, and half have *false*:



If you aren't sure which threads are interesting but you do know which values are interesting, try filtering your Parallel Watch results. Click the text box labeled Filter By Boolean Expression. For example, if you are watching a variable called *sum* in the matrix multiplication sample, you might want to see only threads with a positive sum or only threads with a zero sum. Enter an expression such as *sum > 0* and press Enter to filter the window.



Double-click any line in your filtered results, and the code window will show you the line of code that thread is about to execute. Here, you can hover over other variables, see their values, and learn why *sum* has the value it does. You can then flag the threads being shown to you and set the other windows to show only flagged threads, thus spreading this filter to other tool windows.

Some other features are available to you in the Parallel Watch window. Click any column header to sort the table by that column; click it again to sort in the other order. The button next to Show Only Flagged in the toolbar lets you export the entire window to a CSV or, assuming Microsoft Excel is installed on your machine, to open the table in Excel for further manipulation and calculations.

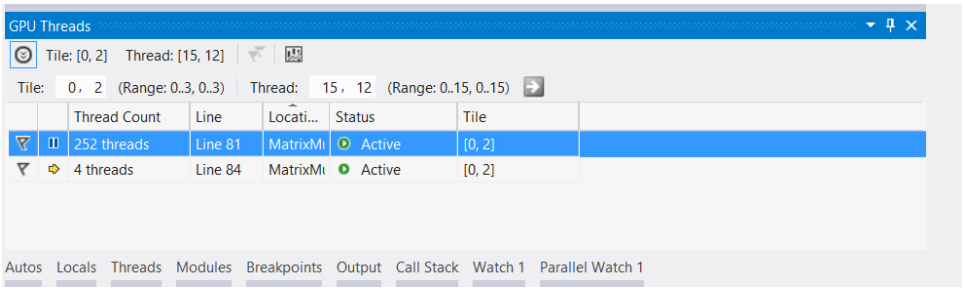
Taking More Control

Under normal circumstances, the scheduling of GPU threads is out of your control. Other than inserting tile barriers to ensure that shared resources like *tile_static* memory are not subject to data races, you have no say in the execution. You can't arrange for threads to run in a certain order, for example. Sometimes while debugging you might wish you had more power and control. Perhaps one particular set of threads is about to do something that you don't want them to do yet, for example. You might have a suspicion that you've found a bug and an idea how to fix it but not want to stop debugging, make a code change, and then repeat all the repro steps to get to this point again. Or it might be as simple as wanting to follow one path of execution all the way through a calculation without jumping to another thread every time you continue debugging. This latter circumstance is more likely to be a concern if you are hardware-debugging, with many more threads executing at once. Or perhaps you are investigating a potential data race condition and want to try to force it to occur by running suspect threads through the code in some specific order.

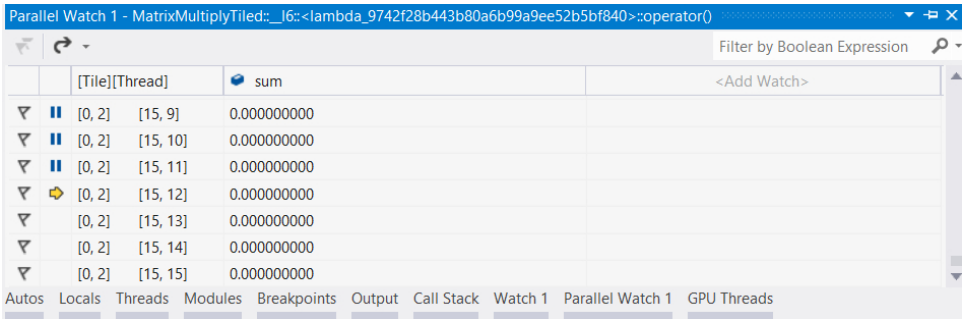
Freezing and Thawing Threads

When you find yourself in this kind of situation, you can use a capability that is also offered in parallel CPU debugging: freezing and thawing threads. It's simple enough to do and the windows all show you what is happening.

To freeze threads, right-click the appropriate row in the GPU Threads window and choose Freeze. A "pause" symbol appears in the row:

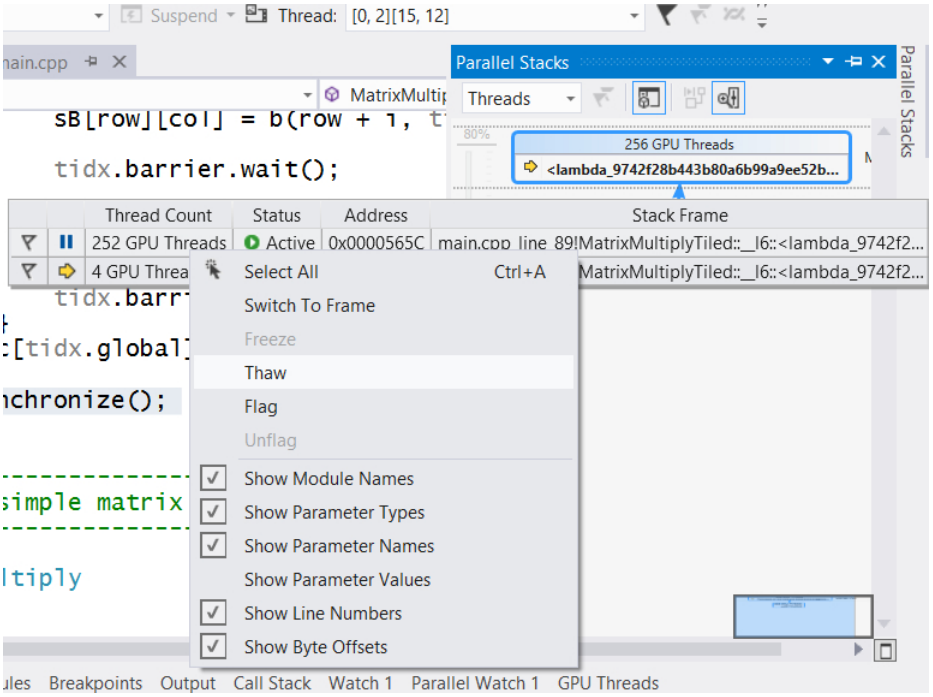


This symbol also appears in the Parallel Watch window next to frozen threads:



When you continue debugging, these threads do not move forward.

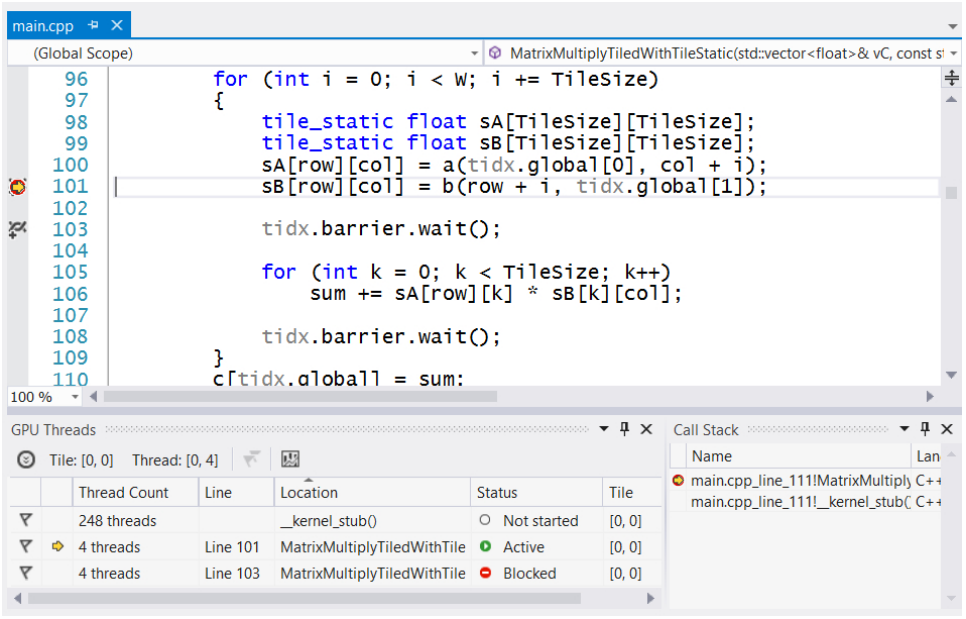
All the places that show you that a thread is frozen also offer you a way to thaw it. For example, you could hover in the Parallel Stacks window and then right-click a thread group and choose Thaw:



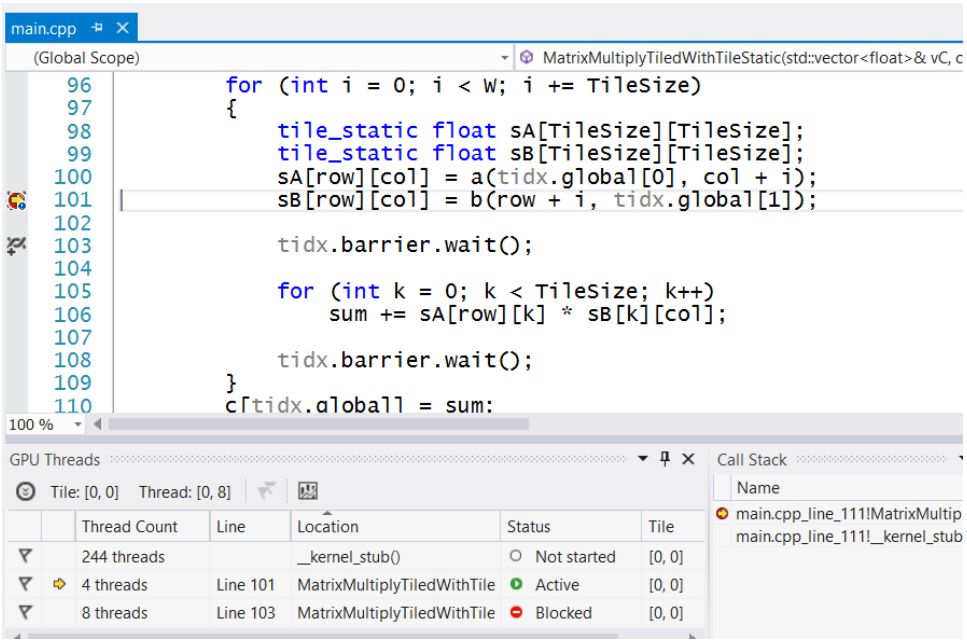
Once a row of threads is thawed, execution will continue at the next opportunity. Thawing a single thread might appear possible—for example, you could right-click in the Parallel Watch window on a single thread's row and choose Thaw—but it will thaw a warp of them, the threads that execute simultaneously while you are debugging. If you're debugging on the reference accelerator, that will be four threads. If you're hardware-debugging, it will be some other number, but it won't be one.

Run Tile to Cursor

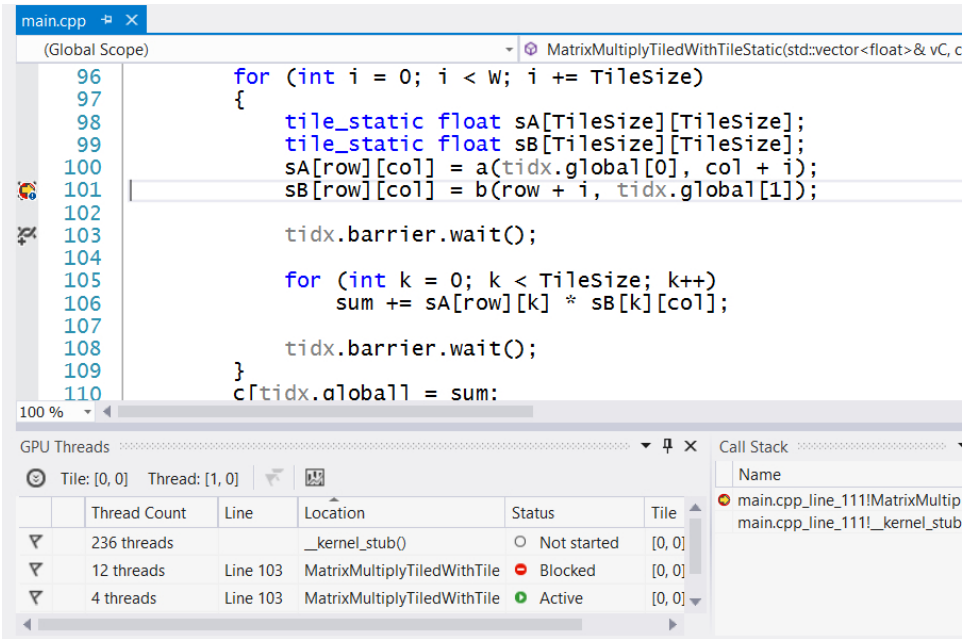
Another way you can take control is with a variant on the Run To Cursor command that is always available when debugging. The Run Current Tile To Cursor command is available when control is at a GPU statement, and it speeds up the process of stepping through until all the threads in a tile get to a specific point. For example, here is a debugging session where just four threads have reached the tile barrier on line 103:



If you click Continue, four threads (one warp on the REF accelerator) go forward to the barrier:



If you don't want to click Continue repeatedly, especially for large tile sizes, you can right-click a line on your code, such as line 103 in this screen capture, and choose Run Tile to Cursor. This will break execution when all the threads in the tile have reached the line.



Now you can see 12 blocked threads and four active threads at line 103. In this example the tile size is 16, and all 16 threads in the tile have reached the tile barrier. One warp (four threads) is active and the remainder of the threads in the tile are blocked.

Summary

Debugging parallel code, whether on the GPU or the CPU, is not like debugging single-threaded code. Visual Studio does a lot to make GPU debugging simple, but you still need to be familiar with important aspects. Just as when you debug concurrent code on the CPU, the value of any particular expression will be different from thread to thread, for example, and execution might appear to jump backward when continuing switches you to another thread that has not progressed as far through the code. Although you need to think a little differently to debug concurrent code than sequential code, you can use the familiar tools of the Visual Studio debugger just as you have done in the past.

Before you can even start to debug a C++ AMP application, you must make sure of three things:

1. If your code requests that the *parallel_for_each* loops run on a particular accelerator, it can't be debugged unless that accelerator supports debugging. If you don't have the drivers for hardware-debugging, debugging happens on the reference accelerator, a slow emulator that runs just four threads at a time. You can use conditional compilation to make sure that debug builds do not request an accelerator other than the default.
2. Select GPU debugging on the debug toolbar or in the project properties.
3. Adjust the workload, such as the number of data points you're processing, so that the slower reference accelerator will reach your breakpoints in a reasonable amount of elapsed time.

GPU debugging is richly supported in Visual Studio. The GPU Threads window, Parallel Stacks window, and Parallel Watch window all give you different information and insight into all the threads that are running through your code. Clicking in one generally affects what information is presented in the others, and by combining these windows, you can watch the threads work together to calculate your results. You can step in and change the value in a variable, change the order in which threads execute, and interact with your application in all the same ways that you can when you are debugging on the CPU. Don't think of your code on the GPU as a mystery—the debugger gives you a way to see it, control it, and understand it.

CHAPTER 7

Optimization

In this chapter:

An Approach to Performance Optimization 127
Analyzing Performance 128
Optimizing Memory Access Patterns 138
Optimizing Computation 158
Summary 169

One of the principal reasons that developers choose C++ is the level of control and performance it can provide. C++ AMP is no different. Getting the best performance out of your C++ AMP application involves both timing and profiling your current code and understanding how to get the most out of the accelerator's hardware. This chapter covers how to analyze your application's performance and offers an in-depth discussion of different aspects of writing high-performance code with C++ AMP. Chapter 8, "Performance Case Study—Reduction," also walks through a case study showing several implementations of a reduction algorithm in C++ AMP.

An Approach to Performance Optimization

Optimization is a complex topic, and some general guidelines might help you improve your application's performance.

Understand the performance criteria for your application. What levels of responsiveness, features, or user scenarios does your application require? For example, it might be acceptable for a video-editing application to take a few seconds to start playing a file when the user loads it, but any video must play at 60 frames per second. Having a clear understanding will allow you to focus on the correct areas of your application to optimize and not waste time improving the performance of features that are already acceptable to your users.

Once you understand your criteria, you should measure your application's performance and identify areas that need improvement. You can then take an iterative approach to performance improvement. Make small improvements to your code and remeasure against the scenarios you identified. You might also want to add lower-level timing code to measure specific functions within your code

and gain a better understanding of performance. Continue to iterate until you meet the performance required by the scenarios you identified.

Although it might be tempting to simply look for performance hot spots within your existing application code and focus on improving those specific areas, it's often worth considering the application's overall workflow and architecture. For example, you might want to modify the underlying data structures to make them more optimal for data-parallel processing. For a discussion of one such approach, see the section of this chapter entitled "Array of Structures vs. Structure of Arrays."

C++ AMP is designed to run on a variety of accelerator hardware. In some cases, you might be able to achieve performance gains by exploiting information that applies to specific hardware. Doing this improves performance at the expense of portability. It's up to you to decide if the loss of portability is acceptable for your application.

The remainder of this chapter describes how to analyze performance and optimize your application for memory access and computational efficiency. The sections within each optimization topic are roughly ordered according to their likely impact—although this is quite application-specific—so you should follow the approach outlined above.

Analyzing Performance

Before looking at how to improve the performance of your application, it's important to understand how to measure it correctly. Many of the topics discussed in this chapter are quite application-specific, so the best approach to applying them is through measurement and profiling of your existing code before modifying it to take advantage of the techniques described here. This section covers how to measure and profile your code.

Before you start measuring the performance of your C++ AMP application, it's important to understand that several steps make up the overall execution time. Some of these are performed only once, rather than for each kernel invocation.

- The C++ AMP run time is initialized on first use. This occurs the first time your code makes use of C++ AMP; for example, it occurs when your code creates an *array_view* instance or enumerates the available accelerators. Make sure that the run time initialization overhead is not inadvertently included in your timing.
- C++ AMP kernels are Just-In-Time (JIT) compiled from High Level Shader Language (HLSL) bytecode to machine code by the GPU driver at run time. Compiled kernels are cached until the process terminates. This means that there is an additional compilation overhead that occurs for each call site of lambda or function marked with *restrict(amp)*. Run the kernel once to force the JIT compiler to run prior to executing your timed kernel.
- On first use, arrays on both the CPU and GPU incur an additional overhead for setting up and zeroing out. You should ensure that arrays are used at least once prior to timing to make sure that they have been initialized beforehand.

In addition, you can also separate out the time taken to execute C++ AMP kernels from the time taken to copy data to and from the accelerator. The following sections cover separating out these different elements.

Measuring Kernel Performance

Understanding how to isolate the different steps that make up the overall time taken to execute a kernel will allow you to better understand your program's performance and is the first step in optimizing your code. This section explains how to do this using the Microsoft Windows high-resolution performance timer API. More details on this and other timer APIs are explained in the "About Timers" article on MSDN: *http://msdn.microsoft.com/en-us/library/windows/desktop/ms644900(v=vs.85).aspx*. The high-resolution performance timer API is sufficient for our purposes, but very accurate timing is complex. For a more in-depth discussion of implementing high-resolution timers, see "Implement a Continuously Updating, High-Resolution Time Provider for Windows" on MSDN: *http://msdn.microsoft.com/en-us/magazine/cc163996.aspx*.

Let's add timing to a very simple C++ AMP function, *DoWork*, using a high-resolution performance counter:

```
inline void DoWork(const array<const float, 1>& input, array<float, 1>& output)
{
    const float k = 1.0f;
    parallel_for_each(output.extent, [=, &input, &output](index<1> idx) restrict(amp)
    {
        output[idx] = input[idx] + k;
    });
}
```

The sample application uses *copy()* to move data to the GPU and then calls *DoWork()*. Finally, it copies the result back to the CPU using another *copy()* call. The naïve approach to measuring the overall time would be to add timing code to record the elapsed time before the first copy and after the second one.

```
std::vector<float> hostInput(20000000, 1.0f);
std::vector<float> hostOutput(hostInput.size());
array<float, 1> gpuInput(hostInput.size());
array<float, 1> gpuOutput(hostInput.size());

LARGE_INTEGER start, end;
gpuOutput.accelerator_view.wait();
QueryPerformanceCounter(&start);

copy(hostInput.cbegin(), hostInput.cend(), gpuInput);
DoWork(gpuInput, gpuOutput);
copy(gpuOutput, hostOutput.begin());

gpuOutput.accelerator_view.wait();
QueryPerformanceCounter(&end);
double elapsedTime = ElapsedTime(start, end);
std::wcout << "   Kernel time:  " << elapsedTime << " (ms)" << std::endl;
```

The *ElapsedTime()* function simply calculates the number of milliseconds between *start* and *end*.

```
double ElapsedTime(const LARGE_INTEGER& start, const LARGE_INTEGER& end)
{
    LARGE_INTEGER freq;
    QueryPerformanceFrequency(&freq);
    return (double(end.QuadPart) - double(start.QuadPart)) * 1000.0 / double(freq.QuadPart);
}
```

Note the additional calls to *accelerator_view::wait()* before timing starts and ends. These are required for accurate timing because functions and lambdas marked with *restrict(amp)* and many other C++ AMP calls cause work to be asynchronously queued. These two calls ensure that any previously queued work has completed before timing starts and that the kernel has finished executing before timing stops. Without these calls, the example above will return the time taken to queue the kernel rather than to execute it.

This example can be expanded to time the run time and array initialization time, data copy, and kernel execution times separately.

```
LARGE_INTEGER initStart, initEnd, copyStart, copyEnd, kernelStart, kernelEnd;

std::vector<float> hostInput(20000000, 1.0f);
std::vector<float> hostOutput(hostInput.size());

QueryPerformanceCounter(&initStart);
array<float, 1> gpuInput(hostInput.size());
array<float, 1> gpuOutput(hostInput.size());

gpuOutput.accelerator_view.wait();
QueryPerformanceCounter(&copyStart);
initEnd = copyStart;

copy(hostInput.cbegin(), hostInput.cend(), gpuInput);

gpuOutput.accelerator_view.wait();
QueryPerformanceCounter(&kernelStart);

DoWork(gpuInput, gpuOutput);

gpuOutput.accelerator_view.wait();
QueryPerformanceCounter(&kernelEnd);

copy(gpuOutput, hostOutput.begin());

QueryPerformanceCounter(&copyEnd);

std::wcout << " Initialize time:    " << ElapsedTime(initStart, initEnd) << " (ms)"
    << std::endl;
std::wcout << " Kernel & copy time: " << ElapsedTime(copyStart, copyEnd) << " (ms)"
    << std::endl;
std::wcout << "  Kernel time:       " << ElapsedTime(kernelStart, kernelEnd) << " (ms)"
    << std::endl;
```

In this example, the application is divided into three phases. First, the C++ AMP run time is initialized and the *gpuInput* and *gpuOutput array* instances are initialized (timed by *initStart* and *initEnd*). In addition, the kernel itself can be broken down into copying data onto the accelerator, executing code, and then copying the output data back from the accelerator.

In this particular case, the program uses *array* types and explicit copying, so isolating the time taken to copy data is trivial. If your application uses *array_view* to take advantage of implicit data synchronization, it's not possible to separate the copying of data to the GPU from the kernel execution because the data wrapped by the *array_view* is copied right before kernel execution.

The full source code for this section is in the Chapter 7 folder; open Chapter7.sln. The matrix multiply example from Chapter 4, "Tiling," also includes further examples of using performance counters to measure the time taken for C++ AMP kernels to execute. These samples all use a simple *TimeFunc()* function (in Timer.h) that can be used to time sections of C++ AMP code. For a more complete timing tool, see the implementation discussed in the MSDN article referenced earlier.

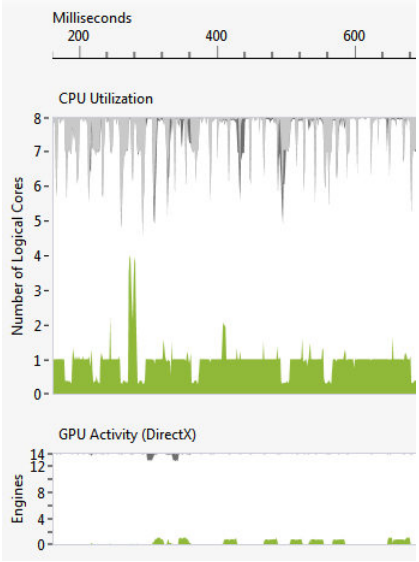
Using the Concurrency Visualizer

Although high-resolution performance counters allow you to time sections of your code, the Concurrency Visualizer allows you to further understand what is actually going on within your code. This section uses the code from one of the reducer examples in the Reduction case study in Chapter 8 to show how to use the visualizer. This isn't the most efficient implementation of the reduction algorithm, but it's a reasonable one for demonstrating how the visualizer can help you to understand your code. You can ignore the lines referring to *g_markerSeries* for now. They are explained later in the "Using the Concurrency Visualizer SDK" section of this chapter.

```
template <int TileSize>
class TiledSharedMemoryReduction : public IReduce
{
public:
    int Reduce(accelerator_view& view, const std::vector<int>& source, double& computeTime)
        const
    {
        int elementCount = source.size();
        g_markerSeries.write_flag(diagnostic::normal_importance, L"Create array");
        array<int, 1> a(elementCount, source.cbegin(), source.cend(), view);
        array<int, 1> temp(elementCount / TileSize, view);
        array_view<int, 1> av(a);
        array_view<int, 1> tmpAv(temp);
        tmpAv.discard_data();

        int result;
        computeTime = TimeFunc(view, [&]()
        {
            while (elementCount >= TileSize)
            {
                extent<1> e(elementCount);

                g_markerSeries.write_flag(diagnostic::normal_importance, L"Reduce");
```

```
                parallel_for_each(view, e.tile<TileSize>(),
                    [=] (tiled_index <TileSize> tidx) restrict(amp)
                {
                    //  Copy data onto tile static memory
                    int tid = tidx.local[0];
                    tile_static int tileData[TileSize];
                    tileData[tid] = av[tidx.global[0]];

                    //  Wait for all threads to finish copying
                    tidx.barrier.wait();

                    //  Reduce values for data on this tile
                    for (int stride = 1; stride < TileSize; stride *= 2)
                    {
                        // Highly divergent code! This will impact performance.
                        if (tid % (2 * stride) == 0)
                            tileData[tid] += tileData[tid + stride];

                        tidx.barrier.wait_with_tile_static_memory_fence();
                    }

                    //  Write the result for this tile back to global memory
                    if (tid == 0)
                        tmpAv[tidx.tile[0]] = tileData[0];
                });

                elementCount /= TileSize;
                std::swap(tmpAv, av);
                tmpAv.discard_data();
            }


            //  Copy the final results from each tile to the CPU and accumulate them there.
            std::vector<int> partialResult(elementCount);
            g_markerSeries.write_flag(diagnostic::normal_importance, L"Copy results");
            copy(av.section(0, elementCount), partialResult.begin());
            av.discard_data();
            result = std::accumulate(partialResult.cbegin(), partialResult.cend(), 0);
        });
        return result;
    }
};
```

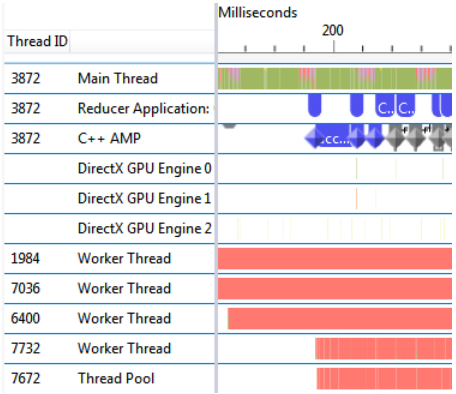
On the Analyze | Concurrency Visualizer menu, select Start With Current Project to launch the visualizer (or use the Alt+Shift+F5 shortcut). The application will launch, and, after it finishes, Microsoft Visual Studio might take a while to process the results and display the default report view.



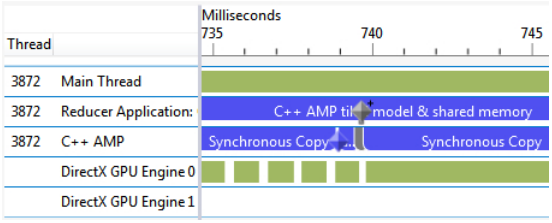
The default Utilization view shows both per-core CPU utilization (top graph) and per-engine GPU activity (bottom graph) for a section of the application run. The CPU Utilization graph shows that for the most part the Reduction application uses a single thread. The area of high CPU utilization early on the timeline corresponds to the parallel CPU implementation. The work from the different GPU reduction implementations can be seen later on the GPU Activity graph.

Switch to the Threads view, and from the Markers drop-down box select Move Markers To Top. This screen still shows time moving from left to right, but now it's divided into channels or lanes representing marker series, CPU threads, and GPU engines. One of the first channels is labeled Reducer Application and shows a time span for each reducer implementation as it executed. The reduction code uses the Concurrency Visualizer SDK to mark different time spans within the application. These are shown in the Reducer Application channel. See the next section for an explanation of how to add these.

For complex applications that make use of both available GPU(s) and multiple threads on the CPU, the Threads view gives an overview of when and where code is executing. You can use it to spot areas where the load balancing or coordination between the CPU and GPU(s) could be improved. Chapter 9, "Working with Multiple Accelerators," describes how to coordinate work between the CPU and GPU(s). The Cartoonizer case study in Chapter 10, "Cartoonizer Case Study," shows an example of this.



Because this sample doesn't make significant use of worker threads on the CPU, for clarity you might want to hide some of the unused threads in the view. Use Ctrl and the left mouse button to select one or more channels and then right-click and select Hide Selection. As the reduction implementation runs on the GPU, you can hide all the (CPU) Worker Thread and Thread Pool channels. Next, look for the time span on the Reducer Application channel named C++ AMP Tiled Model With Shared Memory. Pausing on the span marker will display a tooltip with the full name. To zoom in to this specific time, hold down the left mouse button while the cursor is just to the left of the span marker and drag the mouse to the right until the whole span marker is highlighted. Then release the left mouse button.

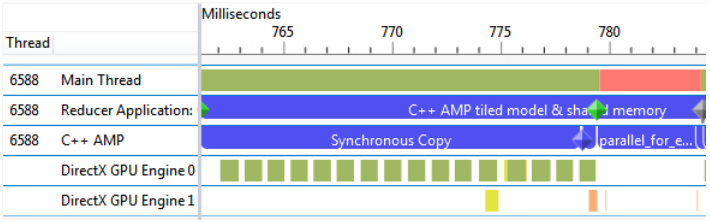


Now the visualizer window shows the following channels:

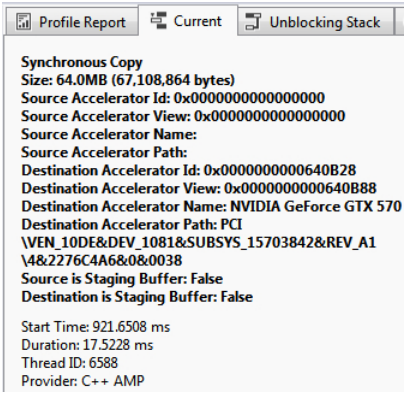
- **Reducer Application** Contains additional markers added by the application using the Concurrency Visualizer SDK.
- **C++ AMP** Contains code markers written by the C++AMP run time.
- **Main Thread** The main application thread running on the CPU.
- **DirectX GPU Engine 0** In this example, the GPU compute engine responsible for executing the C++ AMP kernel. Compute activity on the GPU from the current application on the GPU is shown in green; activity from other processes is shown in yellow.
- **DirectX GPU Engine 1** In this example, the GPU compute engine responsible for copying data to and from the CPU memory. Paging activity is shown in pink.

GPU engines are special-purpose execution hardware on the GPU. GPUs typically have several engines dedicated to different specific tasks. You should not expect your application to use all the available engines all of the time. In this case, the paging and compute engines are used. Depending on your hardware, you might see more GPU engines—for example, if your computer contains more than one GPU or if other processes are executing other work on one of the available GPUs.

The Concurrency Visualizer clearly shows the way C++ AMP uses an asynchronous model for submitting work to the GPU. Because there is an overhead for command submission, in the default queuing mode commands are batched. This means that the markers appearing in the C++ AMP channel show the time work was queued to the GPU. The GPU channels show the time and duration of the actual command-processing on a GPU device. Note how the markers on the GPU engine channels show activity occurring after the C++ AMP channel markers indicate that it was scheduled. For more discussion of queuing modes, see the section of this chapter entitled "Queuing Modes."

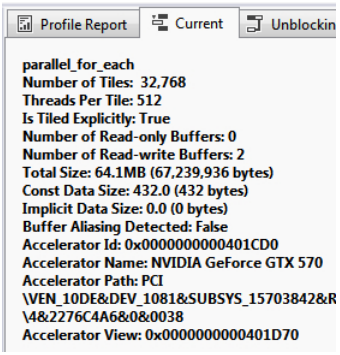


Click the Markers link in the Visible Timeline Profile and then click the Current tab. Now select the first Synchronous Copy marker on the left side of the C++ AMP channel. The Current tab shows that this is copying 64 MB of data and takes 17.5228 ms. You can also see the information shown in the current tab by holding the mouse over the marker and waiting for the tooltip to appear. Examining markers in this way is a quick way to measure the time taken for various parts of your algorithm to execute without having to add timing code to your application, as shown in the previous section.



The visualizer also shows this actual copying taking place in the DirectX GPU Engine 1 channel a short time later. This is the block displayed in the pink Paging color. Clicking different markers will show the activity associated with the marker and further details, such as its duration.

The visualizer can also give further details on the work being done by each *parallel_for_each* within your C++ AMP application. If you zoom further in, you will find markers for the two kernels that execute the reduction. Click the first one of these to see more information on the Current tab. The current tab shows detailed information for the kernel, including the number of tiles, threads per tile, the buffers associated with the kernel, whether the buffers were aliased, and on which accelerator the kernel executed.



The visualizer presents information not only for data copy operations to and from each accelerator and each kernel execution but also for synchronization events like *array_view::synchronize()*, *accelerator_view::wait()*, and *accelerator_view::flush()*.

You can zoom in further and select different markers from the different channels and see how your code executes C++ AMP kernels that in turn schedule work by means of the DMA buffer and how long each activity takes, bearing in mind the asynchronous way C++ AMP schedules work on the GPU. You can also compare how the different reduction algorithms discussed in the Reduction case study in Chapter 8 actually execute on the GPU.

The Concurrency Visualizer is a powerful tool for understanding parallel applications and has many more features not directly related to C++ AMP that are outside the scope of this book. For C++ AMP applications, it allows you to see how data is copied to and from the GPU(s) and how and when kernels are queued and executed onto the different GPU engines. You can see the time taken for each step of your application and view details of each step. It also provides a higher-level view of how operations on the GPU(s) execute relative to other work that might be running on threads on the CPU as part of a braided application (see Chapter 9, "Working with Multiple Accelerators"). It does not provide tools for analyzing memory and compute activity within each C++ AMP kernel.

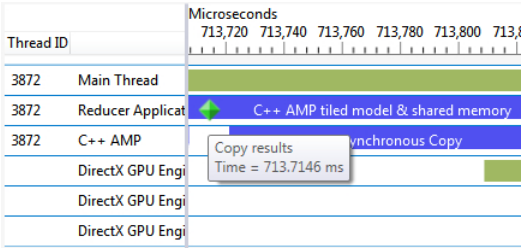
For complete documentation on the Concurrency Visualizer, see "Concurrency Visualizer" on MSDN: *http://msdn.microsoft.com/en-us/library/dd537632(v=vs.110).aspx*. You can also use the Demystify button in the top-right corner of the visualizer window to link directly to the appropriate MSDN content for each visualizer feature.

Using the Concurrency Visualizer SDK

To get a better understanding of how your application is running, you can use the Concurrency Visualizer SDK. The SDK contains functions that allow you to mark different phases within the code executing on the CPU within your application using the concurrency visualizer views. The following code shows a very simple example of using the SDK to add a new markers series and instrument it to add visualizer output.

```
#include <cvmarkersobj.h>
using namespace concurrency::diagnostic;
// ...

marker_series g_markerSeries(L"My series");
{
    span mySpan(g_markerSeries, L"My span");                    // Create a marker
    g_markerSeries.write_message(L"My message");                // Write a message

    // Code to mark with a span

    g_markerSeries.write_flag(normal_importance, L"My flag");   // Write a flag

}           // Span ends when mySpan goes out of scope
```

The markers appear in a separate channel dedicated to the *marker_series*. Move the mouse over the marker to see its name, start time, and duration. If you use too many markers, it can make the visualization harder to read, and the tool might hide some markers to improve visibility. You can use the zoom feature to increase the magnification and to show hidden spans for a specific section of the view.



This screen shot shows a marker series called "Reducer Application ..." with a flag labeled "Copy results" and a span "C++ AMP tiled model & shared memory." You can see the code that generated this output in the previous section. It is part of the Reduction case study source code in CaseStudies \Reduction\TiledSharedMemoryReduction.h.

Remember that C++ AMP executes commands on accelerators asynchronously, so just as when using high-resolution performance timers in your code, additional calls to *accelerator_view::wait()* might be required for accurate timing of different phases. Both the Reduction and Cartoonizer case studies make use of spans and messages. For more information, see "Concurrency Visualizer SDK" on MSDN: *http://msdn.microsoft.com/en-us/library/hh543789(v=vs.110).aspx*.

If you conclude that the application is memory-bound, focus on optimizing the memory accesses. This applies as much to moving data to and from CPU memory as it does to making the best use of global and tile static memory on the GPU itself within a kernel. The following sections show how to efficiently access data to get the maximum overall performance for memory-bound computations.

Before discussing this in detail, it's helpful to revisit how GPUs execute the threads that make up your kernel. GPUs consist of several processors. AMD refers to them as Compute Units whereas NVIDIA calls them Streaming Multiprocessors. Here, the term Compute Unit (CU) is used. Each CU schedules work in chunks or bundles of threads referred to as warps. When a warp is blocked, the CU scheduler can hide latencies by switching to another warp rather than waiting for the current warp. CUs are able to use this approach to hide the latencies associated with memory accesses, provided that sufficient warps are available.

Aliasing and *parallel_for_each* Invocations

Aliasing refers to instances where the same data in memory is accessed through more than one symbolic name within a program. When this happens, modifying the data by using one symbolic name alters the data when accessed via one of the other symbolic names. For a further general discussion of aliasing, see the Wikipedia entry: *http://en.wikipedia.org/wiki/Aliasing_(computing)*.

In the context of C++ AMP, a captured container is defined as an *array* or *texture* captured by reference or an *array_view* or *writeonly_texture_view* captured by value. C++ AMP considers two captured containers to be aliased in the following cases:

- The containers are captured by an amp-restricted *parallel_for_each*.
- Both captured containers refer directly to the same container on an *accelerator_view*.
- At least one of the containers is writable inside the *parallel_for_each*. If both containers are only readable, the containers are not considered to be aliased.

For each amp-restricted *parallel_for_each* invocation, the C++ AMP compiler generates CPU-side code for marshalling data and launching code execution on the accelerator. It also generates device-side code for performing the computation. In many cases, the compiler has insufficient information to determine whether aliasing is taking place.

For example, in the following code, *inputArr* and *outputArr* are separate arrays and are allocated as two buffers on the accelerator.

```
const int size = 1024;
std::vector<int> inputVec(size, 1);
array<int, 1> inputArr(size);
```

```
copy(inputVec.cbegin(), inputArr);
array<int, 1> outputArr(size);
parallel_for_each(outputArr.extent, [&inputArr, &outputArr] (index<1> idx) restrict(amp)
{
    outputArr[idx] = inputArr[idx];
});
```

If the C++ AMP kernel is wrapped in a function, it is more difficult to correctly detect when aliasing is occurring based on information known at compile time. In the following example, if *src* and *dst* refer to different *array* instances, the arrays are not aliased, just like the first example. However, if *CopyArray* is called with the *src* and *dst* parameters referring to the same array instance, this cannot be known during compilation.

```
void CopyArray(const array<int, 1> & src, array<int, 1> & dst)
{
    parallel_for_each(dst.extent, [&src, &dst] (index<1> idx) restrict(amp)
    {
        dst[idx] = src[idx];
    });
}

// ...
CopyArray(inputArr, outputArr);  // Not aliased.
CopyArray(inputArr, inputArr);   // Aliased.
```

As you can see, it is not possible to infer aliasing during compilation in the majority of cases. The current C++ AMP implementation handles aliasing at run time. The compiler generates two DirectX compute shaders for every *parallel_for_each*: one for aliased invocations and one for non-aliased invocations. The C++ AMP run time chooses the appropriate shader based on the inputs to the *parallel_for_each*. The run time detects whether the invocation is aliased or un-aliased and executes the correct shader.

This has further implications when considering *array_view* instances that reference the same *array*.

```
const int size = 100000000;
const int halfSize = size / 2;

array<int, 1> allData(size);
array_view<int, 1> firstHalf = allData.section(0, halfSize);
array_view<int, 1> secondHalf = allData.section(halfSize, halfSize);
parallel_for_each(secondHalf.extent, [=] (index<1> idx) restrict(amp)
{
    secondHalf[idx] = firstHalf[idx];
});
```

Here, the two *array_view* instances *firstHalf* and *secondHalf* represent views onto the same container, *allData*, so the *parallel_for_each* is an aliased invocation.

If the *array_view* instances are created directly on top of host memory or containers, the C++ AMP run time creates separate buffers for each of them, even if the views refer to overlapping host memory regions. In the following example, the *parallel_for_each* is not considered an aliased invocation.

```
std::vector<int> vec(size, 0);
array_view<int, 1> allData(size, vec);
array_view<int, 1> firstHalf(halfSize, vec);
parallel_for_each(firstHalf.extent, [=] (index<1> idx) restrict(amp)
{
    allData[idx + size] = firstHalf[idx];
});
```

However, if the *array_view* instances referenced by the *parallel_for_each* indirectly refer to the same host memory, aliasing will occur. In the following example, *firstHalf* is defined in terms of another *array_view* (returned by the *section()* call), so it is said to indirectly refer to host memory via another *array_view*.

```
std::vector<int> vec(size, 0);
array_view<int, 1> allData(size, vec);
array_view<int, 1> firstHalf = allData.section(0, halfSize);
parallel_for_each(firstHalf.extent, [=] (index<1> idx) restrict(amp) {
    allData[idx + size] = firstHalf[idx];
});
```

The C++ AMP run time does not detect aliasing for *array* instances created via DirectX interop. (See Chapter 11, "Graphics Interop," for more details about creating *array* instances from DirectX buffers.) The run time always uses the non-aliased shader if no other aliasing of other captured containers is detected. This can lead to run-time exceptions or an undefined result if the captured containers are aliased.

In this release of C++ AMP, aliased invocation of *texture* and *writeonly_texture_view* types is not supported. The following code results in a *runtime_ exception*.

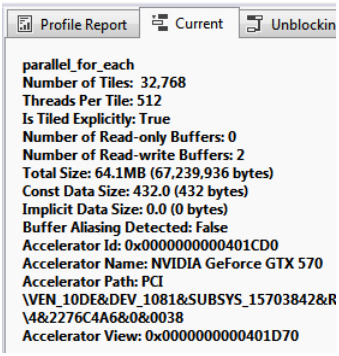
```
void CopyTexture(const texture<int, 1>& src, texture<int, 1>& dest)
{
    parallel_for_each(dest.extent, [&src, &dest] (index<1> idx) restrict(amp)
    {
        dest.set(idx, src[idx]);
    });
}

// ...
texture<int, 1> tex1(10000);
CopyTexture(tex1, tex1);
```

Note that the following is also not possible and results in a *runtime_exception* being thrown.

```
std::vector<int> input((rows * cols), 1);
texture<int, 2> text2(rows, cols, input.data(), input.size() * sizeof(int), 32u);
writeonly_texture_view<int, 2> outputTxVw(text2);
```

```
parallel_for_each(outputTxVw.extent, [outputTxVw, &text2](index<2> idx) restrict(amp)
{
   outputTxVw.set(idx, text2[idx] + 1);
});
```

The kernel fails because texture aliasing is not supported; *text2* and *outputTxVw* refer to the same texture instance and would require aliasing.



You can use the Concurrency Visualizer to find out whether the kernel invocation involved aliasing. The "Buffer Aliasing Detected" field on the Current tab shows whether the invocation was aliased. See the "Using the Concurrency Visualizer" section in this chapter for further details.

Performance Impact of Aliasing

Finally, there are performance implications for aliased and non-aliased kernel invocations. The aliased version of the shader is a lot more conservative when it comes to code generation. This can result in a reduction in performance when compared to the un-aliased shader. The performance impact will depend on the nature of the C++ AMP code and on the GPU driver and accelerator characteristics. One further point to note is that, as mentioned in the "Analyzing Performance" section, there is a JIT overhead for each call site of lambda or function. In this context, aliased and unaliased invocations count as separate call sites, each incurring their own JIT compilation overhead.

For further details of the underlying C++ AMP implementation on DirectCompute, see the product group's blog post, "Data under the covers in C++ AMP," at *http://blogs.msdn.com/b/nativeconcurrency/archive/2012/04/16/data-under-the-covers-in-c-amp.aspx*.

Efficient Data Copying to and from the GPU

Although GPUs offer huge performance gains over the CPU for data-parallel processing, moving data to and from a discrete GPU can significantly affect the performance of your application. Reducing the number of copies and the size of the data copied is one of the most efficient ways to improve application performance. The visualizer can show you the size of the data transfers to and from the GPU and the time taken for those transfers. Understand these transfer patterns and look to reduce their overhead (especially if your application makes frequent copies) before moving on to the memory access within the C++ AMP kernels.

Removing Unnecessary Copies

The *array_view* type provides a simple mechanism for automatically synchronizing data on the accelerator with the CPU as required. It also offers some fine-grained control to prevent unnecessary copies both to and from the accelerator when the data referred to by the *array_view* doesn't need to be synchronized.

If your data is required only as an input to your C++ AMP kernel, declare the element type of the *array_view* to be *const*.

```
std::vector<float> cpuData(20000000, 0.0f);
array_view<const float, 1> view(cpuData.size(), cpuData);
```

The *const* keyword indicates to the C++ AMP run time that the data associated with the *array_view* won't be changed and therefore doesn't need to be copied back from the GPU. Similarly, you can declare an *array* to be used only for input data.

```
const array<float, 1> inputData(cpuData.size());
```

In addition to reducing the number of data copies, specifying that a resource, *array*, *array_view,* or *texture* is read-only has other benefits:

- C++ AMP supports a limited number of writable resources. Explicitly specifying that a resource is read-only means that it does not get allocated from the limited pool of writable resources. DirectCompute supports 128 read-only buffers/textures but only eight writable buffers/textures for DirectX 11 devices and 64 on DirectX 11.1, which is supported on Windows 8.
- Marking a resource as read-only allows the run time to determine if the nonaliased version of the kernel can be used. See the section of this chapter entitled "Aliasing" for further discussion of aliasing and its impact on kernel execution.
- The run time can further maximize the concurrency of operations based on the read-only nature of a resource. For example, a read-only *array* on an accelerator can be concurrently copied to the CPU and used by a kernel.
- The compiler and JIT might be able to further optimize the generated code based on read-only information—for example, by caching values known to be read-only.

Similarly, for data that is only output from a kernel, you can avoid copying that data from the CPU before executing the kernel. You can use the *array_view*::*discard_data()* method to prevent data being unnecessarily copied to the GPU.

```
array_view<float, 1> outputView(cpuData.size(), cpuData);
outputView.discard_data();
```

When an *array_view* goes out of scope, its data is synchronized back to the CPU, unless *discard_data()* is called to inform the C++ AMP run time that there is no need to copy the data. It is a good practice to explicitly call *synchronize()* or *discard_data()* rather than relying on the *array_view* destructor. This will ensure that any exceptions encountered during synchronization will be propa-

gated. (C++ destructors do not propagate exceptions.) You can also use the *discard_data()* method to discard *array_view* data that is not needed on the CPU or that has been explicitly copied back to the CPU. The *SimpleArrayViewReduction::Reduce()* method in the Reduction case study uses this approach. The next chapter discusses this implementation in more detail, specifically the performance impact of the additional data copy required to initialize the *writableSource* vector.

```
int Reduce(accelerator_view& view, const std::vector<int>& source, double& computeTime)
const
{
    int elementCount = source.size();

    // Copy data, create a writable copy that can be associated with the array_view.
    std::vector<int> writableSource(source.size());
    std::copy(source.cbegin(), source.cend(), writableSource.begin());
    array_view<int, 1> av(elementCount, writableSource);
    int tailResult = (elementCount % 2) ? source[elementCount - 1] : 0;
    array_view<int, 1> tailResultView(1, &tailResult);

    std::vector<int> result(1);
    computeTime = TimeFunc(view, [&]()
    {
        for (int stride = (elementCount / 2); stride > 0; stride /= 2)
        {
            parallel_for_each(view, extent<1>(stride), [=] (index<1> idx) restrict(amp)
            {
                av[idx] += av[idx + stride];

                // If there are an odd number of elements then the first thread adds the
                // last element.
                if ((idx[0] == 0) && (stride & 0x1) && (stride != 1))
                    tailResultView[idx] += av[stride - 1];
            });
        }

        copy(av.section(0, 1), result.begin());
        av.discard_data();
    });
    tailResultView.synchronize();
    return result[0] + tailResult;
}
```

Here, *av* can't be declared *const* because the reduction kernel modifies the data in place. It also copies the result back to a vector, *result*. Because only part of a data set needs to be copied, *array_view::section()* is used to create a new *array_view* corresponding to part of the original. Sections are powerful tools for limiting the amount of data captured by a *parallel_for_each* and thereby reducing copy overhead. You should always limit the amount of data captured by a kernel to only what is needed. Sections can also be used to logically partition data, making your application code more readable and reusable. Think of *discard_data()* as a way to inform the run time that the next implicit copy of the data is not required. For readability, you should place calls to *discard_data()* close to the *parallel_for_each* with which it's associated.

Using *const*, *discard_data()*, and *section()* appropriately will improve the performance of your application by reducing the amount of data copied between the CPU and GPU. The gains will vary depending on how data-bound your particular application is. You can also use staging arrays to improve the performance of data that does have to be copied to and from the GPU. For more information on using staging arrays, see the section "Using Staging Arrays" later in this chapter.

Overlapping Asynchronous Copies

Another way to reduce the overhead of copying data to and from the GPU is to use the asynchronous nature of GPUs to overlap copy operations with other computations on either the CPU or GPU. The following example shows how to use *copy_async()* to overlap copying data to the GPU with other work:

```
std::vector<float> cpuData(20000000, 0.0f);
array<float, 1> gpuData(int(cpuData.size()));

completion_future f = copy_async(cpuData.begin(), cpuData.end(), gpuData);

// Do other work on CPU or GPU that does not modify cpuData or access gpuData

f.get();
parallel_for_each(gpuData.extent, [=, &gpuData](index<1> idx) restrict(amp)
{
    gpuData[idx] = ...
});
```

Here, the copy of data from *cpuData* to *gpuData* starts asynchronously while other work can execute on the either the CPU or GPU. To prevent data races, the source data can't be modified until the copy has completed. The kernel defined by the *parallel_for_each* that uses the *gpuData* will execute only once the copy has completed because *f.get()* blocks until the copy is complete before allowing the program to execute the *parallel_for_each*. This approach doesn't reduce the amount of data being copied, but it does allow your application to do other useful work while the copy completes.

Leaving Data on the GPU

Another way to improve performance is to reduce unnecessary copies of data between kernel executions. For example, if several kernels use the same data without requiring it to be updated by the host, there is no need to copy the data to the accelerator again. You can use this strategy to amortize the data transfer overhead across multiple kernels. The *DivergentKernelExample()* function in Chapter7\main.cpp uses this approach. The stencil functions each call a kernel that takes an *array*, *gpuInput*, and return *gpuOutput*.

```
array<float, 2> gpuInput(4000, 4000);
array<float, 2> gpuOutput(gpuInput.extent);
std::vector<float> hostInput(gpuInput.extent.size(), 1.0f);
std::vector<float> hostOutput(gpuOutput.extent.size(), 0.0f);
```

```
copy(hostInput.begin(), gpuInput);

ApplyDivergentStencil(gpuInput, gpuOutput);
copy(gpuOutput, hostOutput.begin());

ApplyImprovedStencil(gpuInput, gpuOutput);
copy(gpuOutput, hostOutput.begin());
```

Some of the code related to kernel timing and output has been removed for clarity. The input array is copied only once because each kernel can reuse the input data. This improves the overall performance by entirely eliminating copies of data that do not change between kernels. Here, the example uses *array* and explicit copies for clarity. Using *array_view* in place of *array* will allow the run time to perform this optimization implicitly.

If your application displays its results using the GPU, you can modify your rendering code to read result data directly from an *array* on the GPU. This removes the overhead of copying it back to the CPU entirely. Chapter 11, "Graphics Interop," covers this approach in more detail.

Using Staging Arrays

This section covers using C++ AMP staging arrays to improve data-copying performance. Consider the following code that creates an array instance, *gpuData,* and copies data from a vector, *cpuData*, into it and so onto the GPU:

```
std::vector<float> cpuData(size);
accelerator gpuAccel(accelerator::default_accelerator);
array<float, 1> gpuData(size, gpuAccel.default_view);
copy(cpuData.begin(), cpuData.end(), gpuData);
```

The array constructor has an overload that both creates and copies the source data so the constructor and the copy can be combined in a single statement.

```
array<float, 1> gpuData(size, cpuData.begin(), cpuData.end(), gpuAccel.default_view);
```

In both cases, this code creates a DirectX 11 staging buffer on the CPU. A staging buffer is an area of system memory that meets the memory alignment requirement for Direct Memory Access (DMA) to and from the GPU. Staging buffers are also pinned so that the GPU can access them.

The staging buffer is mapped to obtain a CPU pointer and prevent the GPU from modifying the memory. Next, data is copied from *cpuData* to the staging buffer and the buffer is unmapped, allowing the GPU to access the memory. Finally, the data is copied from the staging buffer to the GPU buffer, *gpuData*, and the staging buffer is released. The same steps occur in reverse when data is copied from the GPU back to CPU memory.

The staging buffer allocation, mapping and unmapping, as well as the two copies all contribute to the overhead of moving data to the GPU. In cases where efficient data transfer to and from the GPU is critical to your application's performance, C++ AMP provides staging arrays to give you more finely grained control over data movement.

```
accelerator_view cpuAccView = accelerator(accelerator::cpu_accelerator).default_view;
array<float, 1> stagingData(size, cpuAccView, gpuAccel.default_view);
array_view<float, 1> viewData(stagingData);
```

Staging arrays are constructed like ordinary C++ AMP arrays but take an additional *accelerator_view* parameter. In the example code, the staging buffer is located on the CPU accelerator and configured for best performance when copying data to and from *gpuAccel*.

Consider replacing a container on the CPU with a staging array to remove the additional overhead of copying from the container to a staging array prior to copying to the accelerator. In the context of the example here, your application would use the *stagingData* directly, rather than a *std::vector<float>* for storing data on the CPU.

Although staging arrays are a powerful C++ AMP feature, some care should be taken when using them:

- Do not access the data within a staging array while a copy or *parallel_for_each* that accesses the staging array is running. For example, one CPU thread is copying data to or from a staging array and another thread attempts to modify the staging array's data. In this regard, staging arrays are no different from other data structures that do not have thread-safe guarantees.
- Pointers to a staging array, either returned by the *array::data()* method or from the [] index operator, should not be cached. They are not guaranteed to be valid once a copy operation or *parallel_for_each* operating on the staging array starts.
- C++ AMP staging buffers are backed by a DirectX 11 staging buffer that is allocated from system memory. Because this is a limited resource, be careful when using staging buffers. Reuse staging arrays where possible and release any staging buffers that your application is not using.

Staging buffers also add to the complexity of your code. You should use them only in cases where data transfer time significantly affects your application's performance.

Efficient Accelerator Global Memory Access

Because accessing global memory is costly, your C++ AMP code should try to reduce the overall number of reads and writes to global memory wherever possible. This might involve changing your algorithm or just reducing accesses by caching data in tile static memory or registers for the duration of their use. Even when the overall number of accesses has been minimized, the pattern of accesses to global memory is also important.

GPU hardware optimizes memory access performance by transferring data in blocks of several consecutive memory locations. Here, these blocks are referred to as transfer lines and are analogous to a cache line in CPU cache memory. The key to maximizing memory access performance is to arrange your data in such a way that much of the data in each transfer line is used. When all the data in each transfer line is used, the memory accesses are said to be coalesced.

For example, if each thread within a warp reads unit stride (consecutive) *float* values from an *array<float, 1>,* then all of the data within each transfer line is used. Remember that both *array* and *array_view* guarantee that elements in the least significant dimension are stored in consecutive memory locations. In contrast, if the threads read from memory with a nonunit stride, then memory bandwidth is wasted. In the worst case, large strides will cause only one float value to be used from each line. Coalesced memory accesses can have huge performance benefits. Conversely, uncoalesced accesses can lead to large drops in performance.

Even relatively simple algorithms like a matrix transpose can benefit from your understanding how to efficiently access global memory and maximize coalescing. A matrix transpose simply swaps the contents of a matrix along the leading diagonal. The code for transpose looks remarkably similar to a copy. With a copy, data is moved directly from the input array to the output using the same index.

```
static const int tileSize = 32;
const int matrixSize = tileSize * 200;
array<float, 2> inData(matrixSize, matrixSize);
array<float, 2> outData(matrixSize, matrixSize);

parallel_for_each(inData.extent, [=, &inData, &outData](index<2> idx) restrict(amp)
{
    outData (idx[0], idx[1]) = inData(idx[0], idx[1]);
});
```

The example uses *inData(idx[0], idx[1])* for clarity when comparing to the transpose implementation; this is equivalent to *inData[idx]*. In the copy example, both the reads and writes are coalesced; threads within a warp both read and write to contiguous memory addresses within global memory. The diagram below shows this clearly. Here, the memory layout of the input and output arrays is shown, and each element is labeled with the ID of the thread responsible for accessing it. Consecutive threads both read and write consecutive memory locations. For example, thread 2 reads and writes location (0, 1).

inData

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

outData

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Transpose also executes a copy, but with the indices of the *outData* array reversed.

```
parallel_for_each(inData.extent, [=, &inData, &outData](index<2> idx) restrict(amp)
{
    outData(idx[1], idx[0]) = inData(idx[0], idx[1]);
});
```

You might expect that these two kernels would have approximately the same execution time because they copy exactly the same amount of data. In fact, the transpose kernel is much slower.

Time taken to apply operation to 6400 × 6400 matrix of float

Matrix copy	3.0 ms
Matrix transpose	29.6 ms

You can run this sample for yourself. Open Chapter7.sln in the Chapter 7 folder. The function *MemoryAccessExample()* in main.cpp contains the code. The exact times you observe might differ.

This difference in execution time is due to the writes to *outData* being uncoalesced. Although the reads from *inData* on each thread are from adjacent memory addresses, the writes to *outData* from consecutive threads occur on different rows.

inData

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

outData

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

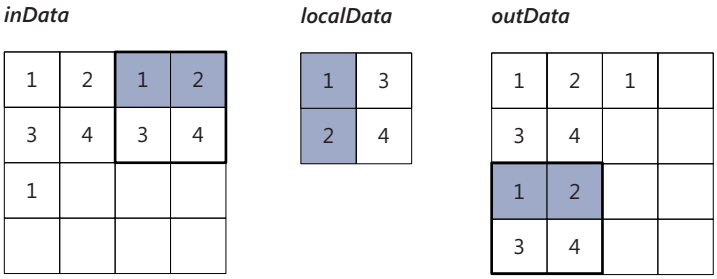
This explains the big difference in performance. The threads are writing to memory locations that are not adjacent.

It's actually possible to use tile static memory to mitigate this by adding an additional set of copies, as shown in the following example:

```
parallel_for_each(inData.extent.tile<tileSize, tileSize>(),
    [=, &inData, &outData](tiled_index<tileSize, tileSize> tidx) restrict(amp)
{
     tile_static float localData[tileSize][tileSize];
     localData[tidx.local[1]][tidx.local[0]] = inData[tidx.global];

     tidx.barrier.wait();

     index<2> outIdx(index<2>(tidx.tile_origin[1], tidx.tile_origin[0]) + tidx.local);
     outData[outIdx] = localData[tidx.local[0]][tidx.local[1]];
});
```

Here, the kernel is divided into two phases using the familiar tiled kernel pattern introduced in Chapter 4. The first part of the kernel copies coalesced data from *inData* in global memory into tile

static *localData* and transposes during the copy to tile static memory. After the barrier, which ensures that all threads have finished the copy, the data is written in a coalesced way back to global memory. Tile static memory has a much higher bandwidth and smaller interface width than global memory, so the penalty for uncoalesced memory accesses is far less. By transferring the matrix elements by means of tile static memory and doing the transpose there, uncoalesced writes to global memory can be eliminated, as shown in the diagram.



Although this might seem counterintuitive, it shows how important coalesced memory access is. The transpose kernel implemented with tile static memory is faster because it uses coalesced read from global memory to tiled memory followed by a coalesced write from tile memory to global memory.

Time taken to apply operation to 6400 × 6400 matrix of float

Matrix copy	3.0 ms
Matrix transpose	29.6 ms
Matrix transpose with coalesced global memory access	9.1 ms

This takes less time than a single coalesced read and an uncoalesced write using only global memory. For best performance, make sure that reads and writes to global memory are coalesced.

Array of Structures vs. Structure of Arrays

GPU hardware is designed to provide the best performance when all threads within a warp are accessing consecutive memory and performing the same operations on that data. Consequently, it should come as no surprise that GPU memory is designed to be most efficient when accessed in this way. In fact, load and store operations to the same transfer line by different threads in a warp are coalesced into as little as a single transaction. The size of a transfer line is hardware-dependent, but in general, your code does not have to account for this if you focus on making memory accesses as contiguous as possible. The Visual Studio Concurrency Visualizer does not provide data for either global or tile static memory access within a kernel, but it's possible to examine your code and look for inefficient memory access patterns.

The following sections on efficient memory access illustrate why many C++ AMP applications make use of a structure containing arrays rather than the more conventional array of structures.

As we saw in the NBody case study described in Chapter 2, "NBody Case Study," the CPU implementation stored the data for each particle as a structure containing the position and velocity, as well as other information.

```
struct ParticleCpu
{
    float_3 pos;
    float_3 vel;
    // ...
};
```

This is a fairly common approach for applications written primarily to run on a CPU, especially those that do not make use of the SIMD extensions. The disadvantage of using an array of structures on a GPU becomes clear when you consider the memory access patterns generated by a simple C++ AMP kernel that accesses an array of *ParticleCpu*.

```
array<ParticleCpu, 1> particles(100);
// Initialize particles...

parallel_for_each(particles.extent, [&particles](index<1> idx) restrict(amp)
{
    float dx = particles[idx].pos.x;
    float dy = particles[idx].pos.y;
    float dz = particles[idx].pos.z;

    // Calculate value based on dx, dy & dz ...
});
```

The global memory accesses are uncoalesced, as shown in the table below, which lists the byte offsets for each load operation across some exemplar threads. Thread accesses to *particles[idx].pos.x* are 24 bytes apart. The threads in a warp will access memory that is not contiguous and spans many memory segments.

Thread	0	1	2	…15
dx = particles[idx].pos.x	*0*	*24*	*48*	*360*
dy = particles[idx].pos.y	4	28	52	364
dz = particles[idx].pos.z	8	32	56	368

This will lead to poor performance when accessing data in global memory. Similarly, if the data is in tile static memory, it will be harder to arrange accesses to avoid bank conflicts. The data structure might need additional padding to avoid bank conflicts. Bank conflicts are covered in the next section.

In the C++ AMP NBody implementation, the body data is stored in a single structure containing an *array<float, 1>* array for position and velocity.

```
struct ParticlesAmp
{
    array<float_3, 1>& pos;
    array<float_3, 1>& vel;
    // ...
};
```

In this case, the same kernel can access positions and velocities separately, which leads to more coalesced memory access patterns.

```
ParticlesAmp particles;
// Initialize particles...

parallel_for_each(particles.pos.extent, [&particles](index<1> idx) restrict(amp)
{
    float dx = particles.pos[idx].x;
    float dy = particles.pos[idx].y;
    float dz = particles.pos[idx].z;

    // Calculate value based on dx, dy & dz ...
});
```

The memory accesses are partially coalesced because this is still an array of structs, albeit smaller ones of *float_3*. The accesses to *particles.pos[idx].x* are now 12 bytes apart, meaning that more accesses within a warp will be within the same memory segment.

Thread	**0**	**1**	**2**	**...15**
dx = particles.pos[idx].x	*0*	*12*	*24*	*180*
dy = particles.pos[idx].y	4	16	28	184
dz = particles.pos[idx].z	8	20	32	188

The NBody case study uses this structure to make passing position data to the DirectX vertex buffer simpler and more efficient. You can achieve further efficiency by reorganizing the position and velocity data structures into three arrays of *float* to allow completely coalesced global memory accesses.

```
struct ParticlesAmp
{
    array<float, 1>& posx;
    array<float, 1>& posy;
    array<float, 1>& posz;
    // ...
    array<int, 1> a(elementCount, source.cbegin(), source.cend());
};
```

The accesses to *posx[idx]* on each thread are now arranged across contiguous sections of memory, each thread loading a *float* from addresses four bytes apart.

Thread	**0**	**1**	**2**	**...15**
dx = particles.posx[idx]	0	4	8	60
dy = particles.posy[idx]	64	68	72	124
dz = particles.posz[idx]	128	132	136	188

Changing the memory layout of the core data structures in your application is often not trivial, so this is something that you should consider in the early stages of design. In general, you should try to

arrange for memory accesses within a warp to be as contiguous as possible. The Reduction case study demonstrates the benefits of optimizing for coalesced global memory access, as well as removing tile static memory bank conflicts.

Efficient Tile Static Memory Access

Chapter 4 already covered why tiling is important to make the best use of faster tile static memory to minimize the number of global memory accesses. Tile static memory is divided into a number of modules referred to as banks. Tile static memory typically consists of 16, 32, or 64 banks, each of which is 32 bits wide. This is specific to the particular GPU hardware and might change in the future. Tile static memory is interleaved across these banks. This means that for a GPU with tile static memory implemented with 32 banks if *arr* is an *array<float, 1>,* then *arr[1]* and *arr[33]* are in the same bank because each *float* occupies a single 32-bit bank location. This is the key point to understand when it comes to dealing with bank conflicts.

Each bank can service one address per cycle. For best performance, threads in a warp should either access data in different banks or all read the same data in a single bank, a pattern typically optimized by the hardware. When these access patterns are followed, your application can maximize the available tile static memory bandwidth. In the worst case, multiple threads in the same warp access data from the same bank. This causes these accesses to be serialized, which might result in a significant degradation in performance.

You can further improve the transpose example discussed in the previous section by reducing the number of bank conflicts associated with the tile static memory accesses. As described in the previous section, the coalesced transpose kernel copies data into and out of the tile static *localData* column-wise. This means that consecutive threads access array elements that are *tileSize* elements apart. The sample uses a *tileSize* of 32, so thread 0 accesses *localData[0, 0]* while thread 1 accesses *localData[1, 0],* which are in the same bank. The same is true for the remaining threads, resulting in more bank conflicts.

A common approach to reducing the number of bank conflicts is to pad the *tile_static* array. By adding one to the row size, the column wise accesses to *localData* are still at the same indices, but their memory locations no longer lie within the same bank because they are *tileSize*+1 elements apart.

```
parallel_for_each(inData.extent.tile<tileSize, tileSize>(),
    [=, &inData, &outData](tiled_index<tileSize, tileSize> tidx) restrict(amp)
{
    tile_static float localData[tileSize][tileSize + 1];
    localData[tidx.local[1]][tidx.local[0]] = inData[tidx.global];

    tidx.barrier.wait();

    index<2> outIdx(index<2>(tidx.tile_origin[1], tidx.tile_origin[0]) + tidx.local);
    outData[outIdx] = localData[tidx.local[0]][tidx.local[1]];
});
```

This results in a further improvement in performance at the cost of a small amount of unused tile static memory.

Time taken to apply operation to 6400 × 6400 matrix of float

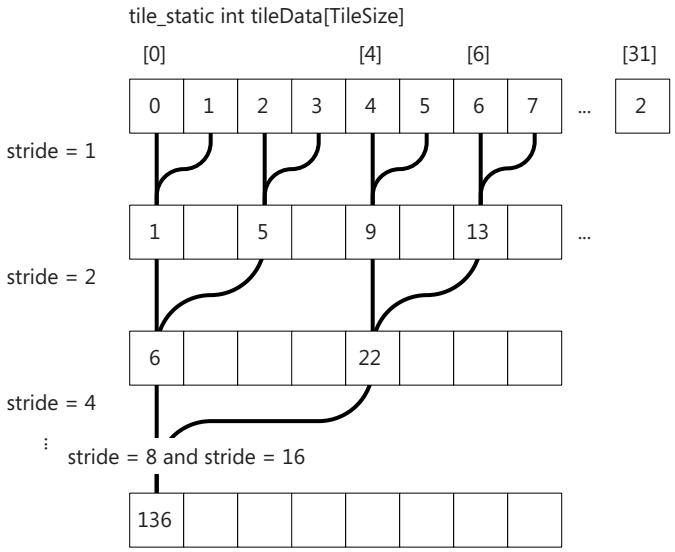
Matrix copy	3.0 ms
Matrix transpose	29.6 ms
Matrix transpose with coalesced global memory access	9.1 ms
Matrix transpose with coalesced global memory access and tile static padding	5.6 ms

The most direct way to reduce the number of bank conflicts is to add padding to the tile static memory to make the array less prime in respect to the tile size. Careful use of padding can also reduce bank conflicts for a variety of tile static memory bank widths.

The Reduction case study also shows an example of bank conflicts in the *TiledMinimizedDivergenceReduction* implementation. The kernel iterates over the *tileData*, adding the values in *tileData[index]* and *tileData[index + stride]* for doubling values of *stride* until a final result is calculated.

```
const int TileSize = 512;
parallel_for_each(view, e.tile< TileSize >(), [=] (tiled_index<TileSize> tidx) restrict(amp)
{
    //  Copy data onto tile static memory
    int tid = tidx.local[0];
    tile_static int tileData[TileSize];
    tileData[tid] = av[tidx.global[0]];

    //  Wait for all threads to finish copying
    tidx.barrier.wait();

    //  Reduce values for data on this tile
    for (int stride = 1; stride < TileSize; stride *= 2)
    {
        int index = 2 * stride * tid;
        if (index < TileSize)
            tileData[index] += tileData[index + stride];

        tidx.barrier.wait();
    }

    //  Write the result for this tile back to global memory
    if (tid == 0)
        tmpAv[tidx.tile[0]] = tileData[0];
});
```

The problem with this kernel is that the memory access pattern within the tile static data creates bank conflicts.



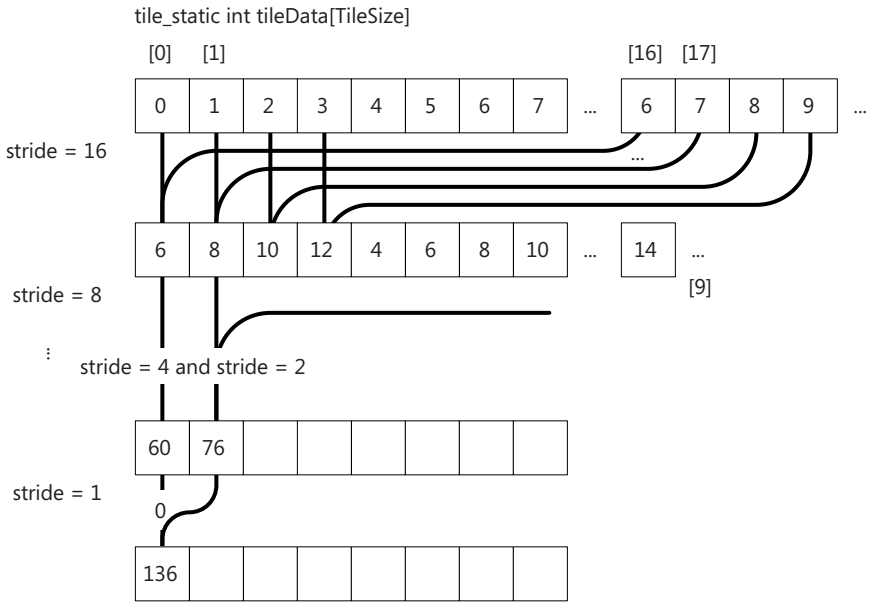
Here, the boxes represent locations within the *tileData* array containing 32 values that are initially set to repeating values 0–15. At each iteration of the loop, threads add elements according to the stride value. For example, in the second iteration of the loop, thread 2 adds *tileData[4]* and *tileData[6]*; 9 + 13 = 22.

This seems fine for the first few elements, but the tile is actually 512 elements long and the stride increments are 1, 2, 4, 8… 256. This means that when thread 16 combines *tileData[32]* and *tileData[33],* a bank conflict occurs with thread 0, which is reading the same banks for elements *tileData[0]* and *tileData[1]*. This is repeated for each iteration of the loop, which results in many bank conflicts.

It's possible to change the memory access pattern to eliminate bank conflicts. The *TiledMinimizedDivergenceAndConflictsReduction* implementation uses a decrementing stride and alters the indexing to combine elements from the upper and lower halves of the array.

```
for (int stride = (TileSize / 2); stride > 0; stride >>= 1)
{
    if (tid < stride)
        tileData[tid] += tileData[tid + stride];
     tidx.barrier.wait();
}
```

This results in the following memory access pattern:



Superficially, this seems the same. Threads access two array elements and combine them. The key difference in this new implementation is that each thread accesses elements that are in the same bank. For example, with a tile size of 512, thread 1 accesses *tileData[0]* and *tileData[256]*, both of which are stored in bank 0, because 256 is exactly divisible by 16 and so also resides in bank 0. Similarly, in the next iteration thread 2 combines elements at 1 and 267, which are both stored in the same bank, bank 1.

The Reduction sample shows another strategy for minimizing bank conflicts. Make sure that threads do not access data stored in the same bank by careful choice of element indexing, taking into account element size. In this example the array contains integers that are exactly one bank wide (32 bits). If your data is larger or smaller, you would need to modify the indexing strategy accordingly. The Reduction case study discusses the results of minimizing bank conflicts in more detail.

Constant Memory

If your kernel makes use of constant data, it makes sense to place this in constant memory, which is a faster memory, local to each CU on the GPU. Constant memory is optimized for all threads in a warp reading the same data. The exact performance of constant memory depends on the access pattern and the target hardware. Using constant memory also has the advantage of freeing up tile static memory and registers for other data. In the simplest case, using constant memory is trivial. Any data captured by value in a C++ AMP kernel is stored in constant memory. For example, here *k* is captured by the default by-value capture. It's stored in constant memory.

```
float k = 1.0f;
parallel_for_each(input.extent, [=, &input, &output](index<1> idx) restrict(amp)
```

```
{
    output[idx] = input[idx] + k;
});
```

Sometimes you might want to use a constant array within a kernel. In this case, you need to wrap the array within a *struct* to avoid attempting to pass a pointer to the kernel, which will result in a compilation error.

```
struct Wrapper
{
    int data[3];
};

void UseArrayConstant()
{
    Wrapper wrap;
    wrap.data[0] = 1;
    // ...

    array<float, 1> input(1000);
    parallel_for_each(input.extent, [wrap, &input](index<1> idx) restrict(amp)
    {
        ... = wrap.data[0];
    });
}
```

In C++ AMP, constant buffers are limited to 16 KB of data; this includes some compiler metadata in addition to the actual data captured by value in the lambda. If you exceed this limit, a compilation error will result.

For further details of the internals of constant memory see "Using Constant Memory in C++ AMP" at *http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/11/using-constant-memory-in-c-amp.aspx*.

Texture Memory

C++ AMP supports both *array* and *texture* data containers. Textures can hold only a limited number of types and do not always support both read and write operations to the same texture within a kernel. The *texture* type allows your program to take advantage of the GPU's texture memory, which is optimized for access patterns with 2D spatial locality and supports hardware packing and unpacking of data.

Textures are covered in detail in Chapter 11, "Graphics," where the "Textures vs. Arrays" section discusses the tradeoffs between the two types and when your application might see performance gains from using textures.

Occupancy and Registers

Keeping all of the CUs on the GPU busy is critical to getting the best performance out of your C++ AMP application. As kernel instructions are executed sequentially when one warp is stalled, the processor hides latencies by executing work from another warp. GPUs hide latency by switching out stalled warps just as CPUs switch out blocked threads. Occupancy is defined as the ratio of the number of executing warps on a CU relative to the number of maximum possible number of warps the CU could support.

Ideally, each CU always has enough warps scheduled to hide latencies and make efficient use of the processor. Warp scheduling is limited by some global resources: tile static memory, threads, and registers. Additional warps can be scheduled on a CU only if there are enough free resources to support it. Here are some general guidelines for improving occupancy:

- Low occupancy almost always means lower performance because there are insufficient warps on the CU to hide any latency. This is especially true for memory-bound kernels because they spend more time waiting for long latency memory accesses. If more warps are available, they can be scheduled on the CU while other warps are waiting on memory accesses.
- Improving occupancy will usually improve performance up to a point, but it's likely to have diminishing returns once there are sufficient warps available to the CU to hide latencies.
- It's a good idea to choose an appropriate tile size so that other resources, like tile static memory, can be used efficiently. See the section of Chapter 4 entitled "Choosing Tile Size" for further discussion of this consideration.
- Register usage is a key factor in determining the occupancy level. If there aren't enough total available registers, the CU is unable to execute warps. Although each CU has thousands of registers, these are divided up among the hundreds of executing threads. This means that each thread might have only tens of registers available to it.
- Using smaller tiles is recommended for better occupancy, except when the reuse of data from using large tiles outweighs the occupancy benefits.

In addition, there is an upper limit of registers per thread. Registers are used to hold variables, and when there aren't enough available registers, the processor will spill variables to global memory, even if the total number of available threads has not been exceeded. Register spilling also results in performance degradation because values must be loaded from memory, which is more expensive and which consumes scarce memory bandwidth.

Calculating occupancy is somewhat specific to the hardware on which your code is running, and register allocation is largely the responsibility of the run time JIT compiler. Both AMD and NVIDIA provide calculation tools to help calculate occupancy on specific hardware. When writing code that targets different hardware at run time, it's better to follow a more general guideline. Minimize the number of local variables within your C++ AMP kernel; this will result in less register allocation. Shortening the lifetime of variable use will also reduce the number of registers required per thread.

Avoiding Divergent Code

Modern CPUs were designed to handle code with lots of divergent branches, efficiently using techniques like branch prediction and speculative execution. Although branching should generally be avoided in inner loops, when it comes to dealing with branching, CPUs are more forgiving than GPUs. GPUs are designed to take advantage of massively parallel operations by providing a large number of simpler processing units that execute the same kernel in parallel. When a GPU processor encounters a branch instruction within a kernel, each branch can no longer execute in parallel.

```
parallel_for_each(input.extent, [&input, &output](index<1> idx) restrict(amp)
{
    if (idx[0] % 2 == 0)
        output[idx] = input[idx] / 42.0f;  // Branch A (even)
    else
        output[idx] = input[idx] * 3.14f;  // Branch B (odd)
});
```

In the example above, the branches are executed sequentially on each warp. While the even threads execute branch A, the odd threads must wait. Next, the odd threads execute branch B while the even threads are idle. This results in 50 percent thread utilization across the tile. Complex Boolean expressions containing *&&* or || operations in *if* statements or the conditional operator (*? ... : ...*) might also lead to further branching. If your code must include branching, try and simplify the Boolean expressions that they use.

The previous example represents a worst case in which all warps must execute divergent code. In some cases only some warps will be divergent. In the next example, only the warp containing thread 98 will diverge because some of its threads will evaluate *idx[0] < 98* as *true* while others will evaluate it to *false,* resulting in divergence. All remaining warps will either evaluate the condition as *true* or *false* for all their threads.

```
parallel_for_each(input.extent, [&input, &output](index<1> idx) restrict(amp)
{
    if (idx[0] < 98)
        output[idx] = input[idx] / 42.0f;  // Branch A
    else
        output[idx] = input[idx] * 3.14f;  // Branch B
});
```

Thinking about how you can arrange your code or data to minimize the number of divergent warps and not just the overall number of code branches might help in cases where branching is unavoidable. For example, take the following example, in which some arbitrary function is applied to all elements of an array, *gpuData*, containing positive single precision numbers.

```
parallel_for_each(view, gpuData.extent, [=, &gpuData](index<1> idx) restrict(amp)
{
```

```
    if (gpuData[idx] > 0.0f)
        gpuData[idx] = fast_math::sqrt(fast_math::pow(gpuData[idx], gpuData[idx]));
});
```

The values in the array are randomly distributed between -10.0 and 10.0, so all warps end up being divergent. If you are able to sort the data prior to running the kernel, then only a single warp will end up diverging—the warp that processes the values closest to zero. The code remains unchanged, but reorganizing the data can reduce the divergence.

Time taken to process an array of 20,000,000 float

Randomly arranged	1.9 ms
Sorted	1.3 ms

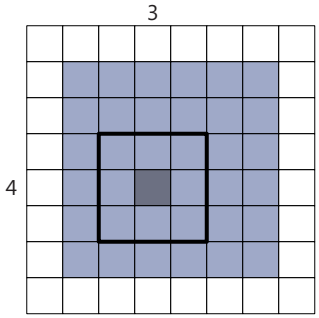
Note that this example is designed to illustrate the impact of memory reordering and does not take into account the time taken to sort the data. You can see the full source code in the *DivergentDataExample* function in main.cpp.

Another less obvious example is loops with a variable number of steps based on the thread index.

```
parallel_for_each(input.extent, [&input, &output](index<1> idx) restrict(amp)
{
    for (int i = 0; i < idx[1] % 10; ++i)
        output[idx] += input(idx + i);
});
```

In this example, all threads within a warp must go through the maximum number of loop steps even though most of them do no calculation for many of the loop steps. This results in a 50 percent reduction in thread utilization.

If at all possible, your code should minimize the use of branching within a kernel. One obvious example of this is when your code applies a stencil operation to a matrix. In this case, the operator calculates the output element value by summing the values in the eight surrounding pixels. The output value of element [4, 3] depends on the eight neighboring elements (within the bounding box). The calculation also must ignore the halo cells (white) because it's not possible to calculate values for elements that do not have eight neighbors.

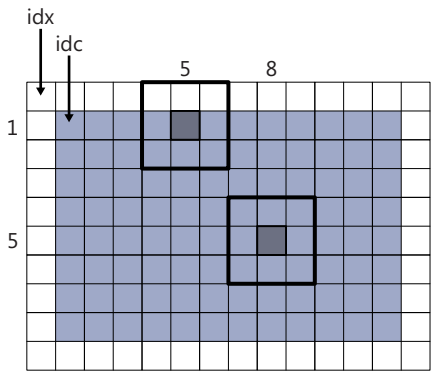


A simple implementation of this algorithm as a C++ AMP kernel looks like this:

```
void ApplyDivergentStencil(const array<float, 2>& input, array<float, 2>& output)
{
    parallel_for_each(input.extent, [&input, &output](index<2> idx) restrict(amp)
    {
        if ((idx[0] >= 1) && (idx[0] < (input.extent[0] - 1)) &&
            (idx[1] >= 1) && (idx[1] < (input.extent[1] - 1)))        // Ignore halo
        {
            output[idx] = 0.0f;
            for (int y = -1; y <= 1; ++y)                             // Loop over stencil
                for (int x = -1; x <= 1; ++x)
                    if ((y != 0) || (x != 0))
                        output[idx] += input(idx[0] + y, idx[1] + x);
        }
    });
}
```

This code contains several branches—first to ensure that halo cells on the edge of the matrix are ignored because the stencil operates only on blocks of nine matrix elements. Then the code uses further branching to support looping over the surrounding matrix elements and ignoring the value in the center of the stencil.

Here is a much more efficient implementation. Instead of using an *if* statement to ignore the elements on the edge of the matrix, the kernel executes over an extent representing the area of the matrix to be calculated (shown in gray in the figure below) and then calculates an corrected index, *idc*, into the input and output arrays.



You can remove the first branching statement by executing the calculation over an *extent* that excludes the halo elements. The second conditional is removed by initializing the *output[idc]* value to contain *-input[idc]*. This allows the loop to run over all nine elements within the stencil without needing to exclude the central value.

```
void ApplyImprovedStencil(const array<float, 2>& input, array<float, 2>& output)
{
    extent<2> computeDomain(input.extent[0] - 2, input.extent[1] - 2);
    parallel_for_each(computeDomain, [&input, &output](index<2> idx) restrict(amp)
    {
        const index<2> idc = idx + index<2>(1, 1);
```

```
        output[idc] = -input[idc];
        for (int y = -1; y <= 1; ++y)
            for (int x = -1; x <= 1; ++x)
                output[idc] += input(idc[0] + y, idc[1] + x);
    });
}
```

An additional loop and the associated conditional can also be removed by using an array, *mask*, to store the stencil offsets. Now all that remains is a single fixed-size loop.

```
void ApplyImprovedStencilMask(const array<float, 2>& input, array<float, 2>& output)
{
    extent<2> computeDomain(input.extent[0] - 2, input.extent[1] - 2);
    parallel_for_each(computeDomain, [&input, &output](index<2> idx) restrict(amp)
    {
        int mask[8][2] = { {-1, -1}, {-1, 0}, {-1, 1},
                           { 0, -1},          { 0, 1},
                           { 1, -1}, { 1, 0}, { 1, 1} };

        index<2> idc(idx + index<2>(1, 1));
        output[idc] = 0.0f;
        for (int i = 0; i < 8; ++i)
            output[idc] += input(idc + index<2>(mask[i]));
    });
}
```

This kernel now contains only a single loop that is not based on a thread index, so the thread utilization is much higher, which makes it more efficient. It runs in less than half the time of the original kernel. The following table shows the results for the three kernels discussed here and an additional unrolled version of the stencil described later in this chapter in the section entitled "Loop Unrolling."

Time taken to array stencil to 4000 × 4000 array of float

Divergent	9.4 ms
Improved	16.9 ms
Improved with mask	3.4 ms
Unrolled (see later in this chapter)	3.3 ms

You can try this for yourself by running the sample in Chapter7\Chapter7.sln. The Cartoonizer sample in Chapter 8 could use this approach for doing color simplification and edge detection on images in the *CalculateSobel()* and *CalculateSobelTiled()* methods (defined in FrameProcessorAmp.cpp).

Choosing the Appropriate Precision

Depending on your application's numerical precision requirements, you might be able to get significant improvements in performance by using less precise but faster math functions or compiler settings, or both. For a more general discussion of numerical accuracy, see "What Every Computer Scientist Should Know About Floating-Point Arithmetic" at *http://docs.oracle.com/cd/E19422-01/819-3693/ncg_goldberg.html*.

In addition to the math functions for use in your CPU code defined in the existing math headers, C++ AMP provides an additional header, amp_math.h. This header declares two sets of math functions for use within amp-restricted lambdas and functions in separate namespaces: *concurrency::precise_math* and *concurrency::fast_math*. There is not a one-to-one correspondence between functions in these two namespaces. The *precise_math* namespace functions are a superset of the *fast_math* namespace functions.

Your code can include the header and use the appropriate namespace to call these libraries.

```
#include <amp_math.h>
using namespace concurrency;
using namespace concurrency::precise_math;

double PreciseSqrt(double x) restrict(amp, cpu)
{
    return sqrt(x);
}
```

Here, the code defines two implementations of *PreciseSqrt()*: an amp-restricted version that calls the precise math implementation of *sqrt()* and a cpu-restricted version that calls the cmath *sqrt()* implementation. This is useful because it saves having to write the two functions with different names.

```
double PreciseSqrtAmp(double x) restrict(amp)
{
    return concurrency::precise_math::sqrt(x);
}

double PreciseSqrtCpu(double x)    // restrict(cpu) implicit
{
    return sqrt(x);                   // cmath function.
}
```

In the next example, the *fast_math::sqrt()* function is explicitly specified. Here, only the amp-restricted version of *FastSqrt()* is implemented explicitly.

```
float FastSqrt(float x) restrict(amp)
{
    return concurrency::fast_math::sqrtf(x);
}
```

The "C++ AMP: Language and Programming Model" specification, section 2.3.2, contains further details on overloading resolution of functions and lambdas marked with the restrict clause. You can find it at *http://download.microsoft.com/download/4/0/E/40EA02D8-23A7-4BD2-AD3A-0BFFFB640F28/CppAMPLanguageAndProgrammingModel.pdf*.

Precise Math Functions

The *concurrency::precise_math* namespace defines both double-precision and single-precision functions. To use any of the precise functions, your target accelerator must support full double precision. This includes functions that take and return only single-precision values because they use double precision internally to ensure the highest accuracy. Use the flag *accelerator::supports_double_precision*

described in the following section to determine if your accelerator supports double precision. For more information about double-precision support, see the "Double-Precision Support" section in Chapter 12.

For a complete list of precise math functions, see "Concurrency::precise_math Namespace" on MSDN: *http://msdn.microsoft.com/en-us/library/hh553049(v=vs.110).aspx*.

Fast Math Functions

The *concurrency*::*fast_math* namespace functions use the DirectX intrinsics and trade accuracy for speed of execution. They do not require an accelerator that supports double precision and have only single-precision implementations. For many applications, like graphics, the precision provided by the fast math functions are sufficient. If your application requires a higher level of accuracy, you should use the functions in the *precise_math* namespace.

For a complete list of fast math functions, see "Concurrency::fast_math Namespace" on MSDN: *http://msdn.microsoft.com/en-us/library/hh553048(v=vs.110).aspx*.

Precise and Fast Compiler Flags

In addition to the C++ AMP library functions, the Visual C++ compiler also has flags for fast (/fp:fast) and precise (/fp:precise) math for operations executing on the CPU and GPU. These flags set the floating-point model used by the compiler and dictate the optimizations that it's able to use.

For a more detailed discussion of these options, see "Microsoft Visual C++ Floating-Point Optimization" on MSDN: *http://msdn.microsoft.com/en-us/library/Aa289157*.

Costing Mathematical Operations

There are a few things worth bearing in mind when you're using general mathematical operations with C++ AMP. Many of these have parallels with writing efficient mathematical code for CPUs.

- Unsigned integer operations are faster than operations on (signed) integers.
- Division operations are more expensive than multiply ones. In some cases, it might be more efficient to calculate the reciprocal and cache it for later use with (more efficient) multiply operations. You should consider the possible numerical inaccuracies that might result from using this approach with floating-point numbers.
- In some cases, it might also be possible to replace integer division operations with faster bitwise operations.
- Use *rsqrtf(x)* in place of *1.0f / sqrtf(x)* for single precision and *rsqrt()* for double precision.

Loop Unrolling

Tight loops, or loops with very few operations within the loop body, represent a potential performance issue on both CPUs and GPUs. In tight loops, the overhead of loop control flow—incrementing loop counters and checking for loop completion—is significant when compared to the small amount of work being done within the loop. One way to address this is to unroll or unwind the loop.

Unrolling a loop might improve performance but does so at the expense of increased program size, register pressure, and code readability. For a general discussion of the advantages and disadvantages of loop unrolling, see *http://en.wikipedia.org/wiki/Loop_unwinding*.

You can try another optimization in the stencil example discussed in the section on divergent code. The loop used to update the output array does very little work within the loop.

```
extent<2> computeDomain(input.extent[0] - 2, input.extent[1] - 2);
parallel_for_each(computeDomain, [&input, &output](index<2> idx) restrict(amp)
{
    int mask[8][2] = { {-1, -1}, {-1, 0}, {-1, 1},
                       { 0, -1},          { 0, 1},
                       { 1, -1}, { 1, 0}, { 1, 1} };

    const index<2> idc = idx + index<2>(1, 1);
    output[idc] = 0.0f;
    for (int i = 0; i < 8; ++i)
        output[idc] += input(idc + index<2>(mask[i]));
});
```

In this case, it would be possible to completely unroll the loop, entirely removing the need to test for loop completion. For fixed-size loops, the loop can be partially unrolled by a divisor of the loop size.

```
extent<2> computeDomain(input.extent[0] - 2, input.extent[1] - 2);
parallel_for_each(computeDomain, [&input, &output](index<2> idx) restrict(amp)
{
    const index<2> idc = idx + index<2>(1, 1);
    output[idc] =  input(idx[0], idx[1]);
    output[idc] += input(idx[0], idx[1] + 1);
    output[idc] += input(idx[0], idx[1] + 2);
    output[idc] += input(idx[0] + 1, idx[1] + 1);
    output[idc] += input(idx[0] + 1, idx[1] + 2);
    output[idc] += input(idx[0] + 2, idx[1]);
    output[idc] += input(idx[0] + 2, idx[1] + 1);
    output[idc] += input(idx[0] + 2, idx[1] + 2);
});
```

In this example, fully unrolling the loop results in a slight improvement in performance, as shown in the table in the section of this chapter entitled "Avoiding Divergent Code." The NBody sample also uses this technique in the *NBodyAmpTiled::TiledBodyBodyInteraction()* method (defined in NBodyAmp.h) and achieves modest performance improvements.

It's important to understand that manual loop unrolling might result in degraded performance due to increased register pressure or code size. It might also prevent the JIT compiler from further

optimization. The DirectCompute JIT is responsible for loop unrolling, so you need to test the performance of any unrolled loops on the target hardware platform(s). Experimentation is usually required to determine if unrolling will improve performance and the correct amount of unrolling on each target hardware platform. The "Loop Unrolling" section in Chapter 8, "Performance Case Study—Reduction" shows an example where loop unrolling doesn't have a positive effect on performance.

Barriers

Barriers are synchronization primitives that allow you to control the flow of execution of groups of threads. They allow you to halt the execution of each thread until all threads involved in the barrier have reached it. For a general discussion of barriers, see the Wikipedia entry *http://en.wikipedia.org/wiki/Barrier_(computer_science)*.

C++ AMP provides the *tile_barrier* class to allow your program to control the synchronization of threads within a tile. The *tile_barrier* supports several methods that cause each thread within a tile to wait until all threads in a tile have reached the barrier. The *tiled_index::barrier* property exposes a *tile_barrier* instance to each tiled thread. A *tile_barrier* cannot be constructed, although it can be copied.

Many of the samples have already used barriers. A very common pattern in C++ AMP code uses barriers to synchronize reading and writing to *tile_static* variables that are shared between threads. The following transpose example was already discussed in the previous section, "Efficient Accelerator Global Memory Access."

```
array<float, 2> inData(1000, 1000);
array<float, 2> outData(1000, 1000);

parallel_for_each(view, inData.extent.tile<tileSize, tileSize>(), [=, &inData, &outData]
    (tiled_index<tileSize, tileSize> tidx) restrict(amp)
{
    tile_static float localData[tileSize][tileSize];
    localData[tidx.local[1]][tidx.local[0]] = inData[tidx.global];

    tidx.barrier.wait();

    index<2> outIdx(index<2>(tidx.tile_origin[1], tidx.tile_origin[0]) + tidx.local);
    outData[outIdx] = localData[tidx.local[0]][tidx.local[1]];
});
```

Here, the *barrier::wait()* call ensures that all the threads have written their data to the tile static *localData* before any of the threads try to read data from *localData*. Without the barrier, race conditions will lead to incorrect results. For example, thread [2, 1] could read data from *localData[1, 2]* before thread [1, 2] had written a value.

Compilers and CPUs frequently reorder the execution of instructions to optimize performance. This can include changing the order of memory read and write operations. The compiler and CPU will guarantee that any operation reordering will not alter the correctness of code running on a single thread. However, it makes no such guarantees for code running on multiple threads and reading and

writing to shared memory. This lack of guarantees could lead to correctness errors if the compiler were to reorder read or write operations from one side of the barrier to the other.

Memory fences cause the compiler and processor to enforce the ordering of memory operations before and after the fence. Reordering can still occur before and after the fence but not across the fence. Memory fences are one mechanism to prevent correctness errors when sharing data between threads. See Wikipedia for further discussion of memory fences: *http://en.wikipedia.org/wiki/Memory_barrier*.

In C++ AMP, a barrier always implies a memory fence. In the preceding example, the *tile_barrier::wait()* method doesn't just synchronize execution—it also implements memory fences for both global and tile static memory operations. This ensures that read and write operations to *localData* are not moved from one side of the barrier to the other by compiler or processor optimization.

C++ AMP supports barriers, each with different memory fence semantics. The *tile_barrier::wait()* method is equivalent to *tile_barrier::wait_with_all_memory_fence()*. You can consider the *wait()* method shorthand for *wait_with_all_memory_fence()*. This prevents the reordering for memory operations to both global and tile static memory. In addition to these two methods, *tile_barrier* supports an additional method that implements a barrier for global memory operations.

```
void tile_barrier::wait_with_global_memory_fence() const restrict(amp)
```

This enforces synchronization of all tile threads and also enforces memory ordering of global memory operations with respect to the barrier. A further method implements a barrier for tile static memory operations.

```
void tile_barrier::wait_with_tile_static_memory_fence() const restrict(amp)
```

C++ AMP also supports stand-alone functions that implement memory fences for all global and tile static operations. Memory fences only prevent reordering of read and write operations. If your program requires synchronization between threads, you must use a barrier.

```
void all_memory_fence(const tile_barrier & barrier) restrict(amp)
void global_memory_fence(const tile_barrier & barrier) restrict(amp)
void tile_static_memory_fence(const tile_barrier & barrier) restrict(amp)
```

Performance Impact of Barriers and Fences

Barriers are blocking operations, so from a performance perspective, you should use them sparingly. Memory fences are nonblocking but restrict the compiler's and CPU's ability to efficiently optimize code. Use the least restrictive memory fence that still meets the correctness requirements of your program. For example, in the previous example, the *tile_barrier::wait()* can be relaxed to use only a tile static memory barrier.

```
parallel_for_each(view, inData.extent.tile<tileSize, tileSize>(), [=, &inData, &outData]
    (tiled_index<tileSize, tileSize> tidx) restrict(amp)
{
    tile_static float localData[tileSize][tileSize];
```

```
    localData[tidx.local[1]][tidx.local[0]] = inData[tidx.global];

    tidx.barrier.wait_with_tile_static_memory_fence();

    index<2> outIdx(index<2>(tidx.tile_origin[1], tidx.tile_origin[0]) + tidx.local);
    outData[outIdx] = localData[tidx.local[0]][tidx.local[1]];
});
```

A good practice is to start using the *wait()* or *wait_with_all_memory_fence()* barriers to ensure that your program produces correct results and then optimize this by weakening the memory fence constraints of the barrier after your application is working. Similarly, if your program uses memory fences, you should take the same approach of using the most restrictive fence (*all_memory_fence()*) to ensure correctness and then weakening it if possible. As always, you should always measure the performance and understand your application's performance goals.

Using Barriers Correctly

C++ AMP requires that if one thread in a tile encounters a barrier, all threads in that tile must encounter the same barrier. Replacing the *barrier::wait()* call in the previous example with the following code will result in a compiler error because only the even-indexed threads will execute the *wait()*.

```
if (tidx.local[0] % 2 == 0) // Apply barrier only to even indexed threads.
    tidx.barrier.wait();
```

All threads in a tile must reach the barrier via the same sequence control flow statements and/or expressions. The following code will not compile. Even though all threads reach the barrier, they do so via different code paths.

```
if (tidx.local[0] % 2 == 0) // Apply barrier only to even indexed threads.
{
    inData[tidx.global] = 0;
    tidx.barrier.wait();
}
else
    tidx.barrier.wait();
```

You can use branching code, provided branching expression uses only variables, literals, or function calls that have a uniform value across all threads in the tile. The following is valid code because *x* does not vary across the threads in the tile.

```
int x = 2;
// ...
if (x == 2)
    tidx.barrier.wait();
```

Similarly, the following is also valid because *tidx.tile* is constant for all threads in each tile, although not across threads in different tiles.

```
if (tidx.tile[0] % 2 == 0)
    tidx.barrier.wait();
```

In most cases, the (HLSL) compiler can detect incorrect usage of barriers, but in some cases, you might see undefined behavior at run time. The C++ AMP specification Section 8.1.1 contains further explanation of the correct usage of tile barriers.

Queuing Modes

As discussed earlier, the C++ AMP run time queues work to accelerators asynchronously. Data-copying to and from the accelerator as well as *parallel_for_each* kernel execution are queued and then submitted. Work is scheduled on the GPU as a DMA buffer that contains commands (kernels) to be executed, along with references to the memory resources used by those commands. The Windows Device Driver Model (WDDM) virtualizes the GPU's resources and ensures that all resource allocations required by a DMA buffer are paged into GPU memory before the buffer is executed.

C++ AMP supports two queuing modes: immediate and automatic. They are defined in the *queuing_mode* enumeration.

```
enum queuing_mode {
    queuing_mode_immediate,
    queuing_mode_automatic
};
```

By default, the C++ AMP run time uses *queuing_mode::queuing_mode_automatic* when submitting work to the GPU. The hardware device driver will automatically queue new commands and then submit them to the GPU as batches in the following cases:

- A command to copy data to the host or another accelerator view is queued. This causes all the previous commands that reference the data—and the copy command itself—to be submitted for execution.
- A call to *accelerator_view::flush()* is made. This causes all queued commands to be submitted. Note that *flush()* itself does not block, so although this submits all commands, it does not make any guarantees as to their completion on return.
- A call to *accelerator_view::wait()* is made. The *wait()* flushes all commands and then blocks until execution has completed.
- If the code uses *accelerator_view::create_marker()* to create a *completion_future* on the CPU, it is bound to a command on the accelerator view. Calling *wait()* on the future will block until the command completes on the accelerator view. For further discussion of this approach, see *http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/28/accelerator-view-create-marker-in-c-amp.aspx*.
- When the hardware device driver's internal heuristic determines that no further work can be queued. The exact heuristic used varies according to the hardware vendor. In general, drivers use a model that submits queued work when adding further work to the queue would exceed the total resource requirements available on the hardware.

You can set the queuing mode when creating an accelerator view. If no queuing mode is specified, the default queuing mode *queuing_mode::queuing_mode_automatic* is used.

```
accelerator acc(accelerator::default_accelerator);
acc.create_view(queuing_mode::queuing_mode_immediate);
```

More information on the inner workings of WDDM can be found on MSDN in "Video Memory Management and GPU Scheduling" at *http://msdn.microsoft.com/en-us/library/ff570508(v=vs.85).aspx*.

In general, you should use automatic mode for queuing work to accelerators because batching makes the most efficient use of the hardware.

Automatic mode uses queuing to improve throughput at the expense of latency. Commands submitted to the queue will remain in the queue until one of the conditions outlined previously is met. If your application has specific latency requirements, you might want to consider using immediate mode or making explicit calls to *accelerator_view::flush()* or *wait()*. This enables the application to reduce latency by starting execution of code immediately on the GPU. This might be at the expense of overall throughput because C++ AMP will no longer be able to take advantage of batching commands for execution.

Summary

Before embarking on any performance optimization, you should measure your application's performance to understand where the bottlenecks are and establish a baseline for measuring improvements. You need to understand if your application is compute-bound or memory-bound because this dictates what the most effective optimization strategies are likely to be.

Investigate algorithmic improvements to remove kernel divergence and optimize memory access patterns before focusing on code optimizations like loop unrolling. In general, the following are the most significant areas on which to focus:

- Minimize data transfers to and from CPU and GPU and transfer only the data you actually need.
- Ensure that global memory accesses are coalesced.
- Use tile static memory instead of global memory.
- Avoid branching within kernels.

This should be considered only a rough guideline, and it depends greatly on the nature of your application. Using the iterative measurement approach outlined at the beginning of this chapter is the best way to ensure that you get good results.

Depending on the target platform(s) for your application, you might need to do further profiling and measurement on different hardware platforms and to be aware of the differences in their architectures and performance characteristics. Remember that making your code too specific to one platform might lead to performance degradation—or even to correctness errors on other platforms.

CHAPTER 8

Performance Case Study—Reduction

In this chapter:

The Problem 171

Case Study Structure 172

CPU Algorithms 178

C++ AMP Algorithms 179

Summary 201

The Problem

An uncountable number of algorithms are used in applications that do a lot of number-crunching. It's often useful to think of them as falling into several large groups and to distinguish differences between the groups. Two of the largest groups are:

- Algorithms that take a number of data points and perform some sort of transformation that produces a similarly large number of data points. For example, matrix multiplication might involve thousands of points in each matrix. When the transformation is complete, there are also thousands of points in the result matrix.
- Algorithms that take a number of data points and perform some sort of transformation that produces significantly fewer points. In the most extreme case, they produce only one point—the total of all the input points, or the largest value, or the average value, or some other single value.

Problems such as matrix multiplication, cartoonizing, and n-body simulation are in the first category; they involve a many-to-many transformation. The second group's many-to-fewer transformation is called reduction; it refers to algorithms that transform a set of data points into a smaller set, maybe even a single data point. Optimizing each type of problem often requires different approaches.

To make sure that both kinds of algorithms are discussed in this book, the performance case study uses a simple reduction. How simple? After the data points to be totaled have been stored into the *source* vector, the sequential version of it is just one line long:

```
int total = std::accumulate(source.cbegin(), source.cend(), 0, std::plus<int>());
```

In this chapter, a C++ AMP version of this single-line calculation will be repeatedly implemented as a way to demonstrate some techniques that might be useful across a wide range of applications. The techniques do not all have a significant impact on a reduction (in fact, some worsen performance), but all are worth discussing. Understanding these techniques is valuable when you're implementing other algorithms with C++ AMP.

A Small Disclaimer

It would be foolish to try to use C++ AMP to increase the performance of a reduction like this in isolation. If your application does nothing but add up a collection of numbers, it's probably not a good candidate for performance improvement through C++ AMP. Adding is a relatively cheap operation (especially adding integers), and the time saved through offloading that operation to the GPU is unlikely to outweigh the time spent copying the source data to the GPU for manipulation there. Reduction is used here because it shows many of the techniques covered in Chapter 7, "Optimization," and also introduces Reduction, which is an important class of algorithms for developers programming with C++ AMP. A reduction of integers avoids correctness problems that can occur with floating-point operations. (Because addition is not always commutative with floating-point numbers, parallelizing can change the result.)

That said, you are quite likely to write a real application that performs a many-to-many operation followed by a reduction. Or perhaps it does several many-to-many operations and several reductions, gathers up the results of the reductions, does a many-to-many with those, and then does a final reduction to come up with a meaningful answer. In that context, being able to make the reductions as fast as possible will be useful. And, of course, you can apply many of the techniques of this chapter to other algorithms. Just please do not take the use of this algorithm in this chapter as a claim that calculating the total of a collection of integers is a great use case for C++ AMP.

Case Study Structure

The Reduction case study consists of a number of implementations of the same arithmetic reduction performed on the same source data so that the time taken for each implementation can be compared. Rather than random numbers, the source data follows a pattern—the numbers 0 to 15 repeat to fill the array—which means that the correct total is already known, allowing the code to check for correctness as well as speed.

Each of the algorithms is implemented as a class that inherits from *IReduce* and implements the pure virtual *Reduce()* function defined in IReduce.h:

```
class IReduce
{
public:
    virtual int Reduce(accelerator_view& view,
        const std::vector<int>& source, double& computeTime) const = 0;
};
```

The *main()* function (in Reduction.cpp), after declaring and initializing some constants like the tile size that will be discussed later in this chapter, populates the source vector and calculates the total:

```
const size_t elementCount = 16 * 1024 * 1024;

// . . .
std::vector<int> source(elementCount);
int i = 0;
std::generate(source.begin(), source.end(), [&i]() { return (i++ & 0xf); });

const int expectedResult = int((elementCount / 16) * ((15 * 16) / 2));
```

Then a vector of *ReducerDescription* instances is created, each of which is just a *std::pair* of a *std::shared_ptr* to a reducer instance and a string description:

```
typedef std::pair<std::shared_ptr<IReduce>, std::wstring> ReducerDescription;
// . . .

std::vector<ReducerDescription> reducers;
reducers.reserve(14);
reducers.push_back(ReducerDescription(std::make_shared<DummyReduction>(),
                                      L"Overhead"));
reducers.push_back(ReducerDescription(std::make_shared<SequentialReduction>(),
                                      L"CPU sequential"));

// ... 12 similar push_back calls omitted ...

reducers.push_back(ReducerDescription(
    std::make_shared<CascadingUnrolledReduction<tileSize, tileCount>>(),
    L"C++ AMP cascading reduction & unrolling"));
```

With a vector of reducers in place, *main()* runs the same reduction with each of them:

```
accelerator_view view = accelerator(accelerator::default_accelerator).default_view;
for (size_t  i = 0; i < reducers.size(); ++i)
{
    int result = 0;
    IReduce* reducerImpl = reducers[i].first.get();
    std::wstring reducerName = reducers[i].second;

    double computeTime = 0.0, totalTime = 0.0;
    totalTime = JitAndTimeFunc(view, [&]()
    {
        result = reducerImpl->Reduce(view, source, computeTime);
    });
```

Notice that each call to *Reduce()* is actually in a lambda that is being passed to *JitAndTimeFunc()*. The *JitAndTimeFunc()* details are in a later section. Finally, *main()* prints the results—some reducers will refuse to run on emulators and will return -1 to indicate this. If the reducer ran, the result is checked for correctness, and if it's correct, the time taken is printed.

```
        if (result == -1)
        {
            std::wcout << "SKIPPED: " << reducerName << " - Accelerator not supported."
                << std::endl;
            continue;
        }
        if (expectedResult != result)
        {
            std::wcout << "FAILED: " << reducerName << " expected " << expectedResult
                << std::endl
                        << "         but found " << result << std::endl;
            continue;
        }

        std::wcout << "SUCCESS: " << reducerName;
        std::wcout.width(max(0, 55 - reducerName.length()));
        std::wcout << totalTime << " : " << computeTime << " (ms)" << std::endl;
    }
```

The following is a typical output from running this code:

```
Running kernels with 16777216 elements, 65536 KB of data ...
Tile size:     512
Tile count:    128
Using device : NVIDIA GeForce GTX 580 (Microsoft Corporation-WDDM v1.2)
                                                              Total : Calc
SUCCESS: Overhead                                              0.13 : 0.00 (ms)
SUCCESS: CPU sequential                                       12.89 : 12.82 (ms)
SUCCESS: CPU parallel                                          3.91 : 3.84 (ms)
SUCCESS: C++ AMP simple model                                 33.57 : 5.38 (ms)
SUCCESS: C++ AMP simple model using array_view                60.88 : 27.15 (ms)
SUCCESS: C++ AMP simple model optimized                       27.62 : 2.05 (ms)
SUCCESS: C++ AMP tiled model                                  39.64 : 18.08 (ms)
SUCCESS: C++ AMP tiled model & shared memory                  33.90 : 7.65 (ms)
SUCCESS: C++ AMP tiled model & minimized divergence           31.45 : 5.56 (ms)
SUCCESS: C++ AMP tiled model & no bank conflicts              30.13 : 4.20 (ms)
SUCCESS: C++ AMP tiled model & reduced stalled threads        28.82 : 2.84 (ms)
SUCCESS: C++ AMP tiled model & unrolling                      27.98 : 1.97 (ms)
SUCCESS: C++ AMP cascading reduction                          28.14 : 1.48 (ms)
SUCCESS: C++ AMP cascading reduction & unrolling              27.07 : 1.53 (ms)
```

The meaning of these results will be discussed in the following sections.

Initializations and Workload

A number of constants are used in this application, and they are not independent. The tiled implementations use a tile size, and the cascading reductions use a tile count that must meet certain constraints:

- The total element count must be a multiple of the tile size.
- The number of tiles can't exceed 65,536.
- The total element count must be a multiple of the product of the tile size and the tile count.

This code sets up the appropriate constants and checks them:

```
const size_t elementCount = 16 * 1024 * 1024;
const int tileSize = 512;
const int tileCount = 128;                    // Used in cascading reductions

// Make sure that elements can be split into tiles so the number of
// tiles in any dimension is less than 65536.
static_assert((elementCount / tileSize < 65536),
    "Workload is too large or tiles are too small. This will cause runtime errors.");
static_assert((elementCount % (tileSize * tileCount) == 0),
    "Tile size and count are not matched to element count.");
static_assert((elementCount != 0), "Number of elements cannot be zero.");
static_assert((elementCount <= UINT_MAX), "Number of elements is too large.");

std::wcout << "Running kernels with " << elementCount << " elements, "
             << elementCount * sizeof(int) / 1024 << " KB of data ..."  << std::endl;
std::wcout << "Tile size:     " << tileSize << std::endl;
std::wcout << "Tile count:    " << tileCount << std::endl;
if (!validateSizes(tileSize, elementCount))
    std::wcout << "Tile size is not factor of element count."
                << std::endl;
```

Why check values that are hard-coded? A common use for this sample code is to experiment and tweak various settings. You might try increasing or decreasing the workload, for example, or making the tiles larger or smaller, but not all combinations of values will meet the constraints mentioned above. A compile time check and warning will remove some of the mystery from the process. When a *static_assert* fails, the build fails and you can't run the application. If you want to trigger the run-time errors in order to see what they look like, comment out the *static_assert* statements before you build and run the application with invalid values.

Concurrency Visualizer Markers

In addition to the timing results printed out by this console application, you can gain insight from using the concurrency visualizer to watch the execution proceed on the CPU and your accelerator. This process is easier if you use markers to tie specific places in the code to activity that you see in the concurrency visualizer. These have been enabled in the sample code for this chapter in the CaseStudies\Reduction folder. To enable markers for a project of your own, you need to do the following:

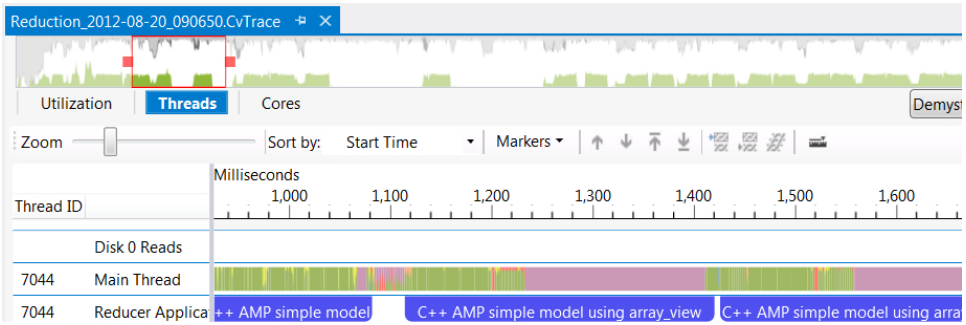
- Ensure that you are running the sample using Microsoft Visual Studio 2012 Professional, Premium, or Ultimate. The Concurrency Visualizer is not supported on other editions of Visual Studio.

- From the Analyze menu, open the Concurrency Visualizer | Add SDK To Project dialog box.
- Select your project, click Add SDK To Selected Project, and then close the dialog box.

You'll find a number of places in reduction.cpp protected by *#ifdef* statements. (To turn off markers in your concurrency visualizer output, you can comment away the *#define* statement defining *MARKERS* in Reduction.cpp.) First, this application declares a global right before *main()*:

```
marker_series g_markerSeries(L"Reducer Application");
```

The *marker_series* class is in the *concurrency::diagnostics* namespace and represents a lane or channel of markers in the Concurrency Visualizer, like this:



Markers can be either spans (as shown in this image) or flags—diamond-shaped glyphs. Code shown later in this chapter will use this *g_markerSeries* variable to add markers to any visualizations that you perform on the running code.

TimeFunc()

Each implementation of *Reduce()* is not called directly from *main()* but from a lambda that's passed to *JitAndTimeFunc()*, which is a function that is used to time how long an operation takes. You can find two timing functions in Timer.h. The first, *JitAndTimeFunc()*, takes an *accelerator_view* and a function and calls the second, *TimeFunc()*:

```
template <typename Func>
double JitAndTimeFunc(accelerator_view& view, Func f)
{
    //  Ensure that the C++ AMP runtime is initialized.
    //  Ensure that the C++ AMP kernel has been JITed.
    f();
    return TimeFunc(view, f);
}
```

This code starts by calling the C++ AMP method, *f()*, just to be completely sure that the C++ AMP run time is initialized. This takes care of any delays associated with JIT-ing the function and other initializations. If performance matters in your application, it's almost certainly because your kernels are run thousands of times, making differences of a few hundred milliseconds per run important. Under those conditions, the JIT-ing and initialization times are not important, and that's why this function

eliminates them from the timing process. For further details of how to measure kernel performance, see Chapter 7.

The second function, *TimeFunc()*, actually times the function being measured. It looks like the following:

```
template <typename Func>
double TimeFunc(accelerator_view& view, Func f)
{
    //  Wait for all previous accelerator work to end.
    view.wait();

    LARGE_INTEGER start, end;
    QueryPerformanceCounter(&start);

    f();

    //  Wait for all accelerator work to end.
    view.wait();
    QueryPerformanceCounter(&end);
    return ElapsedTime(start, end);
}
```

The *ElapsedTime()* function just deals with converting the difference between two performance counter values into milliseconds:

```
inline double ElapsedTime(const LARGE_INTEGER& start, const LARGE_INTEGER& end)
{
    LARGE_INTEGER freq;
    QueryPerformanceFrequency(&freq);
    return (double(end.QuadPart) – double(start.QuadPart)) * 1000.0 / double(freq.QuadPart);
}
```

The calls to *wait()* on the view ensure that any work that was queued earlier is completed and that the function *f()* has completely finished before the timing stops. The function *f()* is actually called twice on each reduction:

- In *main()*, the loop that calls each reduction in turn uses *JitAndTimeFunc()* to determine the overall time taken, including the time to copy the source vector to the accelerator.
- In each implementation of *Reduce()*, *TimeFunc()* is called after the *array* is initialized to determine the time just for the computation.

Without the calls to *accelerator_view::wait()*, the copy time might not be correctly recorded. (If you're timing a different algorithm, you might need to adapt this strategy to record the time to copy results back from the accelerator to the CPU. Because this reduction ships back only a single integer, that time is not measured in this case study.)

Overhead

This case study is not the only sample you can find that takes a simple arithmetic reduction and applies a series of optimizations and alternate implementations. Most don't refactor the code into several classes with virtual functions and templates. There might be a concern that this is introducing overhead. To test for that, the file DummyReduction.h has an implementation of *DummyReduction*, a class whose *Reduce()* function just returns the known-to-be-correct answer:

```
class DummyReduction : public IReduce
{
public:
    int Reduce(accelerator_view& view, const std::vector<int>& source,
        double& computeTime) const
    {
        return int(source.size() / 16) * ((15 * 16) / 2);
    }
};
```

This is obviously the quickest of all the ways to calculate the answer, and any time taken by the function can be ascribed to the overhead caused by the structure of the case study. In dozens of runs this time was usually zero, but occasionally as much as 1 ms. Measuring your overhead like this is a good way to ensure that you draw the correct conclusions about things like the percentage improvement you have achieved when optimizing or reimplementing an application.

CPU Algorithms

This sample uses two CPU algorithms: one sequential and one parallel.

Sequential

Adding up all the numbers in a collection is such a common behavior that there's a function in the Standard Library to make it easy. In SequentialReduction.h, you will find the nonparallel approach to this reduction:

```
class SequentialReduction : public IReduce
{
public:
    int Reduce(accelerator_view& view, const std::vector<int>& source,
        double& computeTime) const
    {
        int total = 0;
        computeTime = TimeFunc(view, [&]()
        {
            total = std::accumulate(source.cbegin(), source.cend(), 0, std::plus<int>());
        });
        return total;
    }
};
```

The results (and the time) are the same if you omit the last parameter to *std::accumulate()* because by default *accumulate()* adds up all the elements of the collection. Using *std::plus<int>()* just makes it clear what this code does without expecting anyone to remember the default operation for *accumulate()*.

Parallel

Depending on how many CPU cores you have available, you might get a significant performance improvement from using the PPL. An arithmetic reduction is easy to parallelize across CPU cores. In fact, the PPL library includes a *parallel_reduce()* function for just this purpose. In measurements during the development of this case study, *parallel_reduce()* outperformed a *parallel_for* or *parallel_for_each* that added the source elements into a running total. Here's how the class looks:

```
class ParallelReduction : public IReduce
{
public:
    int Reduce(accelerator_view& view, const std::vector<int>& source,
        double& computeTime) const
    {
        int total;
        computeTime = TimeFunc(view, [&]()
        {
            total = parallel_reduce(source.cbegin(), source.cend(), 0, std::plus<int>());
        });
        return total;
    }
};
```

Like the sequential implementation, this parallel implementation uses *std::plus<int>()* to accumulate the elements of the *source* vector.

C++ AMP Algorithms

As mentioned earlier, an application that only performs a reduction is probably not a good candidate for performance improvement with C++ AMP. The time taken to copy the source data to the accelerator is likely to outweigh the time spent adding the elements, even in the sequential CPU version. (For example, the sample results shown earlier in this chapter required a copy time of approximately double the sequential accumulation time. Those results might not be typical, but you should expect copy time to outweigh accumulation time on any hardware.) It thus becomes impossible for C++ AMP code to execute a single reduction more quickly overall (that is, including copy time) than CPU-based code. However, if the reduction is part of a multistep calculation on the accelerator or if the data needs to go to a GPU anyway in order to be rendered, the importance of the copy time is diminished and it becomes worthwhile to ensure that your C++ AMP code is as fast as possible.

Keep in mind that each optimization makes the code harder to read, write, and maintain. It might also be harder to debug for correctness. The relative benefit of each succeeding optimization depends on the algorithm and on the hardware where it executes. Without a deep understanding of

C++ AMP and GPU hardware, it is hard to predict what kind of improvement a particular approach will produce. It's even possible that some approaches will worsen performance for some algorithms. Chapter 4, "Tiling," shows how a badly chosen tile size can produce results that are worse than the simple algorithm in which you don't use a *tiled_extent* or access any *tile_static* memory. The series of improvements shown here were tested on a variety of hardware and showed speedups of up to 15× compared to the sequential version (ignoring copy time). By testing your code on a variety of hardware as you implement, with different potential optimizations, you can balance the costs and benefits of the reimplementation for your algorithm and your expected hardware.

Simple

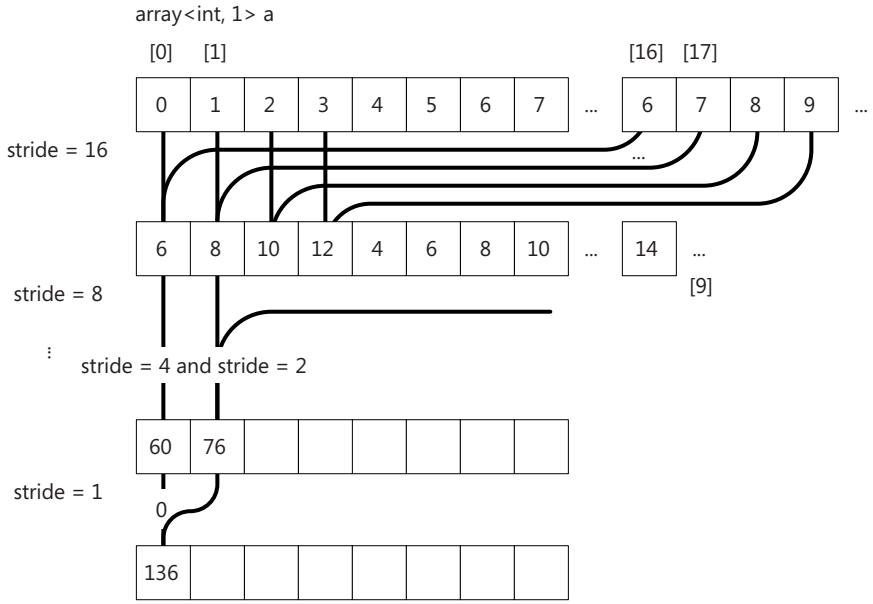
The simple C++ AMP code uses an ordinary *extent* and does not access *tile_static* memory. The class is barely larger than the sequential or parallel version, but it does have a few twists:

```
class SimpleReduction : public IReduce
{
public:
    int Reduce(accelerator_view& view, const std::vector<int>& source,
        double& computeTime) const
    {
        assert(source.size() <= UINT_MAX);
        int elementCount = static_cast<int>(source.size());

        // Copy data
        array<int, 1> a(elementCount, source.cbegin(), source.cend(), view);
        std::vector<int> result(1);
        int tailResult = (elementCount % 2) ? source[elementCount - 1] : 0;
        array_view<int, 1> tailResultView(1, &tailResult);
        computeTime = TimeFunc(view, [&]()
        {
            for (int stride = (elementCount / 2); stride > 0; stride /= 2)
            {
                parallel_for_each(view, extent<1>(stride), [=, &a] (index<1> idx) restrict(amp)
                {
                    a[idx] += a[idx + stride];

                    // If there are an odd number of elements then the first thread
                    // adds the last element.
                    if ((idx[0] == 0) && (stride & 0x1) && (stride != 1))
                        tailResultView[idx] += a[stride - 1];
                });
            }

            //  Only copy out the first element in the array as this contains the final answer.
            copy(a.section(0, 1), result.begin());
        });
        tailResultView.synchronize();
        return result[0] + tailResult;
    }
};
```

If you were to write this code from scratch, you might plan to declare an *array_view* of *const int,* rather than just *int,* to save it from being copied back to the CPU afterward. After all, the sequential and parallel versions of this algorithm don't change the source data at all. However, the C++ AMP code accumulates the partial sums that lead to the answer within the *source* array itself to avoid allocating additional memory on the accelerator. Therefore, you can't use an *array_view* of *const*.

A first draft of the simple C++ AMP algorithm might also write a simpler loop, perhaps with a single *parallel_for_each* that just adds each element to a total. This will require as many threads as there are elements in the source, but the threads won't be able to run truly parallel and return the correct result due to a race condition involving the location where the total is being stored. One thread looks at the current value of the running total, adds an element to it, and then writes the answer back. But what if another thread had looked at that same starting value and now they race to write back the new total? Whichever thread wins, the answer won't be correct. An atomic += operation might ensure correctness, but at the cost of true parallelism. The multiple threads must line up and wait to read and write the total in turns. Atomic operations, and why they should be used with great care, are described in Chapter 12, "Tips, Tricks, and Best Practices."

The code shown here avoids this problem by having each thread in the *parallel_for_each* write its partial sum to a different location in a large array. It's convenient to use the *array* that was passed in for this purpose. The first pass through this code, with *stride* equal to half the element count, uses the lower half of the source array to store the running totals. Each thread writes to a different element of the array, and there are no race conditions. The next pass uses the lower quarter of the source array, the third uses the lower eighth, and so on. Because the number of elements might not be a power of two, each *parallel_for_each* deals with any possible leftover elements by giving a little extra work to the first thread. (If you can arrange the problem space to ensure that the number of elements was a power of two, this complication can be eliminated.) The following is a diagram of the threads working toward the total by using an artificial array of 32 elements:



When all the work is over, the first element in the array contains the total. There's no need to copy the entire array back to the CPU—only this single value is needed. By using an *array* instead of an *array_view* you can take control of the way data is copied and copy only the first element back to the CPU by using *a.section(0, 1)*.

Simple with *array_view*

Sometimes, seemingly small decisions can have a much larger effect than you thought—and not always for the better. Consider the *SimpleArrayViewReduction* class. Its *Reduce()* method looks very much like the one in *SimpleReduction*, with two differences: it uses an *array_view* instead of an *array*, and it makes a local copy of a *vector* for that *array_view* to wrap:

```
int Reduce(accelerator_view& view, const std::vector<int>& source,
    double& computeTime) const
{
    int elementCount = static_cast<int>(source.size());

    // Copy data, create a writable copy that can be associated with the array_view.
    std::vector<int> writableSource(source.size());
    std::copy(source.cbegin(), source.cend(), writableSource.begin());
    array_view<int, 1> av(elementCount, writableSource);
    int tailResult = (elementCount % 2) ? source[elementCount - 1] : 0;
    array_view<int, 1> tailResultView(1, &tailResult);

    std::vector<int> result(1);
    computeTime = TimeFunc(view, [&]()
    {
        for (int stride = (elementCount / 2); stride > 0; stride /= 2)
        {
            parallel_for_each(view, extent<1>(stride), [=] (index<1> idx) restrict(amp)
```

```
            {
                av[idx] += av[idx + stride];

                // If there are an odd number of elements then the first thread
                // adds the last element.
                if ((idx[0] == 0) && (stride & 0x1) && (stride != 1))
                    tailResultView[idx] += av[stride - 1];
            });
        }

        //  Only copy out the first element in the array as this contains the final answer.
        copy(av.section(0, 1), result.begin());
        av.discard_data();
    });
    tailResultView.synchronize();
    return result[0] + tailResult;
}
```

Because the *IReduce* interface specifies that *Reduce()* takes a const vector reference, the *array_view* can't wrap the source vector; it must create a writable copy. Creating this additional copy of the data typically takes many times longer than the calculation time. In addition, the calculation time appears to be much longer in this implementation. It isn't; it's just that the automatic copy of data to the GPU by the *array_view* happens only when the calculation is underway, so it cannot be separated from the calculation time. That effect isn't really important unless it leads you to believe the *array_view* calculation is genuinely slower (it isn't), but the extra vector copy does have a major impact on the overall time the calculation takes.

You could easily make the *SimpleArrayViewReduction* implementation skip the vector copy by changing the *IReduce* interface so that the source vector could be changed. In a sample like this, changing the source vector would cause problems for subsequent implementations because the sample assumes that all the reductions are using the same unchanged source vector. In a real application, you might not need the source vector after accumulating the total. It's important to put thought into seemingly simple choices because of the performance implications they can carry.

Simple Optimized

Because adding is such a low-weight operation, it makes sense to "gang up" several additions for each thread, reducing the time spent accessing memory as well as the number of threads for which the *parallel_for_each* is executed. Consider a change in which each thread did something like this:

```
a[idx] += a[idx + offset]+ a[idx + offset + 1];
```

This code would incur only half the costs of writing to the array. A larger window to be handled by each thread would save even more time:

```
a[idx] += a[idx + offset]+ a[idx + offset + 1] + a[idx + offset + 2] + a[idx + offset + 3];
```

In the simple optimized version of this reduction, a *for* loop determines the sum of a window of elements and then adds that sum to the appropriate element. This lets you run the code with various window sizes to establish which one works well for your algorithm, your data, and your hardware. Test

runs on a variety of hardware available to the authors showed that a window size of 8 works well for this reduction, although a larger or smaller size might be optimal for another algorithm. Here's the code for the class:

```
class SimpleOptimizedReduction : public IReduce
{
public:
    int Reduce(accelerator_view& view, const std::vector<int>& source,
        double& computeTime) const
    {
        const int windowWidth = 8;
        int elementCount = static_cast<int>(source.size());

        // Using array as temporary memory.
        array<int, 1> a(elementCount, source.cbegin(), source.cend(), view);

        // Takes care of the sum of tail elements.
        int tailSum = 0;
        if ((elementCount % windowWidth) != 0 && elementCount > windowWidth)
            tailSum =
                std::accumulate(source.begin() + ((elementCount-1) / windowWidth) * windowWidth,
                    source.end(), 0);

        array_view<int, 1> avTailSum(1, &tailSum);

        // Each thread reduces windowWidth elements.
        int prevStride = elementCount;
        int result;
        computeTime = TimeFunc(view, [&]()
        {
            for (int stride = (elementCount / windowWidth); stride > 0; stride /= windowWidth)
            {
                parallel_for_each(view, extent<1>(stride), [=, &a] (index<1> idx) restrict(amp)
                {
                    int sum = 0;
                    for (int i = 0; i < windowWidth; i++)
                        sum += a[idx + i * stride];
                    a[idx] = sum;

                    // Reduce the tail in cases where the number of elements is not divisible.
                    // Note: execution of this section may negatively affect the performance.
                    // In production code the problem size passed to the reduction should
                    // be a power of the windowWidth.

                    if ((idx[0] == (stride-1)) && ((stride % windowWidth) != 0) &&
                        (stride > windowWidth))
                    {
                        for(int i = ((stride-1) / windowWidth) * windowWidth; i < stride; i++)
                            avTailSum[0] += a[i];
                    }
                });
                prevStride = stride;
            }

            // Perform any remaining reduction on the CPU.
            std::vector<int> partialResult(prevStride);
```

```
            copy(a.section(0, prevStride), partialResult.begin());
            avTailSum.synchronize();
            result = std::accumulate(partialResult.begin(), partialResult.end(), tailSum);
        });
        return result;
    }
};
```

This code is made more complicated by the work to deal with "tails." In the simple unoptimized code of the previous section, any tail was at most a single element. Now there is a tail before you start—the number of elements left over after dividing *elementCount* by *windowWidth*—and there could be one for each stride size as well. Furthermore, when the loop is finished, there are up to *windowWidth* partial sums in the first elements of the array, which need to be summed to produce a final result. (If you have complete control over the number of elements to be summed, you can eliminate the tail-handling code.) These three sets of leftovers are each handled differently.

The first tail is summed on the CPU before the *stride* loop even begins, chopping the source array down to a size that is a multiple of *windowWidth*. This sum is then wrapped in an *array_view* and each stride loop has a branch that selects a single thread to add up any tail elements in this iteration and accumulate them in the *array_view*'s single element. Finally, the partial sums are copied to a *std::vector* and added by using *tailSum* as a starting value. (Because the *array_view* is synchronized, *tailSum* includes all the tails added into the *array_view* as well.)

Experiments on a variety of hardware indicate that this optimization roughly halves execution time on many cards. On some NVIDIA cards, the execution time is reduced to one-quarter of the simple case, probably because of a different approach to scheduling work on those cards. This demonstrates the importance of measuring any changes that you make on real hardware. If, as with most applications, you need to support a wide variety of hardware, it's appropriate to make optimizations in your code that will improve performance on some configurations and never worsen it.

Naïvely Tiled

For most algorithms that are good fits for C++ AMP, tiling provides a substantial performance boost. It does so by enabling access to the programmable GPU cache with its dramatically faster memory access. However, to benefit from using this *tile_static* memory, either an algorithm must gain an important coalescing benefit when memory access is reordered by copying source data to a *tile_static* cache or it must use each value kept in *tile_static* memory more than once. The reduction algorithm in use here meets neither of those criteria as first written. It shouldn't be a surprise, then, that the tiled version of the reduction doesn't outperform the simple version. The code is presented here only because it lays the groundwork for some more careful optimizations that bring substantial benefit. Here is the starting point of the tiled algorithm:

```
template <int TileSize>
class TiledReduction : public IReduce
{
public:
    int Reduce(accelerator_view& view, const std::vector<int>& source,
        double& computeTime) const
```

```
    {
        int elementCount = static_cast<int>(source.size());

        // Copy data
        array<int, 1> arr(elementCount, source.cbegin(), source.cend(), view);

        int result;
        computeTime = TimeFunc(view, [&]()
        {
            while (elementCount >= TileSize)
            {
                extent<1> e(elementCount);
                array<int, 1> tmpArr(elementCount / TileSize);

                parallel_for_each(view, e.tile<TileSize>(),
                    [=, &arr, &tmpArr] (tiled_index<TileSize> tidx) restrict(amp)
                {
                    // For each tile do the reduction on the first thread of the tile.
                    // This isn't expected to be very efficient as all the other
                    // threads in the tile are idle.
                    if (tidx.local[0] == 0)
                    {
                        int tid = tidx.global[0];
                        int tempResult = arr[tid];
                        for (int i = 1; i < TileSize; ++i)
                            tempResult += arr[tid + i];
                        // Take the result from each tile and create a new array.
                        // This will be used in the next iteration. Use temporary
                        // array to avoid race condition.
                        tmpArr[tidx.tile[0]] = tempResult;
                    }
                });

                elementCount /= TileSize;
                std::swap(tmpArr, arr);
            }

            //  Copy the final results from each tile to the CPU and accumulate them there
            std::vector<int> partialResult(elementCount);
            copy(arr.section(0, elementCount), partialResult.begin());
            result = std::accumulate(partialResult.cbegin(), partialResult.cend(), 0);
        });
        return result;
    }
};
```

This implementation starts with a *parallel_for_each* that uses only one thread in each tile, accumulating a total similarly to the way the *windowWidth* totals were accumulated in the optimized simple version. Although this is easy to write, it means that all but one thread on each tile is doing nothing. After all elements for the tile have been summed, the results are copied into a new smaller array, *tmpArr*. The arrays are then swapped, and the entire process is repeated with the new smaller *elementCount*. There's no worry about tails during the computation, and the last version of the smaller array (which will contain at most *TileSize* elements) is copied back to the CPU and summed there.

This code is presented only as a starting point for the other tiled algorithms. On every hardware configuration where it was tested, it was significantly slower than the simple algorithm, often taking five or six times as long to run. Do not use this approach, with one active thread in a tile and the others idle, in code that actually requires performance improvements. You might have also have noticed that there is no *tile_static* memory use in this algorithm.

Tiled with Shared Memory

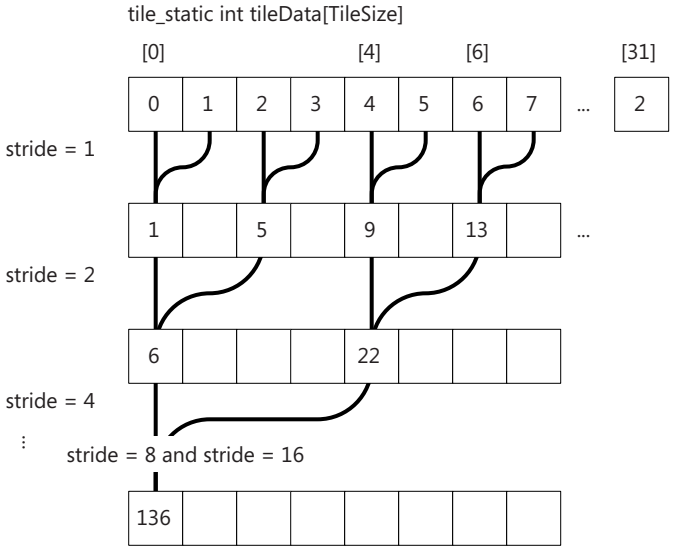
The tiled algorithm in the previous section has a number of temporary variables that are just declared in the usual way:

```
int tempResult = arr[tid];
```

Using *tile_static* memory here might improve performance if the memory can be accessed repeatedly. One way to do this is to copy a tile's worth of data to *tile_static* memory and perform the usual reduction on it. Because a *parallel_for_each* is already under way, it's not possible to loop the index, but it's possible to loop the stride and allow only certain threads to do the work. With the numbers to be accumulated in a C-style array of *tile_static* memory called *tileData*, that loop looks like this:

```
for (int stride = 1; stride < TileSize; stride *= 2)
{
    if (tid % (2 * stride) == 0)
        tileData[tid] += tileData[tid + stride];

    tidx.barrier.wait_with_tile_static_memory_fence();
}
```

Unlike the earlier algorithms, which started with a large stride and decreased it as the calculation progressed, this implementation starts with a small stride and increases it. For a tile size of 32, the calculations look like this:



On the first pass through this loop, every other thread does some work and the rest are idle. On the next pass, a quarter of the threads do work and three-quarters are idle. Each time stride is doubled, the number of working threads gets smaller and the divergence gets larger. (Divergence is first discussed in Chapter 7.) Finally, *tileData[0]* contains the sum for this tile, which can be written into global memory before moving to the next tile. After that's been done for every tile, as before, the *elementCount* is divided by *TileSize* to establish the new smaller size of the problem and the process is repeated. This version of the algorithm uses *array_view* instances to simplify the swapping of the original array and the smaller array representing the partial result each time the tiles have all done their work. The code overall looks like this:

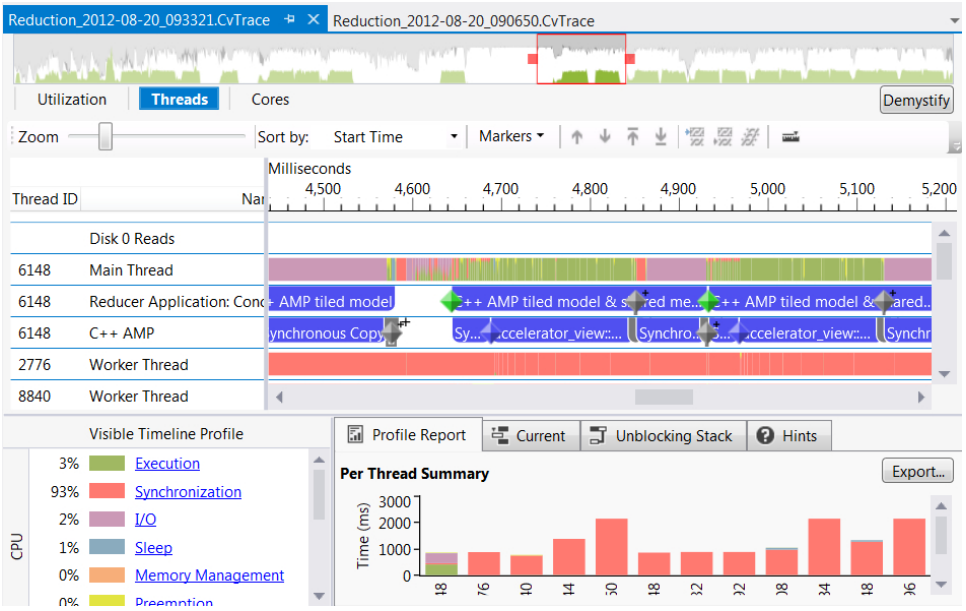
```
template <int TileSize>
class TiledSharedMemoryReduction : public IReduce
{
public:
    int Reduce(accelerator_view& view, const std::vector<int>& source,
        double& computeTime) const
    {
        int elementCount = static_cast<int>(source.size());
#ifdef MARKERS
        g_markerSeries.write_flag(diagnostic::normal_importance, L"Create array");
#endif
        array<int, 1> a(elementCount, source.cbegin(), source.cend(), view);
        array<int, 1> temp(elementCount / TileSize, view);
        array_view<int, 1> av(a);
        array_view<int, 1> tmpAv(temp);
        tmpAv.discard_data();

        int result;
        computeTime = TimeFunc(view, [&]()
        {
            while (elementCount >= TileSize)
            {
                extent<1> e(elementCount);
#ifdef MARKERS
                g_markerSeries.write_flag(diagnostic::normal_importance, L"Reduce");
#endif
                parallel_for_each(view, e.tile<TileSize>(),
                    [=] (tiled_index<TileSize> tidx) restrict(amp)
                {
                    //  Copy data onto tile static memory
                    int tid = tidx.local[0];
                    tile_static int tileData[TileSize];
                    tileData[tid] = av[tidx.global[0]];

                    //  Wait for all threads to finish copying
                    tidx.barrier.wait();

                    //  Reduce values for data on this tile
                    for (int stride = 1; stride < TileSize; stride *= 2)
                    {
                        // Highly divergent code! This will impact performance.
                        if (tid % (2 * stride) == 0)
                            tileData[tid] += tileData[tid + stride];
```

```
                    tidx.barrier.wait_with_tile_static_memory_fence();
                }

                //  Write the result for this tile back to global memory
                if (tid == 0)
                    tmpAv[tidx.tile[0]] = tileData[0];
            });

            elementCount /= TileSize;
            std::swap(tmpAv, av);
            tmpAv.discard_data();
        }

        //  Copy the final results from each tile to the CPU and accumulate them there.
        std::vector<int> partialResult(elementCount);
#ifdef MARKERS
        g_markerSeries.write_flag(diagnostic::normal_importance, L"Copy results");
#endif
                copy(av.section(0, elementCount), partialResult.begin());
        av.discard_data();
        result = std::accumulate(partialResult.cbegin(), partialResult.cend(), 0);
    });
    return result;
  }
};
```

This code includes three uses of the *g_markerSeries* instance to communicate with the concurrency visualizer. For details about how to make sure that the Reduction project has the Concurrency Visualizer enabled, see the "Concurrency Visualizer Markers" section in this chapter. To see the visualizer output, start a visualizer run by choosing Analyze, Concurrency Visualizer, Start With Current Project. Let the application run, and it will exit when it's complete. The Concurrency Visualizer will produce results something like this:

Chapter 5, "Tiled NBody Case Study," contains more details about using the Concurrency Visualizer and the symbols and colors used in each view.

Each green lump in the GPU Activity graph represents one of the C++ AMP reductions running. Recall that each reduction runs twice—once to warm up anything that might need to warm up and then again with a timer underway. You should see the CPU activity spike up during the PPL version of the reduction. The long gaps in CPU activity with GPU activity shown below are the two runs of the first tiled reduction, which allows only one thread in each tile to do any work. The impact of that decision is clearly visible here. The area of interest in this section is the two lumps immediately after that. Zoom to that area by clicking and dragging in the CPU Utilization graph and switching to the Threads view. You should see something like this:

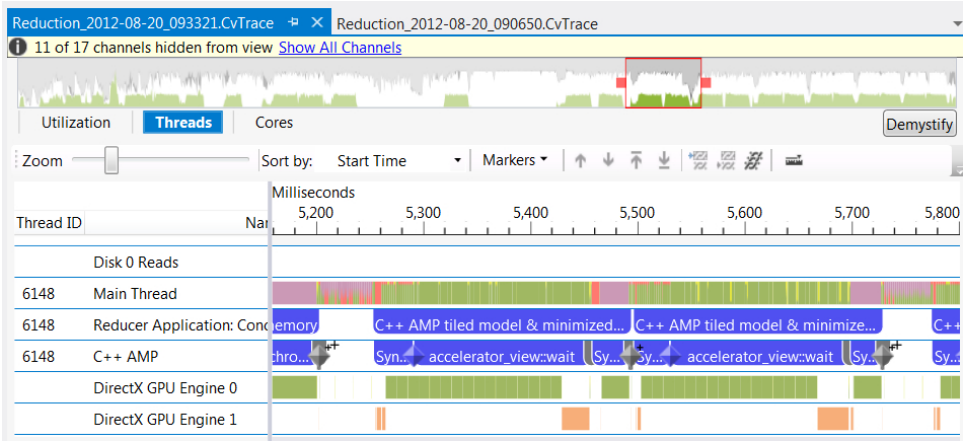


Three colors of markers are visible in this view in the rows, or channels, labeled Reducer Application and C++ AMP. The first marker in the Reducer Application row has the text Create array next to it. That's because of this line of code:

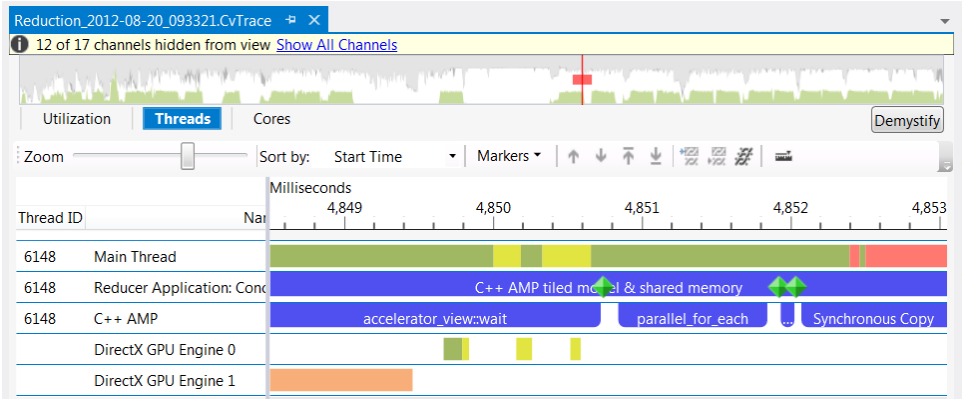
```
g_markerSeries.write_flag(diagnostic::normal_importance, L"Create array");
```

This doesn't give an indication of how long creating the array takes, only when it happens. The first instance of this marker and the gray marker (with a "+" that indicates several markers close together) are drawn on an empty white background. Shortly after that, in the Reducer Application row, is a blue span with the text (partly occluded by markers) "C++ AMP Tiled Model & Shared Memory." This span is created in the lambda expression passed to *JitAndTimeFunc()* in *main()*, and this is how you know that you are looking at the first run without the timer, followed by the second run with the timer. Zoom in on the second run only. As discussed in Chapter 5, you can hide rows representing worker threads that are not running your code but are running kernel32.dll, d3d11ref.dll, or dynamic-link libraries (DLLs) that start "ati" for AMD or "nv" for NVIDIA cards. Alternatively, you can move one

or more of the DirectX GPU Engine rows up so that they are near the Reducer Application row. You might then see something like this:



You can see that the marker labeled Synchronous Copy in the C++AMP channel does not actually correspond to any activity in the Memory Management color in the DirectX GPU Engine 1 row. This is why the timing process calls *wait()* on the accelerator view. The span describing that does line up with the memory management activity. There is a small amount of calculation after the copy and a cluster of three markers in the Reducer Application channel that match up with execution activity on the DirectX GPU Engine 0 row. Zooming in on that area will show the individual markers:



Clearly these three markers are very close together on the overall scale of the application's execution. Zooming in further shows what they represent.

Pausing the mouse over a marker will reveal its text; the first two have the text "Reduce" and the last one is "Copy Results." The first two come from this line at the top of the *while* loop:

```
g_markerSeries.write_flag(diagnostic::normal_importance, L"Reduce");
```

On the execution run used for this concurrency visualizer work, element count was 16 × 1024 × 1024 and *TileSize* was 512. The loop ran once with an element count of 16,777,216, and then again with an element count of 32,768. The third time the element count was only 64, so the loop did not execute. You can see all of this on the concurrency visualizer. Notice that the two *parallel_for_each* loops take a very similar amount of time to execute.

Using shared memory improves the performance of the reduction significantly compared to the one-thread-per-tile algorithm. The execution time is about 40 percent to 70 percent of the time taken by the naïve version. The divergence, however, is hurting performance. That's the next thing to work on.

Minimizing Divergence

The innermost loop in the tiled with shared memory algorithm adds up the values on the current tile:

```
for (int stride = 1; stride < TileSize; stride *= 2)
{
    if (tid % (2 * stride) == 0)
        tileData[tid] += tileData[tid + stride];

    tidx.barrier.wait_with_tile_static_memory_fence();
}
```

This code is divergent; some threads will add up a total and some will not. Here is a version of the loop that calculates an index from *tid* instead of seeing if *tid* meets a condition:

```
for (int stride = 1; stride < TileSize; stride *= 2)
{
    int index = 2 * stride * tid;
    if (index < TileSize)
        tileData[index] += tileData[index + stride];

    tidx.barrier.wait_with_tile_static_memory_fence();
}
```

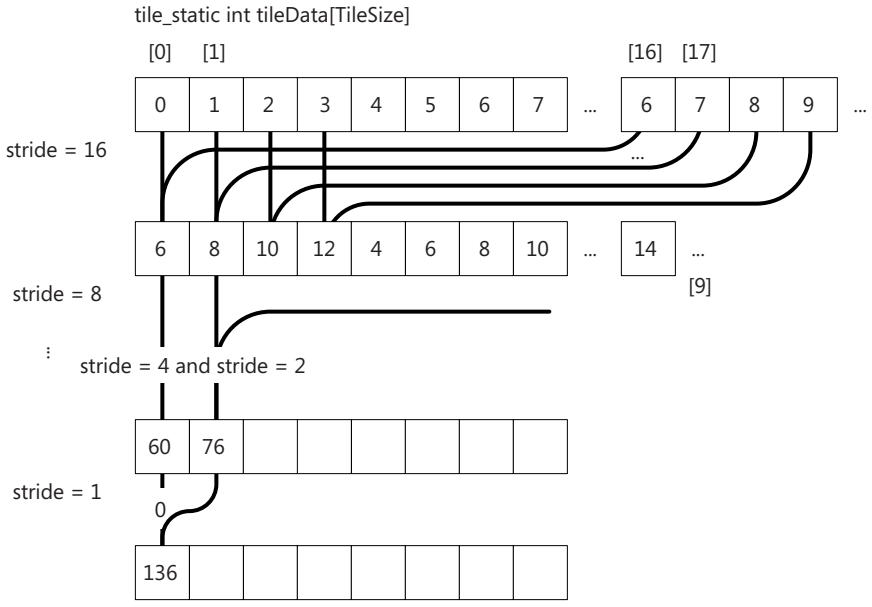
(All the other lines of *TiledMinimizedDivergenceReduction::Reduce()* are the same as *TiledSharedMemoryReduction::Reduce()* except that the former does not include the markers for the concurrency visualizer. In the interests of space, the whole class does not need to be shown here.)

In this version of the code, one consecutive group of threads will all calculate a value of *index* less than *TileSize*, and the rest of the threads in the tile, which also represent a consecutive group of threads, will calculate a value that is not less than *TileSize*. Each set of threads will either calculate a partial sum or will not. For example, when stride is 1 and *TileSize* is 512, threads 0 to 255 will calculate a sum and threads 256 to 511 will not. On typical hardware that was available while this book was being written, warp size is 32 or 64, meaning that four or eight warps will all be active together and four or eight warps will all be idle together. This is a big improvement over the previous version, in which every warp had a mixture of idle and active threads on every iteration of the outer loop. In tests on available hardware, this version of the reduction takes about 30 percent to 50 percent of the time that the naïve tiling algorithm took. The impact of divergence on execution time can be surprising.

After all, the total number of calculations is the same in both of these versions of the algorithm. Any time there is an *if* statement (or any kind of branching) in an algorithm, look for a way to keep consecutive threads doing the same thing for a big speed boost.

Eliminating Bank Conflicts

As mentioned in Chapter 7, when you are working with *tile_static* memory, either all the threads in a warp should access data in a single bank or each thread should access data in different banks. The section of Chapter 7 entitled "Efficient Tile Static Memory Access" illustrates (with a diagram) how to adapt the Reduction code to do away with bank conflicts. The loop starts with a large stride, and each thread accesses two elements that are in the same bank because they are in memory locations that are a multiple of 16 32-bit words apart. Each thread accesses a different bank from its neighbors.



If you compare this to the previous diagram, it's easy to see that now different threads are accessing elements in other banks. The inner loop looks like this:

```
for (int stride = (TileSize / 2); stride > 0; stride >>= 1)
{
    if (tid < stride)
        tileData[tid] += tileData[tid + stride];

    tidx.barrier.wait_with_tile_static_memory_fence();
}
```

(As before, the rest of the *TiledMinimizedDivergenceAndConflictsReduction* class is the same and is omitted to save space.)

On a variety of hardware, making this change lowers the execution time compared to the previous version by as much as 40 percent. The improvement is more dramatic on NVIDIA cards than on AMD cards, perhaps because the bank size is larger on the AMD cards used in testing. Another advantage of this version is that it makes obvious how many threads are idle as this iteration proceeds. In the diagram, a *TileSize* of 32, *stride* starts at 16 and on the first time through the loop half the threads are idle. The next time through, it's worse: *stride* is 8, and three-quarters of the threads are idle.

Reducing Stalled Threads

The innermost loop allows only half the threads in the tile to be active and leaves the other half idle. This is a result of the way the algorithm has been written; to add up 512 numbers in pairs requires only 256 additions. Then, adding those 256 numbers in pairs requires only 128 additions, and so on. Still, it seems a waste to have fully half the threads in the tile do no work other than copying data into *tile_static* memory.

If you look through the *Reduce()* implementations that use shared memory, you'll see that they all copy elements from the source collection to *tile_static* memory. The "Tiled with Shared Memory" section of this chapter showed how using *tile_static* memory lowers execution time. But there's no reason that the threads are restricted to only a simple copy into *tile_static* memory. If you set up an extent that is only half of the element count and have each thread add two elements (one from the first half and one from the corresponding position in the second half) and then store that in *tile_static* memory, you can now let the threads carry on with reducing the new, half-sized problem. Every thread will do at least one addition as part of the initial copy to *tile_static* even if it's idle after that during the innermost loop.

Here's how it looks in the *TiledMinimizedDivergenceAndConflictsReduction* implementation's code. This code works on the entire *elementCount* and has each thread in the tile copy one value to *tile_static* memory:

```
while (elementCount >= TileSize)
{
    extent<1> e(elementCount);

    parallel_for_each(view, e.tile<TileSize>(), [=] (tiled_index<TileSize> tidx) restrict(amp)
    {
        //  Copy data onto tile static memory
        int tid = tidx.local[0];
        tile_static int tileData[TileSize];
        tileData[tid] = av[tidx.global[0]];

        //  Wait for all threads to finish copying
        tidx.barrier.wait();
        // ...
```

This code in the *TiledMinimizedDivergenceConflictsAndStallingReduction* class works on a smaller extent and has each thread do a first add as well as writing to *tile_static* memory:

```
while (elementCount >= TileSize)
{
```

```
//  Tile extent is now halved.
extent<1> e(elementCount / 2);
assert((e.size() % TileSize) == 0);

parallel_for_each(view, e.tile<TileSize>(), [=] (tiled_index<TileSize> tidx) restrict(amp)
{
    // Instead of just loading from global memory perform the first reduction step during
    // load, so load becomes load two elements and store the result.
    int tid = tidx.local[0];
    tile_static int tileData[TileSize];
    // Partition input data among tiles, (2 * TileSize) because threads spawned is also
    // halved.
    int relIdx = tidx.tile[0] * (TileSize * 2) + tid;
    tileData[tid] = av[relIdx] + av[relIdx + TileSize];

    //  Wait for all threads to finish copying
    tidx.barrier.wait();
    // ...
```

The innermost loop doesn't change, but there are fewer tiles now because the overall extent is smaller. Not surprisingly, this version of reduction runs in just over half the time (50 percent to 60 percent) of the "no bank conflicts" version across a variety of hardware. After all, it's now dealing with half as much data in the inner loop and running half as many threads.

Loop Unrolling

To reduce stalled threads, the algorithm arranged for the first action that each thread takes to be more than just a copy and to have the thread start doing the work (in this case, adding values) before the loop. This is a form of unrolling. Loop unrolling as an optimization technique can backfire. Although it might seem like a great idea to save the time of evaluating a condition, your code might now be using registers differently, which might hurt performance. Alternatively, you might unroll a loop by hand and discover no performance improvement at all because that loop was being unrolled for you anyway! In CPU code, an optimizing compiler will unroll small loops whose bounds are known at compile time. On an accelerator, the JIT process that transforms your C++ AMP kernels into code that runs on the accelerator does include some loop unrolling and other optimizations. By getting in the way, you might make no difference or you might make things worse. There is no substitute for measuring to see what happens on your application's target hardware; this is not something you can reason about in advance with confidence.

Unrolling the innermost loop means transforming code like this:

```
for (int stride = (TileSize / 2); stride > 0; stride >>= 1)
{
    //  Remember that this is a branch within a loop and all threads will have to execute
    //  this but only threads with a tid < stride will do useful work.
    if (tid < stride)
        tileData[tid] += tileData[tid + stride];

    tidx.barrier.wait_with_tile_static_memory_fence();
}
```

into something that doesn't include a *for* loop. For example, if *TileSize* was 8 (a highly unrealistic value that makes this code simpler), the loop would be replaced with these lines:

```
if (tid < 4)
    tileData[tid] += tileData[tid + 4];
tidx.barrier.wait_with_tile_static_memory_fence();
if (tid < 2)
    tileData[tid] += tileData[tid + 2];
tidx.barrier.wait_with_tile_static_memory_fence();
if (tid < 1)
    tileData[tid] += tileData[tid + 1];
tidx.barrier.wait_with_tile_static_memory_fence();
```

It might seem at first that this loop can't be unrolled because you don't know *TileSize*, but it's a template parameter and is known at compile time. Any *if* statement involving *TileSize* can be evaluated at compile time, and the body of the *if* can then be included or excluded without any impact on run-time performance. Assuming an upper limit of 2,048 on tile size (and even a tile size of 1,024 produces a poorer performance than 512), you can use a series of such *if* statements to generate the unrolled loop:

```
if (TileSize >= 1024)
{
    if (tid < 512)
        tileData[tid] += tileData[tid + 512];
    tidx.barrier.wait_with_tile_static_memory_fence();
}
if (TileSize >= 512)
{
    if (tid < 256)
        tileData[tid] += tileData[tid + 256];
    tidx.barrier.wait_with_tile_static_memory_fence();
}
if (TileSize >= 256)
{
    if (tid < 128)
        tileData[tid] += tileData[tid + 128];
    tidx.barrier.wait_with_tile_static_memory_fence();
}
// . . .
```

Remember that this loop keeps adding numbers in the steadily shrinking "first part" of the source collection—the first half, then the first quarter, then the first eighth, and so on. Eventually this "first part" will be small enough to fit into a single warp.

Threads within a single warp are guaranteed to execute in lockstep. As a result, you no longer need a full-on *tile_barrier.wait_with_tile_static_memory_fence()* call, which is required to synchronize threads running on different warps. Instead, you can use just a memory fence. Because this has a lower overhead than using a barrier, it can result in faster execution. Although it's possible to remove the synchronization overhead of the barrier when running on a single warp, a memory fence is still required. A memory fence forbids the compiler or processor from rearranging reads and writes around the fence. It also prevents an optimizer from rearranging the code and introducing correctness bugs. It doesn't make all the threads wait, so it's a faster approach. Replacing a barrier with a

fence within a tile is correct only when all the active threads are within a single warp and are guaranteed to be running in lockstep.

When you use a fence, you can save even more effort by skipping the test of whether *tid* is in the first half of the collection or not. For example, the last two lines of the progression could look like this:

```
if (TileSize >= 4)
{
    if (tid < 2)
        tileData[tid] += tileData[tid + 2];
    tile_static_memory_fence(tidx.barrier);
}
if (TileSize >= 2)
{
    if (tid < 1)
        tileData[tid] += tileData[tid + 1];
    tile_static_memory_fence(tidx.barrier);
}
```

Consider a case in which *TileSize* is 4 or more (which is essentially every case): the *TileSize* conditions disappear at compile time and this code remains:

```
if (tid < 2)
    tileData[tid] += tileData[tid + 2];
tile_static_memory_fence(tidx.barrier);
if (tid < 1)
    tileData[tid] += tileData[tid + 1];
tile_static_memory_fence(tidx.barrier);
```

The thread with *tid* of 0 evaluates *tileData[0] += tileData[2]* and then *tileData[0] += tileData[1]*. The thread with *tid* of 1 evaluates *tileData[1] += tileData[3]* and then doesn't do any more work. When running within a warp, these threads are guaranteed to execute in lockstep. The memory fence is still required to prevent the operations being reordered.

What would happen if you did not have the last *if* statement? The thread with *tid* of 0 uses the value of *tileData[1]*, and if, on some runs, the thread with *tid* of 1 has (incorrectly) written the value of *tileData[1] + tileData[2]* to it, you would have a correctness bug. However, the presence of the fence allows you to remove this branching. The fence guarantees that the read and write operations on *tileData[1]* can't be reordered and will occur lockstep on each thread in the warp in the order shown in the code. Because the threads are in lockstep, the thread with *tid* of 0 and the thread with *tid* of 1 both read the value of *tileData[1]* before any writes to *tile_static* memory are allowed. After they've both read it, the thread with *tid* of 0 writes a correct sum to *tileData[0],* and the thread with *tid* of 1 writes a useless value to *tileData[1]*. At this point, the calculation is over and no code uses the value of *tileData[1]*, so there's no harm done by allowing the thread to write a value there. This means that the last two additions can be written like this:

```
tileData[tid] += tileData[tid + 2];
tile_static_memory_fence(tidx.barrier);
tileData[tid] += tileData[tid + 1];
tile_static_memory_fence(tidx.barrier);
```

This logic applies for all the threads with very small *tid* values. How small? The key is that data is read a warp at a time. For the threads whose IDs are all the same warp, the addition is quicker than checking to see whether the addition should be done (because the read has already happened). In this sample, the line is drawn at *tid* values below 32 under the assumption that warp size is 32. If it's actually 64, you could save a few more *tid* checks. If it's actually 16, you could be doing slightly expensive additions that a *tid* check would have spared you. You are encouraged to adjust the code to switch at 16 or 64 and run on a variety of hardware to see the effects of making the change. If the actual warp size is less than the value in your code, this change can actually introduce a correctness bug. The emulated accelerators, WARP and REF, have warp sizes of 1 and 4, respectively. WARP actually calculates a wrong answer with this unrolling. To prevent that, this version of the algorithm refuses to run on emulated accelerators:

```
if (accelerator(accelerator::default_accelerator).is_emulated)
    return -1;
```

If you use this sort of warp-size-specific optimization in your programs, it is critical that you only run on GPUs with an appropriate warp size. You can also provide an additional alternative warp-agnostic implementation. This unrolling improves performance by 30 percent to 60 percent from the "reduced stalling" version on a variety of hardware. Although you can't count on a benefit from loop unrolling, it can be significant.

Cascading Reductions

The continued transformation of the reduction algorithm has made it longer and longer, from the single-line calling *std::accumulate()* to the 179 lines of TiledMinimizedDivergenceConflictsAndStallingUnrolledReduction.h, and each change has kept the same basic structure while refining one part of the algorithm to reduce execution time. Across a variety of hardware, the tiled, divergence-minimized, bank-conflicts-minimized, stalled-threads-reduced, loop-unrolled version of the algorithm is faster (ignoring copy time to the accelerator) than the sequential CPU version, the parallelized CPU version, and the simple (not tiled) version. Considering that a simple arithmetic reduction is actually a poor fit for C++ AMP, this shows just how helpful these sorts of optimizations can be. You might well adopt a number of them in your own applications for similar benefit.

An alternative to optimizing a particular algorithm to reduce divergence, bank conflicts, stalled threads, and so on is to rearrange the algorithm entirely. Every one of the tiled algorithms has the same structure:

```
while (elementCount >= TileSize)
{
    extent<1> e(elementCount);
    parallel_for_each(e.tile<TileSize>(), [=, &a] (tiled_index<TileSize> tidx) restrict(amp)
    {
         // copy to tile_static memory
         // loop stride from 1 to TileSize doubling, or TileSize to 1 halving
         // add elements
    });
    elementCount /= TileSize;
    std::swap(tmpAv, av);
}
// finish on the CPU
```

This takes, for example, a collection of 512 × 512 elements, reduces that to 512 elements, and then, on the second pass through the *while* loop, reduces that to one final result. In all the tiled algorithms shown so far, the *parallel_for_each* is inside a *while* loop. It's possible to write the reduction differently, with the *parallel_for_each* as the outermost loop. It's an extension of the logic in the "stalled threads reduction" version of the algorithm.

To reduce stalled threads, the original line to load data into tile static memory

```
tileData[tid] = av[tidx.global[0]];
```

was changed to:

```
tileData[tid] = av[relIdx] + av[relIdx + TileSize];
```

This did one addition at the same time as the copy to *tile_static* memory and cut the problem space in half as a result. The cascading reduction runs a loop to build up a partial sum and then copies that partial sum to *tile_static* memory:

```
int tid = tidx.local[0];
tile_static int tileData[TileSize];
int i = (tidx.tile[0] * 2 * TileSize) + tid;
int stride = TileSize * 2 * TileCount;

//  Load and add many elements, rather than just two
int sum = 0;
do
{
    sum += a[i] + a[i + TileSize];
    i += stride;
}
while (i < elementCount);
tileData[tid] = sum;
```

To see how this works, consider an *elementCount* of 16 × 1,024 × 1,024, or 16,777,216, a *TileSize* of 512 and a *TileCount* (passed as a template parameter) of 128. (Notice that *TileSize* * 2 * *TileCount* is not equal to *elementCount*—*TileSize* represents how many threads are in a tile, not how many elements of the source collection are handled by a tile.) With these numbers, *stride* is 131,072.

For the thread with a local index of 0 in tile 0, this loop will add elements 0 and 512, then 131,072 and 131,584, and so on—128 times in all until *i* reaches *elementCount*. The partial sum that started from element 0 goes into element 0 of the *tile_static* array. Meanwhile, a thread with a local index of 1 in tile 1 adds 1 and 513, then 131,073 and 131,585, and so on. Each thread has added *2 * elementCount / stride* numbers to get its own partial sum, and there's no overlap—the last thread in the last tile uses element 131,071 and onward.

After waiting to ensure that all the threads in the tile have calculated a sum, this single tile now has a partial sum in each element of *tileData*. Between them, all the tiles will have accounted for every starting element after executing this code. That's why there's no need for an outer loop. All that's required is to sum up each tile into a single element and then finish the job on the CPU.

The code to sum up a whole tile into a single element will look very familiar:

```
for (stride = (TileSize / 2); stride > 0; stride >>= 1)
{
    //  Remember that this is a branch within a loop and all threads will have to execute
    //  this but only threads with a tid < stride will do useful work.
    if (tid < stride)
        tileData[tid] += tileData[tid + stride];

    tidx.barrier.wait_with_tile_static_memory_fence();
}

//  Write the result for this tile back to global memory
if (tid == 0)
    partial[tidx.tile[0]] = tileData[tid];
```

This has some of the same issues with stalled threads that earlier versions did, but they will be dealt with in the next section. In this section, the code is written for the best readability.

Finally, the numbers in *partial* are copied back to the CPU and added together:

```
std::vector<int> partialResult(TileCount);
copy(partial, partialResult.begin());
result = std::accumulate(partialResult.cbegin(), partialResult.cend(), 0);
```

On a variety of hardware, this entirely different algorithm beats the fastest of all the versions of the tiled algorithm shown in this chapter, taking 50 percent to 75 percent of the time.

Cascading Reductions with Loop Unrolling

Unrolling the innermost loop, just as the other tiled calculations did, is a good example of an ambivalent optimization. The best it could do was to reduce execution time by 23 percent for one rather slow card. All the other cards saw speeds that were reduced by 8 percent, by only 2 percent, or that actually took longer. (The actual code is not shown here in the interests of space; the innermost loop in *CascadingUnrolledReduction::Reduce()* is identical to the one in *TiledMinimizedDivergenceConflictsAndStallingUnrolledReduction::Reduce()*, and the remainder of the algorithm is the same as in the previous section, a cascading reduction.) As in the previous case, the algorithm will not run on emulated accelerators because their tiny warp size introduces correctness errors when unrolling.

Because this reimplementation makes the code longer and introduces the possibility of calculation errors and also makes changes more difficult, it probably shouldn't be used for this particular algorithm. As shown earlier, loop unrolling was helpful for the other version of the reduction algorithm. Predicting whether loop unrolling will help is effectively impossible—you must measure to know whether or not to do it.

Summary

Improving the performance of even a very simple algorithm can be a lot of work. You need to understand the algorithm itself and how it executes on the GPU quite well; for example, how many times does it use each piece of input data? You can't avoid learning a little about the architecture of a GPU and of other accelerators on which your code might run. And there's no substitute for measuring and testing all the optimizations you undertake.

You might discover that a particular change makes a big improvement on one set of cards (those from one vendor, more expensive ones, or ones with more memory) and little or no improvement on others (from other vendors, less expensive, and so on.) The issue is not so much hardware vendor (although scheduling and other architectural decisions might play a role) as it is warp or wavefront size. Some of these implementations work best for certain sizes and not as well for others. If your testing shows that an optimization produces an improvement on both AMD and NVIDIA hardware, you can use it with confidence. If it shows an improvement on one and has no effect on another, try changing any tweakable numbers to see if you can reverse the effect. You might need to leave the tweakable numbers at a value that improves performance for the hardware you think is most likely to be used with your software. In the worst case, you might find that a change improves performance on one set of cards and worsens it on another. If adjusting the tweakable numbers can't change this for your algorithm, it would be best to forgo the optimization altogether. You should write less readable code that's harder to maintain and test only when you're sure it carries a benefit.

To make your code run at 10 or 15 times the speed of the shortest and simplest implementation, you might need to write 10 or 15 times as many lines of code—or more—and it can be difficult to know whether writing and testing the extra code will carry any benefit. Seeing some particular optimizations applied to the same problem is one way to gauge the possibility that the optimization will be useful for you. Be prepared to iterate and explore and to rethink both your algorithm and your data structures to achieve the maximum possible speedup from C++ AMP.

CHAPTER 9

Working with Multiple Accelerators

In this chapter:

Choosing Accelerators 203
Using More than One GPU 208
Swapping Data among Accelerators 211
Dynamic Load Balancing 216
Braided Parallelism 219
Falling Back to the CPU 220
Summary 222

So far, the examples in the book have covered using C++ AMP with a single accelerator: a single physical GPU, a WARP accelerator, or the reference (REF) accelerator. Each of these accelerator types is described in this chapter. Although using a single accelerator is probably the most common scenario today, the computer running your application might soon have more than one accelerator. This could be a combination of one or more discrete GPUs, a GPU integrated with the CPU, or both. If your application wants to use all of the available compute power, it needs to efficiently orchestrate executing parts of the work on each accelerator and combining the results to give the final answer. This chapter shows how to choose among different C++ AMP accelerators and select the best ones for your code. It also covers running C++ AMP on more than one accelerator and using Parallel Patterns Library (PPL) code running on the CPU to orchestrate the GPU accelerators or execute work more suited to the CPU. These strategies will maximize your application's performance.

Choosing Accelerators

C++ AMP allows you to enumerate the available accelerators and choose the ones on which your application will run. Your application can also filter the accelerators based on their properties and select a default accelerator.

Enumerating Accelerators

The following code uses *accelerator::get_all()* to enumerate all the available accelerators and print their device paths and descriptions to the console:

```
std::vector<accelerator> accls = accelerator::get_all();

std::wcout << "Found " << accls.size() << " C++ AMP accelerator(s):" << std::endl;
std::for_each(accls.cbegin(), accls.cend(), [](const accelerator& a)
{
    std::wcout << "  " << a.device_path << std::endl
        << "    " << a.description << std::endl << std::endl;
});
```

The *description* property provides a user-friendly name for the accelerator, but the *device_path* property provides a persistent unique identifier that is more useful for programmatically selecting accelerators. The device path is also persistent across processes and Microsoft Windows–based sessions, provided the hardware isn't changed or the system reinstalled. For example, your application can use the *device_path* to refer to an accelerator selected by the user in a previous application session.

You can run this example by loading the Chapter9\Chapter9.sln solution. Build the sample in Release configuration and run it using Ctrl+F5 to start it without the debugger attached. Here's some example output from this code:

```
Using device : NVIDIA GeForce GTX 570

Enumerating accelerators

Found 4 C++ AMP accelerator(s):
  PCI\VEN_10DE&DEV_1081&SUBSYS_15703842&REV_A1\4&2EB3824&0&0018
    NVIDIA GeForce GTX 570

  PCI\VEN_10DE&DEV_1081&SUBSYS_15703842&REV_A1\4&2276C4A6&0&0038
    NVIDIA GeForce GTX 570

  direct3d\warp
    Microsoft Basic Render Driver

  direct3d\ref
    Software Adapter

  cpu
    CPU accelerator

Found 2 C++ AMP hardware accelerator(s):
  PCI\VEN_10DE&DEV_1081&SUBSYS_15703842&REV_A1\4&2EB3824&0&0018
  PCI\VEN_10DE&DEV_1081&SUBSYS_15703842&REV_A1\4&2276C4A6&0&0038
Has WARP accelerator: true
```

```
Looking for accelerator with display and 1MB of dedicated memory...
  Suitable accelerator found.

Setting default accelerator to one with display and 1MB of dedicated memory..
  Default accelerator is now: NVIDIA GeForce GTX 570
```

The list shows all the available C++ AMP accelerators. In this example, it shows the following accelerators:

- Two GPUs, each with unique device paths and a description containing the GPU's friendly name.
- The WARP accelerator with description "Microsoft Basic Render Driver."
- The reference, or REF accelerator, also referred to as the "Software Adapter."
- The CPU accelerator.

Your application can select a device using one of the following device path names that are predefined as static properties on the C++ AMP *accelerator* class:

- **accelerator::direct3d_ref** The REF accelerator, also called the Reference Rasterizer or "Software Adapter" accelerator. It emulates a generic graphics card in software on the CPU to provide Direct3D functionality. It is used for debugging and will also be the default accelerator if no other accelerators are available. As the name suggests, the REF accelerator should be considered the de facto standard if you suspect a bug with your hardware vendor's driver. Typically, your application will not want to use the REF accelerator because it is much slower than hardware-based accelerators and will be slower than just running a C++ implementation of your algorithm on the CPU.
- **accelerator::cpu_accelerator** The CPU accelerator can be used only for creating arrays that are accessible to the CPU and used for data staging. Your application can't use this for executing C++ AMP code in the first release of C++ AMP. Further details on using the CPU accelerator to create staging arrays and host arrays are covered in Chapter 7, "Optimization."
- **accelerator::direct3d_warp** The WARP accelerator, or Microsoft Basic Render Driver, allows the C++ AMP run time to run on the CPU. The WARP accelerator uses the WARP software rasterizer, which is part of the Direct3D 11 run time. The WARP accelerator uses multicore and data-parallel Single Instruction Multiple Data (SIMD) instructions to execute data-parallel code very efficiently on the CPU. Your application can use WARP as a fallback when no physical GPU is present. The WARP accelerator supports only single-precision math, so it can't be used for fallback for kernels that require double precision or limited double-precision kernels. An overview of WARP can be found in "Windows Advanced Rasterization Platform (WARP) Guide" on MSDN: *http://msdn.microsoft.com/en-us/library/gg615082.aspx*.
- **accelerator::default_accelerator** The current default accelerator. See the next section for more information on the default accelerator.

Note that although the WARP accelerator runs directly on the CPU, it is also considered to be an emulated accelerator. The *accelerator::is_emulated* property is true for both the REF and WARP accelerators.

You can filter out accelerators by examining each accelerator's properties, as shown in the following code:

```
std::vector<accelerator> accls = accelerator::get_all();
accls.erase(std::remove_if(accls.begin(), accls.end(), [](accelerator& a)
    {
        return a.is_emulated;
    }), accls.end());
std::wcout << "Found " << accls.size() << " C++ AMP hardware accelerator(s):" << std::endl;
```

Now *accls* contains only the GPU accelerators available. Similarly, accelerator device paths can be used to test for the presence of a particular type of accelerator. For example, your application might check for the presence of a WARP accelerator and give the user an option to fall back on this if no C++ AMP-capable GPUs are present.

```
std::vector<accelerator> accls = accelerator::get_all();
bool hasWarp = std::find_if(accls.begin(), accls.end(), [=](accelerator& a)
    {
        return a.device_path.compare(accelerator::direct3d_warp) == 0;
    }) != accls.end();
std::wcout << "Has WARP accelerator: " << (hasWarp ? "true" : "false") << std::endl;
```

The *accelerator* class also provides properties to query various attributes of an accelerator: the amount of dedicated memory, whether a display is attached, double-precision support, version number, and whether a debug layer is enabled. For example, the following code searches for a GPU accelerator with at least 2 MB of memory, limited double-precision support, and a connected display:

```
std::vector<accelerator> accls = accelerator::get_all();
bool found = std::find_if(accls.begin(), accls.end(), [=](accelerator& a)
    {
        return !a.is_emulated && a.dedicated_memory >= 2048 &&
            a.supports_limited_double_precision && a.has_display;
    }) != accls.end();
std::wcout << "Suitable accelerator " << (found ? "found." : "not found.") << std::endl;
```

See Chapter 12, "Tips, Tricks, and Best Practices," for further discussion of double, limited-double, and single-precision support. See the "accelerator Class" topic on MSDN for further details about the properties and methods on *accelerator* for filtering: *http://msdn.microsoft.com/en-us/library/hh350895*.

The Default Accelerator

The C++ AMP run time selects the default accelerator according to the following rules. If the application is being debugged under the GPU debugger, then the default accelerator is specified by the project properties setting (see Chapter 6, "Debugging"). When the application is not launched in debug mode, the *CPPAMP_DEFAULT_ACCELERATOR* environment variable, if defined, is used to determine the default accelerator. Otherwise, the default will be set to the nonemulated accelerator with the

largest amount of dedicated memory. When more than one such accelerator has the same amount of dedicated memory, the first accelerator without a display is chosen. This is an implementation detail and might change in subsequent releases. Regardless of the implementation specifics, the C++ AMP run time will always try to pick the best accelerator as the default.

The C++ AMP run time sets the default accelerator when your code asks for it, either with an explicit call or by creating an array. The default accelerator is also set by a call to *parallel_for_each* that does not either explicitly specify an *accelerator_view* or capture an *array* or *texture* that would implicitly specify one. Before that point in your code, you can set the default accelerator yourself, using the *accelerator::set_default()* method. Calls to *set_default()* after the run time has already set a default will return *false*, indicating that the call failed to change the default accelerator. The following example sets the default accelerator to a GPU with 1 MB of memory and a connected display:

```
std::vector<accelerator> accls = accelerator::get_all();
std::vector<accelerator>::iterator usefulAccls = std::find_if(accls.begin(), accls.end(),
    [=](accelerator& a)
    {
        return !a.is_emulated && (a.dedicated_memory >= 1024) && a.has_display;
    });
if (usefulAccls != accls.end())
{
    accelerator::set_default(usefulAccls->device_path);
    std::wcout << "  Default accelerator is now "
        << accelerator(accelerator::default_accelerator).description << std::endl;
}
else
    std::wcout << "  No suitable accelerator available" << std::endl;
```

As discussed in Chapter 3, "C++ AMP Fundamentals," all C++ AMP kernels run on an *accelerator_view*. An *accelerator_view* represents a logical, isolated view on a particular accelerator. If no *accelerator_view* is specified, an *accelerator_view* on the default accelerator is used. You can specify which accelerator to use by passing an *accelerator_view* associated with a particular accelerator to the invocation of *parallel_for_each* or by capturing an *array* or *texture* stored on the desired accelerator. In this example, *accls* is a *std::vector* containing two or more *accelerator* instances. The default accelerator is set to *accls[0],* but the *array*, *onData1*, is initialized with an additional *accelerator_view* parameter associating it with *accls[1].default_view*. The following kernel runs on *accls[1]* even though *accls[0]* is the default accelerator because the *parallel_for_each* captures *dataOn1*, an *array* associated with the default *accelerator_view* of *accls[1]*:

```
accelerator::set_default(accls[0].device_path);    // Accelerator 0 is now the default
array<int> dataOn1(10000, accls[1].default_view);

parallel_for_each(dataOn1.extent, [&dataOn1](index<1> idx) restrict(amp)
{
    dataOn1[idx] = // ...
});
```

If your kernel uses *array_view* rather than array, the *accelerator_view* must be passed as an additional parameter to the *parallel_for_each*. Again, the following kernel executes on *accls[1]*:

```
std::vector<int> dataOnCpu(10000, 1);
array_view<int, 1> dataView(1, dataOnCpu);

parallel_for_each(accls[1].default_view,
    dataView.extent, [dataView](index<1> idx) restrict(amp)
{
    dataView[idx] = // ...
});
```

Attempting to execute a kernel on one accelerator that contains references to an *array* stored on a different accelerator will result in a *concurrency::runtime_exception*. If the kernel references an *array_view* that wraps data stored on a different accelerator, the data will be implicitly copied onto the accelerator specified by the *parallel_for_each* invocation.

Using More Than One GPU

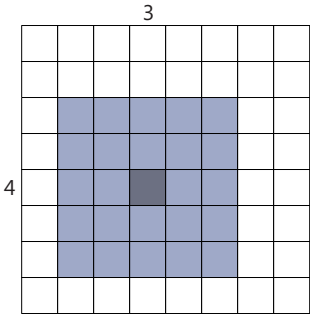
If your application detects more than one C++ AMP-capable GPU accelerator, the question becomes: How can you take advantage of this? The answer is to schedule work on all accelerators concurrently, allocating a portion of the total work to each accelerator, and finally to combine the results. This approach is often called the scatter-gather or master-worker pattern. The CPU divides the work and scatters it among the available workers. Workers complete their portion of the work and the result is gathered back up to the CPU master, which then either uses the final result or scatters more work to the GPU workers.

The following example calculates the weighted average of the elements in a matrix using a single *parallel_for_each* to execute the computation on the default C++ AMP accelerator. Each thread on the GPU calculates the weighted average of an element in matrix C from the corresponding elements in matrix A using a weighting function, *WeightedAverage()*.

```
const int rows = 2000, cols = 2000; shift = 60;
std::vector<float> vA(rows * cols);
std::vector<float> vC(rows * cols);
std::iota(vA.begin(), vA.end(), 0.0f);

array_view<const float, 2> a(rows, cols, vA);
array_view<float, 2> c(rows, cols, vC);
c.discard_data();

extent<2> ext(rows - shift * 2, cols - shift * 2);
parallel_for_each(ext, [=](index<2> idx) restrict(amp)
{
    index<2> idc(idx[0] + shift, idx[1] + shift);
    c[idc] = WeightedAverage(idc, a, shift);
});
c.synchronize();
```

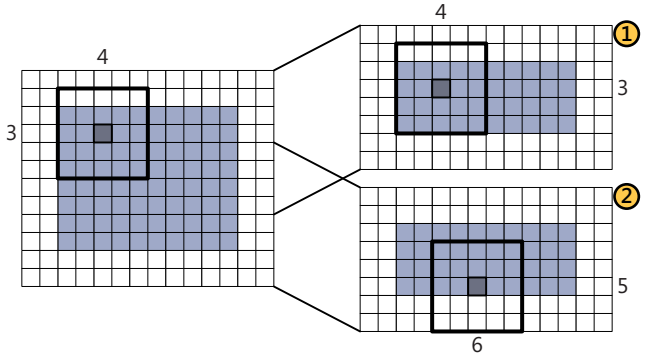
The *WeightedAverage()* function simply calculates an average using the weighted sum of the surrounding pixels, and the parameter *shift* specifies the size of the surrounding pixel window. This actual function isn't that important; for the purposes of the example, it is just work being done on the

GPU that depends on surrounding values in the matrix. Although this is a trivial example, it serves to demonstrate some of the complexities when partitioning a computation across more than one GPU. The Cartoonizer case study in Chapter 10, "Cartoonizer Case Study," shows an example of a much more computationally intensive application that uses a similar algorithm.



In the diagram, matrix element [4, 3] is calculated based on the 24 surrounding elements with a *shift* parameter of 2 for a 8×8 matrix. Even on a single accelerator new values can be calculated only for matrix elements sufficiently far away from the edge of the matrix that a full window can be used to calculate the average. These elements are represented by the 8×10 shaded area on the next diagram. The border around the edge is called the halo; it holds read-only values that are required to correctly calculate new values for elements that lie inside the halo.

It's possible to divide the work across several accelerators by creating *array_view* instances corresponding to subregions of the matrices and executing these on different accelerators. In this case, each accelerator must be passed to not only the elements for which it will calculate new values but also the halo elements. This increases the amount of data being transferred. For large arrays, where the halo width is much smaller than the overall matrix dimensions, this does not present a significant additional overhead. The following diagram shows the partitioning of the matrix onto two accelerators. Note that each accelerator is allocated a half of the computable matrix, a 4×10 region, and the halo elements needed to calculate the result. Now the accelerators can work in parallel to calculate the weighted sum of the respective portions of the matrix allocated to them.



A *TaskData* structure is used to track the work assigned to each C++ AMP accelerator. It stores the default *accelerator_view* for each accelerator and the start row and read and write extents of the

submatrices that the accelerator will use for its part of the overall calculation. The *writeExt* holds the dimensions of the shaded rows, and *writeOffset* holds the number of offset rows to the top of the shaded areas.

```
struct TaskData
{
    int id;
    accelerator_view view;
    int startRow;
    extent<2> readExt;
    int writeOffset;
    extent<2> writeExt;

    TaskData(accelerator a, int i) : view(a.default_view), id(i) {}
    // ...
};
```

The *TaskData* structures are initialized to divide up the rows of the matrix between the available accelerators. The *TaskData* struct defines a static method to do this.

```
    static std::vector<TaskData> Configure(const std::vector<accelerator>& accls,
        int rows, int cols, int shift)
    {
        std::vector<TaskData> tasks;
        int startRow = 0;
        int rowsPerTask = int(rows / accls.size());
        int i = 0;
        std::for_each(accls.cbegin(), accls.cend(),
            [=, &tasks, &i, &startRow](const accelerator& a)
        {
            TaskData t(a, i++);
            t.startRow = std::max(0, startRow - shift);
            int endRow = std::min(startRow + rowsPerTask + shift, rows);
            t.readExt = extent<2>(endRow - t.startRow, cols);
            t.writeOffset = shift;
            t.writeExt = extent<2>(t.readExt[0] - shift -
                ((endRow == rows || startRow == 0) ? shift : 0), cols);
            tasks.push_back(t);
            startRow += rowsPerTask;
        });
        return tasks;
    }
```

Your application can then create an *array_view* for the subregion of matrices of A and C and execute a C++ AMP kernel on each accelerator to calculate the values for the corresponding subregion of matrix C.

```
    const int rows = 2000, cols = 2000; shift = 60;
    std::vector<TaskData> tasks = TaskData::Configure(accls, rows, cols, shift);

    std::vector<float> vA(rows * cols);
    std::vector<float> vC(rows * cols);
    std::iota(vA.begin(), vA.end(), 0.0f);

    std::for_each(tasks.cbegin(), tasks.cend(), [&avCs](const TaskData& t)
```

```
{
    avCs.push_back(array<float, 2>(t.readExt, t.view));
});

std::for_each(tasks.cbegin(), tasks.cend(), [=](const TaskData& t)
{
    array_view<const float, 2> a(t.readExt, &vA[t.startRow * cols]);
    array_view<float, 2> c = avCs[t.id];
    index<2> writeOffset(t.writeOffset, shift);
    parallel_for_each(t.view, t.writeExt, [=](index<2> idx) restrict(amp)
    {
        index<2> idc = idx + writeOffset;
        c[idc] = WeightedAverage(idc, a, shift);
    });
});

std::for_each(tasks.cbegin(), tasks.cend(), [=, &vC](const TaskData& t)
{
    array_view<float, 2> outData(t.writeExt, &vC[(t.startRow + t.writeOffset) * cols]);
    avCs[t.id].section(index<2>(t.writeOffset, 0), t.writeExt).copy_to(outData);
});
```

This example uses a *std::for_each* to launch a kernel on each GPU and then a second loop to synchronize the results back to the CPU. The full implementation is in the *MatrixMultiGpuSequentialExample* function in Main.cpp.

If you run the sample on a machine with more than one C++ AMP-capable GPU, you will see output similar to the following. The exact times will vary based on the GPUs being used, as well as other factors, such as the type of CPU and the speed of the PCI bus and RAM on your computer.

```
Matrix weighted average 2000 x 2000 matrix, with 121 x 121 window
Matrix size 15625 KB

Single GPU matrix weighted average took                          1198.91 (ms)
2 GPU matrix weighted average (p_f_e) took                        649.923 (ms)
2 GPU matrix weighted average took                                652.042 (ms)
```

The matrix weighted average on two GPUs is faster, showing an improvement of 84 percent. This is not 100 percent because there is some overhead associated with distributing the calculation across two GPUs.

This is a small sample designed to make the code easier to read, but it doesn't represent the sort of real workloads that will be able to take full advantage of more than one GPU and the CPU. The NBody and Cartoonizer case studies can both be run on multiple GPUs.

Swapping Data among Accelerators

The weighted average example does not share any data between accelerators during the calculation. The result for each matrix element depends on the surrounding elements, but each accelerator contains a halo of additional read-only elements. Iterative calculations that rely on neighboring

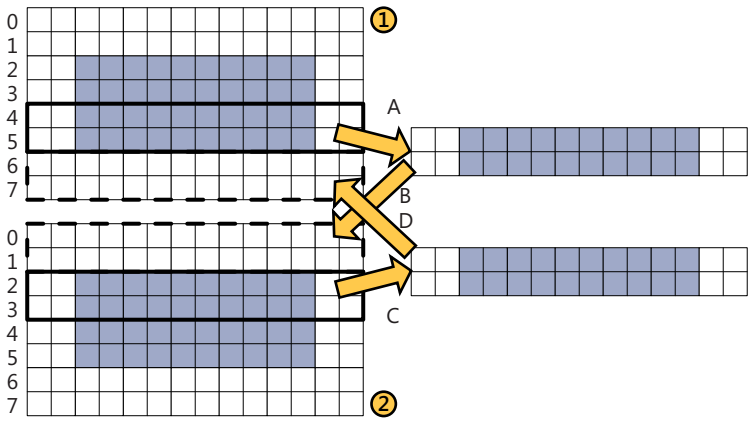
elements stored on other GPUs will need to refresh updated elements before the next iteration step can proceed.

When using multiple GPUs, it's often necessary to swap some or all of the data between steps in a calculation. Typically, the process of a calculation step looks like this:

1. Divide current data among different GPUs.
2. Calculate results on each GPU based on its local data.
3. Swap some or all result data among GPUs by copying it to CPU memory and then back to the other GPUs.
4. Go to step 2.

Depending on your application, you might need to share some or all of the result data after each calculation step. For example, the multi-GPU implementation in the NBody case study (in Chapter 2, "NBody Case Study") shares all the data after each time step. In contrast, the Cartoonizer case study has only to share the edges of each subregion of the image being processed (see Chapter 10). In either case, there must be sufficient computation at each step to outweigh the cost of the additional data transfers.

Let's consider a modified version of the original multi-GPU weighted average code that repeats the weighting calculation 10 times. This version of the code is defined in the *LoopedMatrixMultiGpu()* function in Main.cpp. Implementing the iterative algorithm efficiently on two accelerators requires swapping additional data, as illustrated in the following diagram:



At the end of each loop iteration, the new results from rows 4 and 5 on accelerator 1 must be copied into the halo cells in rows 0 and 1 of accelerator 2. Similarly, the new values from accelerator 2 must be copied into the halo of accelerator 1. No further calculation can take place while this is happening. The larger the averaging window, the more data will be transferred after each step of the calculation.

In this iterative example, the data is broken up and stored in separate *array* instances on each accelerator. The *std::vector* instances *arrAs* and *arrCs* store these *array* instances for each accelerator.

```
const int rows = 2000, cols = 2000; shift = 60;
std::vector<TaskData> tasks = TaskData::Configure(accls, rows, cols, shift);

std::vector<float> vA(rows * cols);
std::vector<float> vC(rows * cols);
std::iota(vA.begin(), vA.end(), 0.0f);

std::vector<array<float, 2>> arrAs;
std::vector<array<float, 2>> arrCs;

std::for_each(tasks.begin(), tasks.end(), [&](const TaskData& t)
{
    arrAs.push_back(array<float, 2>(t.readExt, &vA[t.startRow * cols], t.view));
    arrCs.push_back(array<float, 2>(t.readExt, t.view));
});
```

Two additional arrays on the CPU are used to swap the data among the *array* instances stored on each GPU.

```
array<float, 2> swapTop = array<float, 2>(extent<2>(shift, cols),
    accelerator(accelerator::cpu_accelerator).default_view);
array_view<float, 2> swapViewTop = array_view<float, 2>(swapTop);
array<float, 2> swapBottom = array<float, 2>(extent<2>(shift, cols),
    accelerator(accelerator::cpu_accelerator).default_view);
array_view<float, 2> swapViewBottom = array_view<float, 2>(swapBottom);
```

The new multiaccelerator code is shown below. The full source is in the *LoopedMatrixMultiGpuExample()* function in Main.cpp. During each loop iteration, it calculates the updates to the submatrices on each accelerator and then swaps just the upper and lower edges to update the halo elements of each matrix. Finally, it swaps the *vector* containing the results for each accelerator, *arrCs*, for the vector containing next inputs, *arrAs*.

```
for (int i = 0 ; i < iter; ++i)
{
    //  Calculate a portion of the result on each GPU

    std::for_each(tasks.cbegin(), tasks.cend(), [=, &arrAs, &arrCs, &vC](const TaskData& t)
    {
        array<float, 2>& a = arrAs[t.id];
        array<float, 2>& c = arrCs[t.id];

        parallel_for_each(t.view, t.readExt, [=, &a, &c](index<2> idx) restrict(amp)
        {
            c[idx] = a[idx];
            if ((idx[0] >= shift) && (idx[0] < (rows - shift)) &&
                (idx[1] >= shift) && (idx[1] < (cols - shift)))
                c[idx] = WeightedAverage(idx, a, shift);
        });
    });

    //  Swap edges

    std::vector<completion_future> copyResults((tasks.size() - 1) * 2);
    parallel_for(0, int(tasks.size() - 1), [=, &arrCs, &copyResults](size_t i)
    {
```

```
        array_view<float, 2> bottomEdge =
            arrCs[i + 1].section(index<2>(tasks[i + 1].writeOffset, 0),
                swapViewBottom.extent);
        array_view<float, 2> bottomEdge =
            arrCs[i + 1].section(index<2>(tasks[i +1].writeOffset,0),
                swapViewBottom.extent);
        copyResults[i] = copy_async(topEdge, swapViewTop);
        copyResults[i + 1] = copy_async(bottomEdge, swapViewBottom);
    });

    parallel_for_each(copyResults.begin(), copyResults.end(), [=](completion_future& f)
    { f.get(); });

    parallel_for(0, int(tasks.size() - 1), [=, &arrCs, &copyResults](size_t i)
    {
        array_view<float, 2> topEdge =
            arrCs[i].section(index<2>(tasks[i].writeOffset + tasks[i].writeExt[0] - shift, 0),
                swapViewTop.extent);
        array_view<float, 2> bottomEdge = arrCs[i + 1].section(swapViewTop.extent);
        copyResults[i] = copy_async(swapViewTop, bottomEdge);
        copyResults[i + 1] = copy_async(swapViewBottom, topEdge);
    });

    parallel_for_each(copyResults.begin(), copyResults.end(), [=](completion_future& f)
    { f.get(); });

    //  Swap results of this iteration with the input matrix
    std::swap(arrAs, arrCs);
}
```

The swapping steps use *copy_async()* rather than *copy()* to minimize the impact of any copy operations by parallelizing them as much as possible. The copying takes place in two phases: first, the edges are copied into CPU memory (operations A and C in the previous diagram), and then they are copied back to the other GPU (operations B and D in the diagram). Each copy operation returns a *completion_future*. After all the copy operations have been started, the program uses *completion_future::get()* to wait until all the copy operations have finished before starting the next phase.

The following example also uses *copy_async()* rather than *copy()* to move data from the CPU memory to the accelerator.

```
    const int size = 1024 * 1024;
    std::vector<float> vA(size, 0.0f);
    array<float, 1> arrA(size);

    std::cout << "Data copy of " << size << " bytes starting." << std::endl;
    completion_future f = copy_async(vA.cbegin(), vA.cend(), arrA);
    f.then([=] ()
    {
        std::cout << "  Finished asynchronous copy!" << std::endl;
    });
    std::cout << "Do more work on this thread..." << std::endl;
    f.get();
    std::cout << "Data copy completed." << std::endl;
```

The output from this code looks like this. The output from the main application thread "Do more work on this thread..." is displayed before the output from the *completion_task::then* function, "Finished asynchronous copy," which executes after the copy is complete.

```
Data copy of 1048576 bytes starting.
Do more work on this thread...
  Finished asynchronous copy!
Data copy completed.
```

The example demonstrates two things: first, *copy_async()* allows the calling thread to continue to do more work without waiting for the copy to complete; and second, it uses the *completion_task::then()* method to specify further operations to execute after the task itself has completed.

On Windows 7, this asynchronous approach to copying is more important because it also minimizes lock contention. On Windows 7, C++ AMP copy operations from the GPU to CPU involve two locking operations: first a process-wide DirectX kernel lock is taken, followed by a read lock on the source data. If the C++ AMP kernel calculating the results still has a write lock on the data being copied, the copy operation will take the DirectX kernel lock and block when attempting to acquire a read lock on the source data until the C++ AMP kernel completes and frees the resource lock. This means that the copy operation holds the DirectX kernel lock for the entire duration of the C++ AMP kernel execution and the data transfer, preventing other threads from submitting work to other GPUs during this period. If your application is executing C++ AMP kernels from more than one CPU thread, this lock contention will result in serialization of kernels that were intended to run concurrently on different GPUs. The end result is that your application will not see the performance gains you expect from adding more GPUs.

The key to getting the best possible performance is to minimize the length of time during which the copy operation takes these locks. The code here can be rewritten to minimize the time that the copy call holds the process-wide DirectX kernel lock.

```
std::vector<float> resultData(100000, 0.0f);
array<float, 1> resultArr(resultData.size());

// parallel_for_each calculates resultArr data...

copy(resultArr, resultData.begin());
```

The following code does this by queuing the copy on another thread by using *copy_async()*. It then waits for all work on the accelerator view to complete before attempting to get the result of the copy. This means that the locks are taken for the shortest possible time.

```
// parallel_for_each calculates resultArr data...

completion_future f = copy_async(resultArr, resultData.begin());
resultArr.accelerator_view.wait();
f.get();
```

On Windows 7, the *accelerator_view::wait()* method has some CPU impact because it is a spin wait, so you should only use this approach where the benefits of the improved concurrency when using multiple GPUs outweighs the additional load placed on the CPU.

Note This example uses *completion_future::get()* rather than *completion_future::wait()*. In the current futures implementation, exceptions thrown by the future are only surfaced by calls to *get()*. Using *get()* ensures that your application can handle errors correctly.

Finally, after all the iterations have completed, the data is copied back into *vC* on the CPU.

```
array_view<float, 2> c(rows, cols, vC);
std::for_each(tasks.crbegin(), tasks.crend(), [=, &arrAs, &c](const TaskData& t)
{
    index<2> ind(t.writeOffset, shift);
    extent<2> ext(t.writeExt[0], t.writeExt[1] - shift * 2);
    array_view<float, 2> outData = c.section(ind, ext);
    arrAs[t.id].section(ind, ext).copy_to(outData);
});
```

The output window shows the relative performance of this iterative averaging implementation compared to the single average version described previously. Based on the time for a single average calculation, you would expect the iterative version to take 5790 ms (10 times as long). In fact, it takes 6309 ms, or an additional 582 ms. This represents an overhead of roughly 9 percent.

```
Matrix weighted average 2000 x 2000 matrix, with 121 x 121 window
Matrix size 15625 KB

Single GPU matrix weighted average took                          1070.74 (ms)
2 GPU matrix weighted average (p_f_e) took                        579.947 (ms)
2 GPU matrix weighted average took                                585.191 (ms)

Weighted average executing 10 times

2 GPU matrix weighted average took                               6309.93 (ms)
```

Dynamic Load Balancing

The example above used two identical GPUs and assumed that no other applications were scheduling work on them. What if your application is running on a machine with two or more different GPUs? For example, your computer might have come with an integrated GPU on the motherboard but you added a more powerful discrete GPU, or other applications are using some of the available GPUs for other work. In both cases, scheduling the same amount of work on each of the available GPUs will not give the best results; the application's performance will be limited by the slowest GPU.

The answer is to implement a load-balancing algorithm to allocate work between the available GPU accelerators. Your application can load-balance based on the relative performance of each GPU, using either the time taken for a kernel to run or the amount of work completed. In some cases, if the

GPUs have wildly differing performance characteristics, then just using the best one or two might be the more efficient solution.

A common approach for doing this is the master-worker pattern. The master breaks the problem up into tasks and adds them to a queue. The master then assigns tasks from the queue to the available workers. Once a worker completes a task, it returns the results to the master. The master then assigns another task to the worker until no more tasks remain. Finally, it shuts down the workers and hands the results off to the application. The example shown here uses a work-stealing variation of master-worker in which worker GPUs take work from a master task queue on the CPU rather than waiting for it to be assigned to them.

The advantage of this approach is that it automatically load-balances across worker GPUs with different performance characteristics. It will also efficiently handle scheduling work on GPUs with varying workloads from other applications running on them. For effective load balancing there must be enough tasks to occupy all the available GPUs. Smaller tasks make for effective load balancing but add to the management overhead. Which task size you use largely depends on the application.

The following is a simple example to show the task partitioning of a C++ AMP kernel that modifies a one-dimensional array. The sample uses a *Task* type to track the range within the input data associated with each task.

```
typedef std::pair<size_t, size_t> Task;

inline size_t GetStart(Task t) { return t.first; }
inline size_t GetEnd(Task t) { return t.first + t.second; }
inline size_t GetSize(Task t) { return t.second; }
```

The example creates a *concurrent_queue<Task>* of tasks and then starts a thread for each accelerator using a *parallel_for*. Each thread pops tasks from the queue and executes a C++ AMP kernel to process the section of the array associated with the task. Once the queue is empty, the *parallel_for* completes.

```
const size_t dataSize = 101000;
const size_t taskSize = dataSize / 20;
std::vector<int> theData(dataSize, 1);

//  Divide the data up into tasks

concurrent_queue<Task> tasks;
for (size_t i = 0; i < theData.size(); i += taskSize)
    tasks.push(Task(i, std::min(i + taskSize, theData.size()) - i));

//  Start a task for each accelerator

parallel_for(0, int(accls.size()), [=, &theData, &tasks, &critSec](const unsigned i)
{
    Task t;
    while (tasks.try_pop(t))
    {
        array_view<int> work(extent<1>(GetSize(t)), theData.data() + GetStart(t));
        parallel_for_each(accls[i].default_view, extent<1>(GetSize(t)),
```

```
            [=](index<1> idx) restrict(amp)
        {
            work[idx] = // ...
        });
        // Wait in order to stop synchronize from blocking the process
        accls[i].default_view.wait();
        work.synchronize();
    }
});
```

For clarity, the code that writes status updates to the console has been removed. You can see the full source code in Chapter8\main.cpp in the *WorkStealingExample()* function.

Note The preceding code sample contains an additional call to *accls[i].default_view.wait();* prior to synchronizing the *work* data. This is required on Windows 7 to ensure that calls to *array_view::synchronize()* do not block because this will prevent all other threads from accessing the GPUs. This is not the case on Windows 8.

As you can see from the output when running a Debug build, tasks are executed on both the available GPUs, but GPU 1 does the majority of the work. In this case, GPU 0 is also being used by other processes and has a display connected to it.

```
Queued 20 tasks
 Starting tasks on 1: NVIDIA GeForce GTX 570
 Starting tasks on 0: NVIDIA GeForce GTX 570
  Finished task 0 - 5050 on 1
  Finished task 10100 - 15150 on 1
  Finished task 15150 - 20200 on 1
  Finished task 20200 - 25250 on 1
  Finished task 25250 - 30300 on 1
  Finished task 30300 - 35350 on 1
  Finished task 5050 - 10100 on 0
  Finished task 35350 - 40400 on 1
  Finished task 40400 - 45450 on 0
  Finished task 45450 - 50500 on 1
  Finished task 50500 - 55550 on 0
  Finished task 55550 - 60600 on 1
  Finished task 60600 - 65650 on 0
  Finished task 65650 - 70700 on 1
  Finished task 70700 - 75750 on 0
  Finished task 75750 - 80800 on 1
  Finished task 80800 - 85850 on 0
  Finished task 85850 - 90900 on 1
  Finished task 90900 - 95950 on 0
  Finished task 95950 - 101000 on 1
 Finished 7 tasks on 0
 Finished 13 tasks on 1
```

You might be considering using the WARP accelerator in conjunction with the GPUs to add another accelerator and thereby improve overall performance. Often, this might not result in any significant improvement in performance. First, the WARP accelerator usually achieves only a small

fraction of the performance of a dedicated/physical GPU, so the additional CPU overhead and code complexity required to coordinate the additional WARP accelerator don't result in any overall gains. Second, the CPU is already being used to coordinate the task parallelism to control the GPUs and copy data to and from them. Running a WARP accelerator workload on the CPU in combination with a physical GPU might degrade performance rather than improve it because the WARP accelerator uses CPU resources that would otherwise be used to distribute work to the GPUs. Therefore, it's recommended that your application use the WARP only as a fallback CPU solution when no GPU accelerators are available.

Note For a more general discussion of the master-worker pattern and other patterns for distributing work on parallel computers, see *Patterns of Parallel Programming* by Mattson, Sanders, and Massingill.

Braided Parallelism

Combining task parallelism with data parallelism is often referred to as braided parallelism. This pattern has obvious applications when it comes to programming today's heterogeneous computers. For maximum performance, your application should make use of all the available processors, both on the CPU and GPUs.

So far the examples in this chapter have used the CPU just to orchestrate work being executed on C++ AMP-enabled accelerators. Braided parallelism can be taken further because it allows you to leverage the power of both the CPU's cores and any available GPUs. If some parts of your application lend themselves to massive data parallelism on the GPU but others are more suitable to execution on the CPU, then it's possible to combine the PPL and C++ AMP to take advantage of both.

When deciding which parts are best placed on the GPU and which should remain on the CPU, you should think carefully about your application's overall workflow. Even some data-parallel algorithms might be better suited to executing on the CPU. For example, if the algorithm doesn't use enough data to keep the majority of the GPU's threads occupied or can't meet the restrictions required of code running in a C++ AMP kernel, it's a poor fit for C++ AMP. You should also consider reorganizing your workflow to minimize both the number of data transfers between the GPU and CPU and the volume of data transferred.

The Cartoonizer case study in Chapter 10 illustrates using braided parallelism to process images and video using a task-parallel pipeline on the CPU combined with data-parallel image processing on the GPU. The pipeline on the CPU loads, reformats, and resizes images or video frames. The GPU(s) are used to cartoonize the images before the CPU finally displays the result. Here, PPL tasks running on the CPU execute part of the processing and orchestrate C++ AMP accelerators.

When designing a braided application, it's important to consider the overall workflow of your application. It might be tempting to simply measure and profile your application and then to rewrite the data-parallelizable hotspots as C++ AMP kernels so that they can execute on the GPUs.

Although this will certainly make some parts of your application run faster, Amdahl's law will eventually limit overall application performance. Taking a more holistic view during (re)design will probably lead to finding more exploitable opportunities for parallelism and consequently better application performance.

The PPL, the Standard Library, and C++ AMP all provide support for creating parallel workflows using asynchronous methods. This allows you to create applications that execute work on both the CPU and GPU(s) concurrently, maximizing your application's performance.

Although a full discussion of asynchronous programming and the Futures and Task Graph patterns are outside the scope of this book, good introductions to both can be found on MSDN "Parallel Programming with Microsoft Visual C++, 5: Futures" at *http://msdn.microsoft.com/en-us/library/gg663533* and on the Berkeley Patterns Wiki at *http://parlab.eecs.berkeley.edu/wiki/patterns/patterns*.

Many of the tradeoffs and guidelines for designing braided parallel applications are the same as those for designing all parallel applications. There is a significant overhead associated with moving data to and from a discrete GPU, and the design should seek to minimize this. In some cases, this might mean reordering your application workflow to reduce the number of data copies. In others, it might mean implementing some parts of your workflow in C++ AMP even though they are more suited to a task-parallel implementation on the CPU.

The design should also account for the very different performance characteristics of GPUs for different workloads. They perform data-parallel work very efficiently but perform poorly when the workload can't be (re)written in a data-parallel way. Some types of computation are hard to implement in a data-parallel manner—for example, code that makes heavy use of branching. These parts of your application might be better executed on the CPU.

The Cartoonizer case study in Chapter 10 covers a complete application implemented with braided parallelism.

Falling Back to the CPU

If no C++ AMP-capable GPUs are available, your application could default back to parallel implementation on the CPU by using the PPL or the C++ AMP WARP accelerator. The section "Enumerating Accelerators" covers how to enumerate the available accelerators and choose the best one for your application. By default, the C++ AMP run time will fall back to the WARP accelerator if it is available (on Windows 8) and no C++ AMP-capable GPU accelerators are present.

Using a WARP accelerator allows your application to run on the CPU the same code that runs on the GPU, so there is less code to maintain. The WARP accelerator takes advantage of multicore and SIMD instructions and can result in comparable or even better performance than PPL code running on the CPU. This is particularly true if your algorithm would have been implemented on the CPU in a data-parallel way. Coding your algorithm in C++ AMP makes it simpler for the compiler to make good use of all the CPU cores and to vectorize your code.

In some cases, you might be able to use a different algorithm and data structures on the CPU to improve the performance that C++ AMP code running on WARP would achieve. This is especially true if there is a very efficient task-parallel approach that maps better to a multicore CPU than the data-parallel C++ AMP code. The case studies included in this book illustrate these tradeoffs.

The NBody case study (see Chapter 2) does not use WARP; if no suitable GPU is available, it falls back to a custom implementation written for the CPU, the advanced CPU integrator. The advanced CPU integrator is able to halve the number of force calculations by taking advantage of the force particle A exerts on particle B being the exact opposite of the force particle B exerts on particle A. It also breaks down the calculation in such a way as to maximize cache coherence, and therefore it improves core utilization as the application becomes memory-bound. The advanced CPU integrator also uses explicitly coded SSE vectorization using intrinsic functions. This also improves the performance of the advanced CPU integrator. In contrast, the NBody sample's C++ AMP integrators rely on the massive data parallelism of the GPUs and directly calculate both forces for each particle pair. Incurring the additional cost of these calculations is more efficient on the GPU than implementing an integrator that tries to take advantage of the pair calculations with a much more complex kernel.

The Reduction case study has no code to detect or choose accelerators and compares both sequential and parallel CPU implementations to C++ AMP implementations on every run. The copy time, whether to a GPU accelerator or to a WARP accelerator, outweighs the execution time, but the Reduction code might be appropriate for C++ AMP if it were part of a larger calculation that could justify the copy time. On a variety of hardware, the execution time on WARP was never less than the CPU execution time, but the most optimized WARP time was not significantly more than the CPU time. It's possible that the effort saved by not needing to maintain separate CPU and accelerator versions of the same algorithms would be significant. In that case, getting roughly the same performance on WARP and not needing to write a CPU version would be a good solution, producing an application that runs on a variety of hardware without needing to be written twice.

In Chapter 10, the Cartoonizer case study shows an example in which WARP delivers better performance than the CPU implementation. In this case, the CPU code uses the same data-parallel algorithm as the C++ AMP code and relies on the C++ compiler's autovectorization features to take advantage of SIMD. The C++ AMP implementation using WARP runs faster than the CPU implementation because it is able to better take advantage of all the cores and their vector processing units.

Few developers can afford to declare that their application won't run on hardware that doesn't include a DirectX 11 accelerator. Whether you choose to support configurations without a hardware accelerator by using WARP or by creating a CPU-based implementation using PPL—and possibly SSE—largely depends on the nature of your application. WARP might well be the best choice if your algorithm is data parallel and does not use double precision or it's not possible to take advantage of task parallelism on the CPU to write a more efficient implementation.

Summary

C++ AMP provides a flexible model for selecting the right accelerator for your application. When no GPUs are available, your application can fall back on the CPU using the WARP accelerator (on Windows 8) to execute your data-parallel code. If your algorithm can be expressed more efficiently in a task-parallel way on the CPU, then your application can also provide an alternative implementation using the PPL. You will also need to implement a CPU version of your algorithm if you intend to support target machines running Windows 7 that do not have a C++ AMP-capable GPU.

C++ AMP and the PPL can also be combined to leverage the power of multiple GPUs and multiple CPU cores. The performance gains from running on more than one GPU can be very significant, provided the algorithm can be split efficiently between GPUs and the overhead of any synchronization and data copying minimized. Braided parallelism also provides more opportunities for taking advantage of both data parallelism on the GPU and task parallelism on the CPU to maximize application performance.

The NBody case study code from Chapter 2 shows how it's possible to use C++ AMP to take advantage of multiple GPUs. The *NBodyAmpMultiTiled* class defined in NBodyAmpMultiTiled.h shows how to implement n-body on more than one GPU accelerator. In the NBody example, the particle update calculation is divided among the available GPUs. At the end of each time step the new particle positions and velocities are copied back onto the CPU and then the new data for all particles is sent to the GPUs. The Cartoonizer case study presented next in Chapter 10 also discusses using C++ AMP on more than one GPU in more detail. In this case, the Cartoonizer shares only image halo data among GPUs after each stage of the calculation.

CHAPTER 10

Cartoonizer Case Study

In this chapter:

Prerequisites 224
Running the Sample 224
Structure of the Sample 228
The Pipeline 229
The Pipeline Cartoonizing Stage 236
Using Multiple C++ AMP Accelerators 246
Cartoonizer Performance 252
Summary 255

This chapter introduces another example of how to use C++ AMP, this time for image processing. The Cartoonizer sample shows how to apply a sequence of image processing operations that cartoonize an image or series of images. First, the image is resized to fit into the output window. Then a color simplification process is iteratively applied to the image to flatten the color palette. Finally, the sample applies an edge detection algorithm to add the appearance of a drawing to the color-flattened image. It draws the completed image on screen, and the process repeats for the next image in the series. The Cartoonizer demonstrates how to get the most out of the available C++ AMP-capable GPU(s) by using the CPU to coordinate work across multiple GPUs with potentially different performance characteristics. It also shows how to use the CPU cores to preprocess and post-process data associated with the C++ AMP kernel.

Prerequisites

To follow along, you should download the Cartoonizer code from *http://ampbook.codeplex.com/* and ensure that you have the following software installed:

- Microsoft Windows 7 or Windows 8
- Microsoft Visual Studio 2012
- DirectX SDK, June 2010 release

It's not necessary to have a video card with a DirectX 11 driver to run the sample, but to see any benefit of moving calculations to the GPU you will need such a card. In fact, the "reference accelerator" provided with Visual Studio is much slower than any real accelerator and really usable only when debugging. See the section entitled "Prerequisites for Running the Example" in Chapter 2, "NBody Case Study," for details on how to determine if your GPU supports DirectX 11 and C++ AMP.

The Cartoonizer application also makes use of a video camera for image capture. If your computer doesn't have a camera, you can still run the application but it will be limited to cartoonizing JPEG images from disk. By default, the sample comes with three images; you can replace these or add more by simply copying them to the appropriate folder as described in the following section, "Running the Sample."

Note If you run a Debug version of this code (perhaps out of curiosity), you will notice two things. First, it will be much slower because it runs on a reference accelerator. Resize the Cartoonize window to the smallest size possible when running a Debug build. Debugging in general, and the reference accelerator in particular, are covered in Chapter 6, "Debugging." Second, if you see leak warnings like **"Detected memory leaks!"** when the application closes, you should know that the leaks are minor and caused by the order in which library code unloads. You can't change your code (or the book samples) to eliminate them, and there is no harm caused by these leaks in any event.

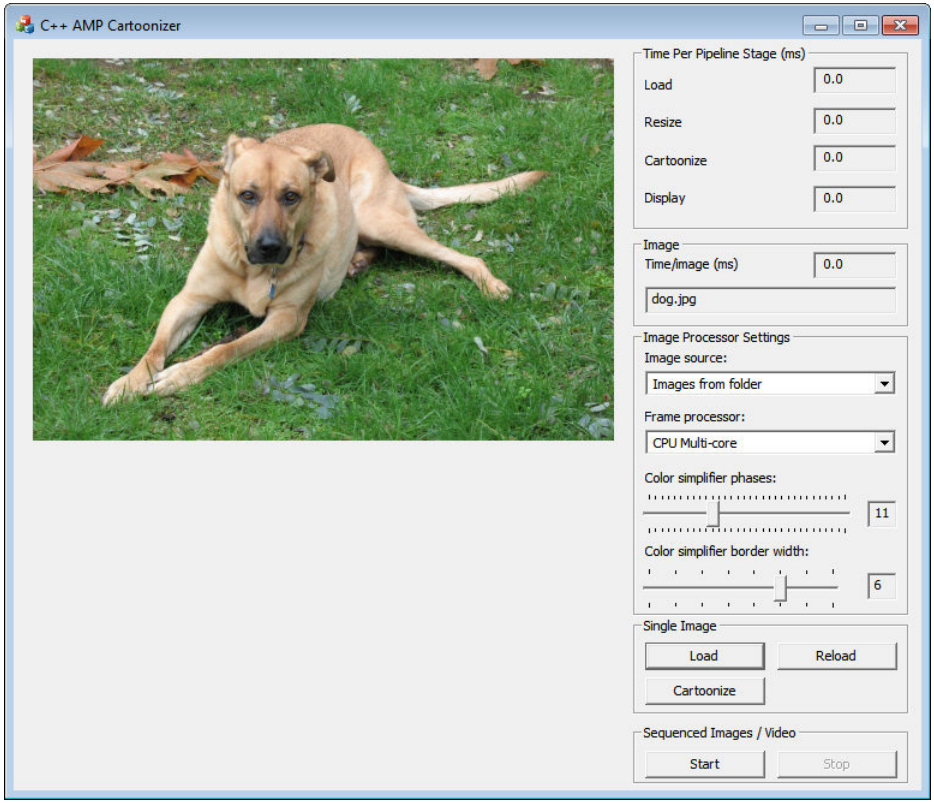
Running the Sample

The Cartoonizer demo will run on any machine with at least one C++ AMP-capable GPU. It also implements a CPU version of the algorithm and can use the REF accelerator, but with greatly reduced performance. The sample makes extensive use of the Parallel Patterns Library (PPL) and Asynchronous Agents Library to implement a parallel image processing pipeline and also to parallelize the CPU implementation of the cartoonizer algorithm. A full discussion of these libraries and the pipeline implementation is outside the scope of this book. Chapter 7, "Pipelines," of the book *Parallel Programming with Microsoft Visual C++* discusses a similar image processing pipeline and its implementation on the CPU in more detail.

Note You can find the *Parallel Programming with Microsoft Visual C++* book on O'Reilly's website at *http://shop.oreilly.com/product/0790145310507.do*.

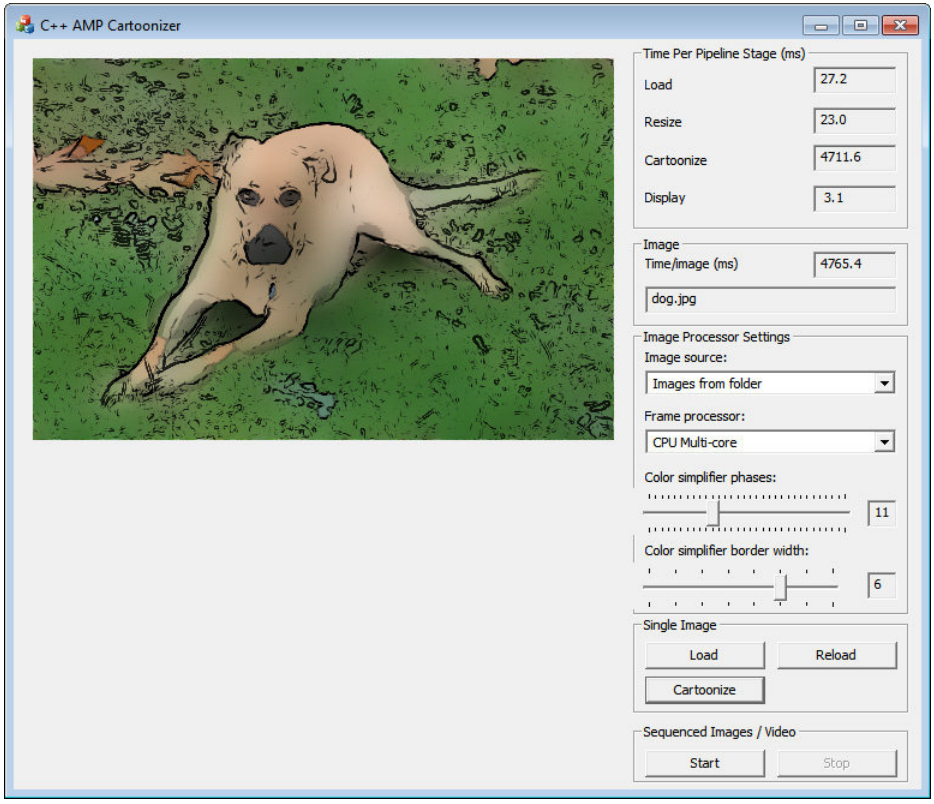
To run the sample, build the Cartoonizer solution's Release configuration and then run it with Ctrl+F5 or by navigating to the executable (at Cartoonizer\Release\Cartoonizer.exe) and double-clicking it. When the application starts, you might see warning dialog boxes if the program can't find any images or fails to detect a camera. For the application to run correctly, one or more image sources must be available. To add more JPEG images to the sample, copy them into the Cartoonizer\SampleData folder and then rebuild the application.

After the Cartoonizer starts, use the Image Source drop-down box to select Images From Folder as the input source and then click the Load button to display an image. The program will display one of the sample pictures.



Next, select the CPU Multi-Core option from the Frame processor combo box and click the Cartoonize button to run the image processor over the image. The program displays the cartoonized image in the main window, along with the image name and the time required to render the image. Depending on your hardware, this might take several seconds. Cartoonized images look as if they

were drawn with a black pen outline and then painted with a simpler color palette than the original image.



You can adjust the image produced by the application by modifying the sliders that control the number of color simplifier phases and the width of the border used to calculate the simplified color of each pixel. You can also resize the Cartoonizer application window to make the image itself smaller or larger. Processing larger images, or processing images with more color simplifier phases or a larger color simplifier border, will result in longer processing times.

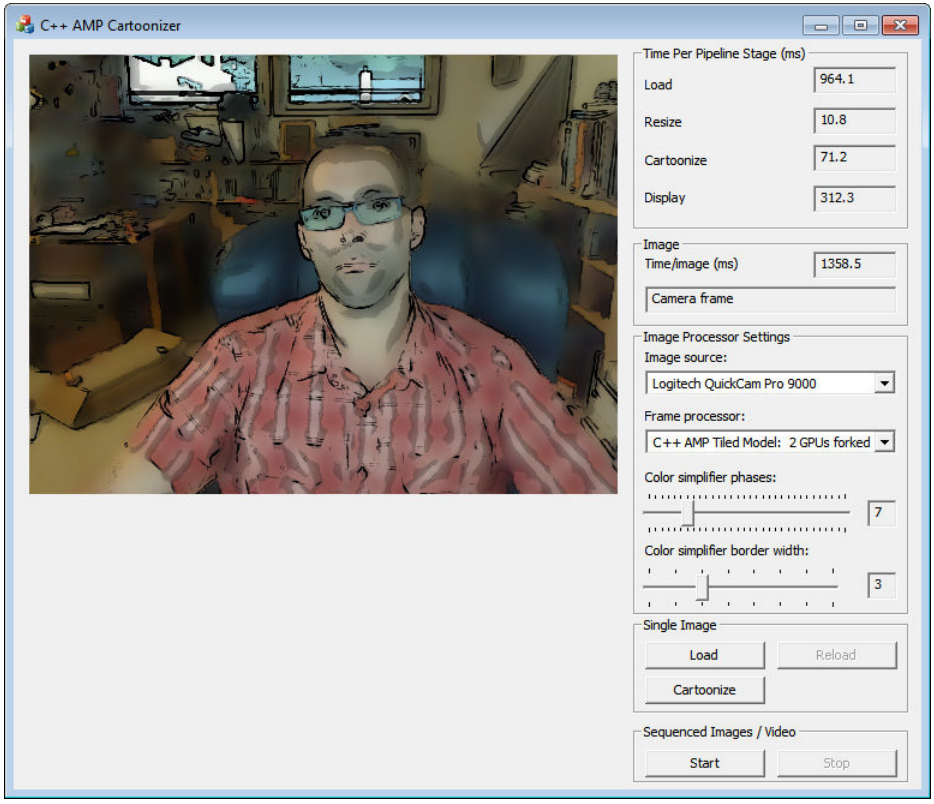
In the top right of the Cartoonizer window, you can see the time required for each stage of the image processor pipeline and the total time for each image. The time taken to process a single image is the sum of the time for the load, resize, cartoonize, and image display phases—in other words, the times for each phase add up to roughly the time required to process the entire image. You will also see that the program spends most of its time in the cartoonize stage of the process. You can experiment with different image sizes and image processor settings to see how such variations affect the time taken to process images.

The pipeline stages run in parallel. The cartoonizing stage is already parallelized on the CPU. However, the cartoonizing stage of the pipeline is still the slowest. So the best strategy for improving the pipeline's performance is to further parallelize the cartoonizing stage on the GPU using C++ AMP.

To do that, select the C++ AMP Tiled Model from the Frame Processor combo box. Then load and cartoonize the image. You will see that the time required for the cartoonizing stage is dramatically reduced but that the time for the other stages remains unchanged. However, this still results in a significant reduction in overall image processing time. Depending on your hardware and the image processor settings, the cartoonizer might no longer be the slowest stage in the pipeline.

The Cartoonizer application is actually intended to process an ordered sequence of images: either a series of JPEG images in a folder or a stream of images from a video camera connected to your computer. Click the Start button to see the pipeline process a sequence of images. After a few dozen images have been displayed, click Stop to finish processing images and review the times taken by each pipeline stage and the overall time per image.

When processing multiple images, the pipeline executes the different stages in parallel for consecutive images in the sequence instead of processing a single image at a time. One of the most important properties of a pipeline is that the average time taken to process an image is equal to the time taken by the slowest stage. If you look at the average times for each pipeline stage in the Time Per Pipeline Stage region of the Cartoonizer window, you can see that the time per image is approximately equal to the time taken by the slowest stage of the pipeline.



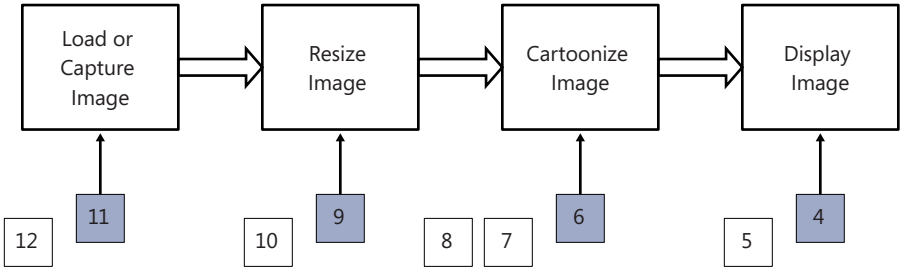
If your computer has a video camera, you can use the Cartoonizer application to process a stream of images from the camera. Select your camera from the Image Source combo box and use the Start

button to process video images. The Cartoonizer application displays a sequence of cartoonized images from the camera. Depending on your GPU, this might exhibit some time lag. You can reduce the number of color simplifier phases, the border width, and the image size in order to reduce the lag and display live cartoonized images. Depending on the camera connected to your computer, the limiting factor on the pipeline performance might be the camera's frame rate, rather than the cartoonizer stage. For example, for a camera with a maximum frame rate of 30 frames per second, the image load stage will take at least 33 ms. (This will vary depending on your camera and the room's lighting.)

For computers with more than one C++ AMP-capable GPU, you can also run the frame processors that take advantage of multiple GPUs. The case study shows two approaches to distributing the work across the GPUs that result in different performance characteristics. These differing approaches are discussed later in this chapter.

Structure of the Sample

The Cartoonizer application uses the PPL and Asynchronous Agents Library to implement a parallel pipeline. It uses agents (tasks that process input data and output results) and *message blocks* to connect and buffer messages between those agents. The main UI thread launches a *concurrency::agent* that in turn creates the pipeline and also acts as the start of the pipeline, loading or capturing images and passing them through a message block to the next agent. Each stage of the pipeline receives an image from the previous stage, processes it, and sends it to the next stage. This allows images to be processed in parallel and also maintains the ordering of images. A PPL *parallel_for_each* would not be appropriate here because it does not guarantee that the ordering of images will be maintained.

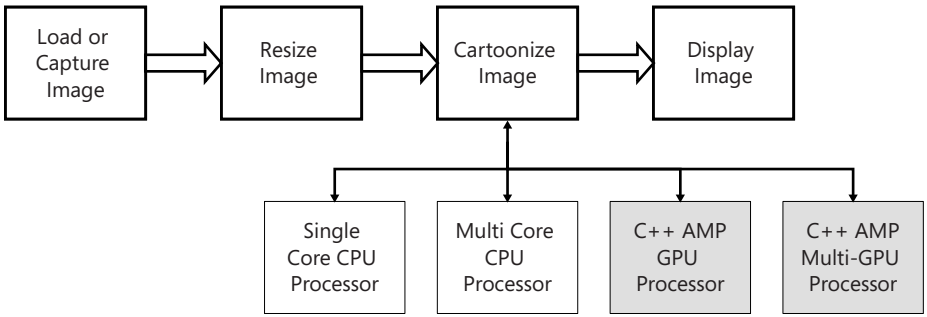


The parallel pipeline works on different images in the sequence concurrently. By running each pipeline stage as a separate parallel agent it's possible to process four images at once. As shown in the diagram, the application is displaying image 4, cartoonizing image 6, resizing image 9, and loading image 11. Between each stage, further images can be queued to minimize latency between stages in the pipeline. For example, the cartoonizing stage is processing image 6, but images 7 and 8 are queued to be cartoonized.

This approach allows the application to take advantage of some of the CPU cores available by scheduling each agent on a different thread on separate cores. The CPU multicore frame processor takes further advantage of a multicore CPU by parallelizing the cartoonizer stage using the PPL's data

parallel *parallel_for* function. The frame processor processes individual rows of pixels in parallel on the CPU, making use of any remaining idle cores. This combination of task and data parallelism is often referred to as braided parallelism.

The Cartoonizer application implements several frame processors, which run on a single CPU core, multiple CPU cores, and on single or multiple GPUs. You can configure the Cartoonize Image stage to run different image processors using the Frame Processor drop-down combo box.



The remainder of this chapter focuses on how the pipeline is implemented and on how to improve the performance of the cartoonizing stage. It describes how to implement frame processors with C++ AMP. The application uses the braided parallelism described in Chapter 9, "Working with Multiple Accelerators," to coordinate tasks running on the CPU with data-parallel processing on the GPU. For computers with more than one GPU, the sample also demonstrates two strategies for coordinating image processing across multiple GPUs.

The Pipeline

The Cartoonizer application is built around a single pipeline implementation launched from the main window. This section reviews the application's key data structures and explains how the pipeline is launched. Later sections contain more details on the individual cartoonizing frame processors.

Data Structures

The pipeline passes images between stages as *ImageInfoPtr*—shared pointers to *ImageInfo* instances (defined in ImageInfo.h) that wrap both the image *Bitmap* and other associated data.

```
typedef std::shared_ptr<Gdiplus::Bitmap> BitmapPtr;

class ImageInfo
{
private:
    int m_sequenceNumber;
    std::wstring m_imageName;
    BitmapPtr m_pBitmap;
```

```
    // ...
};

typedef std::shared_ptr<ImageInfo> ImageInfoPtr;
```

Note that the pipeline passes *ImageInfoPtr* instances rather than *ImageInfo* between each stage of the pipeline. Using pointers improves performance by removing the overhead of copying *ImageInfo* instances between each stage.

The application loads both image files and video frames as ARGB32 format GDI *Bitmap* objects. Each pixel is stored in 32 bits with 8 bits for each of the Alpha, Red, Blue, and Green values. A 32-bit ARGB value can be mapped onto an (32-bit) *unsigned long,* which is a valid C++ AMP type. For clarity, the sample defines an *ArgbPackedPixel* type to refer to pixel data.

```
typedef unsigned long ArgbPackedPixel;
```

After the data has been transferred to the accelerator as an *array_view<const ArgbPackedPixel, 2>,* it is unpacked into an *RgbPixel* struct during each processing operation. An operation is a single color simplification phase or edge detection.

```
struct RgbPixel
{
    unsigned long r;
    unsigned long g;
    unsigned long b;
};
```

Similarly, after the processing operation is complete, the *RgbPixel* data is packed back into an *ArgbPackedPixel*. The RgbPixel.h header file contains the definition of the data types and the *UnpackPixel()* and *PackPixel()* functions.

```
const int fixedAlpha = 0xFF;

inline ArgbPackedPixel PackPixel(const RgbPixel& rgb) restrict(amp)
{
    return (rgb.B | (rgb.G << 8)  | (rgb.R << 16) | (fixedAlpha << 24));
}

inline RgbPixel UnpackPixel(const ArgbPackedPixel& packedArgb) restrict(amp)
{
    RgbPixel rgb;
    rgb.b = packedArgb & 0XFF;
    rgb.g = (packedArgb & 0xFF00) >> 8;
    rgb.r = (packedArgb & 0xFF0000) >> 16;
    return rgb;
}
```

Note that the *UnpackPixel()* function ignores the pixel's alpha value, which the cartoonizing algorithm doesn't require, and the *PackPixel()* function sets the alpha value to *0xFF* (100 percent opacity).

Storing the data as a *ArgbPackedPixel* (an *unsigned long*) coalesces memory accesses and reduces the total amount of data that needs to be read and written using each processing operation from 12 bytes per pixel to 4 bytes per pixel.

The alternative would be to convert the entire image from ARGB values to *RgbPixel* structures on the GPU and then run the image processing on an *array_view<RgbAmp, 2>* before finally converting it back to *ArgbPackPixel* data. Experimentation showed that in this case it's more efficient to store the data as *ArgbPackedPixel*. The benefits of coalesced memory access and reduced data volume outweigh the additional overhead of unpacking and packing pixels during each frame processing operation.

The *CartoonizerDlg::OnBnClickedButtonStart()* Method

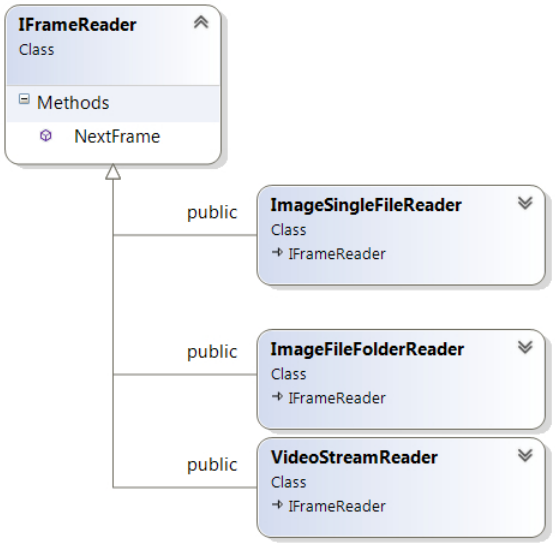
You can see the code that starts the pipeline in CartoonizerDlg.cpp in the project's UI folder. The *OnBnClickedButtonStart()* method handles the Start button click event. It creates the appropriate *IFrameReader* instance and a new *ImagePipeline* agent that processes either sequences of images or video frames and then calls the *agent::start()* method. The remaining code in the method updates the UI and configures data structures associated with performance monitoring.

```
void CartoonizerDlg::OnBnClickedButtonStart()
{
    UpdateData(DDXReadData);
    StopPipeline();

    std::shared_ptr<IFrameReader> reader;
    if (IsPictureSource())
        reader = std::make_shared<ImageFileFolderReader>(FileUtils::GetApplicationDirectory());
    else
        reader = std::make_shared<VideoStreamReader>(GetInputSource().Source);

    m_pipeline = std::unique_ptr<ImagePipeline>(
        new ImagePipeline(this, reader, m_frameProcessorType,
            kPipelineCapacity, m_cancelMessage, m_errorMessages));
    m_pipelinePerformance = PipelinePerformanceData(m_pipeline->GetCartoonizerProcessorCount());
    m_pipeline->start();
    m_pipelinePerformance.Start();
    SetButtonState(kPipelineRunning);
}
```

The pipeline continues to run on a separate thread and notifies *CartoonizerDlg*, running on the main UI thread, when it should update both the current image and any associated performance data. The display agent, *ImageDisplayAgent::DisplayImage(),* does this by calling *CartoonizerDlg::NotifyImageUpdate()*. This posts a *WM_UPDATEWINDOW* message to the window, causing the *CartoonizerDlg::OnPaint()* method to be called when the window message is processed. The *OnPaint()* method then updates the UI.

The *ImagePipeline* Class

The *ImagePipeline* class configures the pipeline and then acts as the first pipeline stage, feeding images into the pipeline. The image source is configurable and provided by the *std::shared_ptr<IFrameReader> reader* constructor parameter. *IFrameReader* interface has a single method, *NextFrame(),* that returns *ImageInfoPtr* instances.

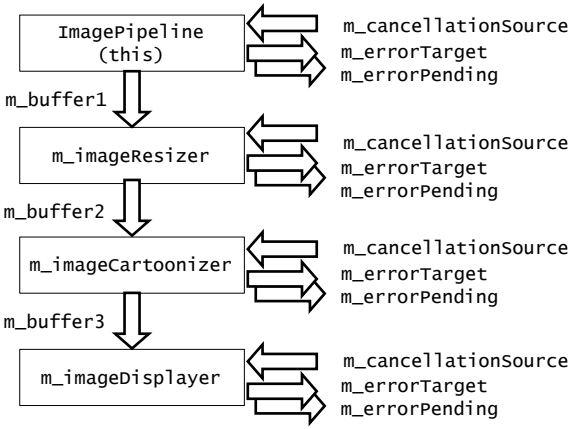


The *NextFrame()* method for each *IFrameReader* implementation behaves as follows:

- **ImageFileFolderReader.** Returns images from a folder in alphabetical order, returning to the first image when it reaches the last one
- **ImageSingleFileReader.** Returns a single image from disk and then returns *nullptr* to shut down the pipeline
- **VideoStreamReader.** Captures and returns video frames from the computer's camera

You can find the full code for all three *IFrameReader* implementations in IFrameReader.h.

The *ImagePipeline::Initialize()* method first configures the image processing pipeline, connecting the individual resizer, cartoonizer, and displayer agents with message blocks: *m_buffer1*, *m_buffer2,* and *m_buffer3*. It also connects a cancellation message source, *m_cancellationSource,* that the *CartoonizerDlg* class uses to shut down the pipeline in response to user input. Similarly, it connects a Boolean error message target, *m_errorPending*, to signal that one of the pipeline stages has encountered an error, and another message target, *m_errorTarget*, to allow the pipeline agents to send *ErrorInfo* objects to the *CartoonizerDlg* for display. The following figure shows how the pipeline agents and buffers are configured:



The *Initialize()* method also calls *CartoonizerFactory::Create()* to create an instance of a class derived from the abstract *ImageCartoonizerAgentBase* class. This approach allows different cartoonizer implementations to be used depending on the user's selection in the Frame processor drop-down box. The *Initialize()* method also creates instances of the resizer and displayer agents and configures them to communicate through the *unbounded_buffer<ImageInfoPtr>* message blocks.

```
class ImagePipeline : public AgentBase
{
private:
    std::shared_ptr<IFrameReader> m_frameReader;
    FrameProcessorType m_processorType;
    PipelineGovernor m_governor;
    unbounded_buffer<ImageInfoPtr> m_buffer1, m_buffer2, m_buffer3;
    std::unique_ptr<ImageResizeAgent> m_imageResizer;
    std::shared_ptr<ImageCartoonizerAgentBase> m_imageCartoonizer;
    std::unique_ptr<ImageDisplayAgent> m_imageDisplayer;

public:
    ImagePipeline(IImagePipelineDialog* const dialog, std::shared_ptr<IFrameReader> reader,
        FrameProcessorType processorType, int pipelineCapacity,
        ISource<bool>& cancel, ITarget<ErrorInfo>& errorTarget) :
        AgentBase(dialog, cancel, errorTarget),
        m_frameReader(reader), m_processorType(processorType),
        m_governor(pipelineCapacity), m_imageResizer(nullptr), m_imageCartoonizer(nullptr),
        m_imageDisplayer(nullptr)
    {
        Initialize();
    }
    // ...

private:
    void Initialize()
    {
        MFRatio aspectRatio;
        aspectRatio.Numerator = aspectRatio.Denominator = 1;
        m_imageResizer =
            std::unique_ptr<ImageResizeAgent>(new ImageResizeAgent(m_dialogWindow,
                m_cancellationSource, m_errorTarget,m_buffer1, m_buffer2, aspectRatio));
```

```
        m_imageCartoonizer =
            CartoonizerFactory::Create(m_dialogWindow, m_processorType,
                m_cancellationSource, m_errorTarget, m_buffer2, m_buffer3);
        m_imageDisplayer = std::unique_ptr<ImageDisplayAgent>(new ImageDisplayAgent(
            m_dialogWindow, m_cancellationSource, m_errorTarget, m_governor, m_buffer3));
    }
};
```

The *run()* method then starts the pipeline agents and uses a *do* ... *while* loop to read images from an *IFrameReader* instance and send them to *m_buffer1* in the pipeline. The method reads images until one of three conditions is satisfied:

- The frame reader returns a *nullptr*. The *ImageSingleFileReader* single frame reader does this to shut down the pipeline after reading one frame from disk.
- The user stops image processing. When the Stop button is clicked, the *CartoonizerDlg::StopPipeline()* method sends a message to the *m_cancellationSource*. The *AgentBase::IsCancellationPending()* method returns *true* if the message block associated with *m_cancellationSource* contains a cancel message.
- An error occurs. If one of the agents throws an exception error, information is propagated back to the UI by way of the *m_errorTarget* buffer and a message is also sent to the *m_errorPending* buffer. The *AgentBase::IsCancellationPending()* method returns *true* if the message block associated with the *m_errorPending* contains a message indicating that an error occurred.

The following shows the image pipeline's main processing code. Additional code to generate trace output and handle exceptions has been elided for clarity.

```
void run()
{
    m_imageResizer->start();
    m_imageCartoonizer->start();
    m_imageDisplayer->start();

    LARGE_INTEGER clockOffset;
    QueryPerformanceCounter(&clockOffset);
    int sequence = kFirstImage;
    ImageInfoPtr pInfo;
    try
    {
        do
        {
            LARGE_INTEGER start;
            QueryPerformanceCounter(&start);
            pInfo = m_frameReader->NextFrame(sequence++, clockOffset);
            if (pInfo != nullptr)
            {
                pInfo->PhaseEnd(kLoad, start);
                m_governor.WaitForAvailablePipelineSlot();
            }
            asend(m_buffer1, pInfo);
```

```
            }
            while ((pInfo != nullptr) && !IsCancellationPending());
        }
        catch ( ... ) { ... }

        m_governor.WaitForEmptyPipeline();
        if (pInfo != nullptr)
            asend<ImageInfoPtr>(m_buffer1, nullptr);

        agent* agents[3] = { m_imageResizer.get(),
            m_imageCartoonizer.get(), m_imageDisplayer.get() };
        agent::wait_for_all(3, agents);
        done();
    }
```

The message blocks between the agents in the pipeline are *unbounded_buffer* message blocks. The *unbounded_buffer* block can hold an unlimited number of items. This would usually cause the pipeline to fill up with unprocessed images because the load stage of the pipeline can add images to the pipeline faster than the later pipeline stages can process them. To prevent this, the *ImagePipeline* uses a *PipelineGovernor* instance, *m_governor*, to monitor and limit the number of images in the pipeline. The *PipelineGovernor::WaitForAvailablePipelineSlot()* method blocks when the pipeline is at capacity, thereby limiting the number of items that can be in the pipeline at any one time.

Finally, the pipeline waits for all the agents to finish before signaling that it has completed by calling *agent::done()* to inform the controlling code in *CartoonizerDlg* that the pipeline has shut down. Note that some of the timing-related and exception-handling code has been elided to simplify the code shown here. The complete source can be found in ImagePipeline.h in the Cartoonizer project's Pipeline folder.

If you build the debug configuration of the Cartoonizer and press F5 to start the sample with the debugger attached, you can view the trace output in Visual Studio's output window. This will show the overall workflow of the pipeline. For example, if you cartoonize a single image, the output window will contain something similar to the following output:

```
Using cartoonizer processor: 3, Sequential pipeline agent, simple.
Using frame processor: 3, C++ AMP tiled.
Reading file: 1 dog.jpg
Image pipeline waiting for pipeline to empty...
Resize image: frame 1.
Resize image: empty frame.
Resize agent shutting down.
Cartoonize image: frame 1.
Configure frame buffers: New image size 505 x 759
Cartoonize image: empty frame.
Cartoonizer shutting down.
Display image: frame 1.
Display image: empty frame.
Image pipeline is empty.
Display agent shutting down.
Image pipeline agents done.
Image pipeline shutdown complete.
```

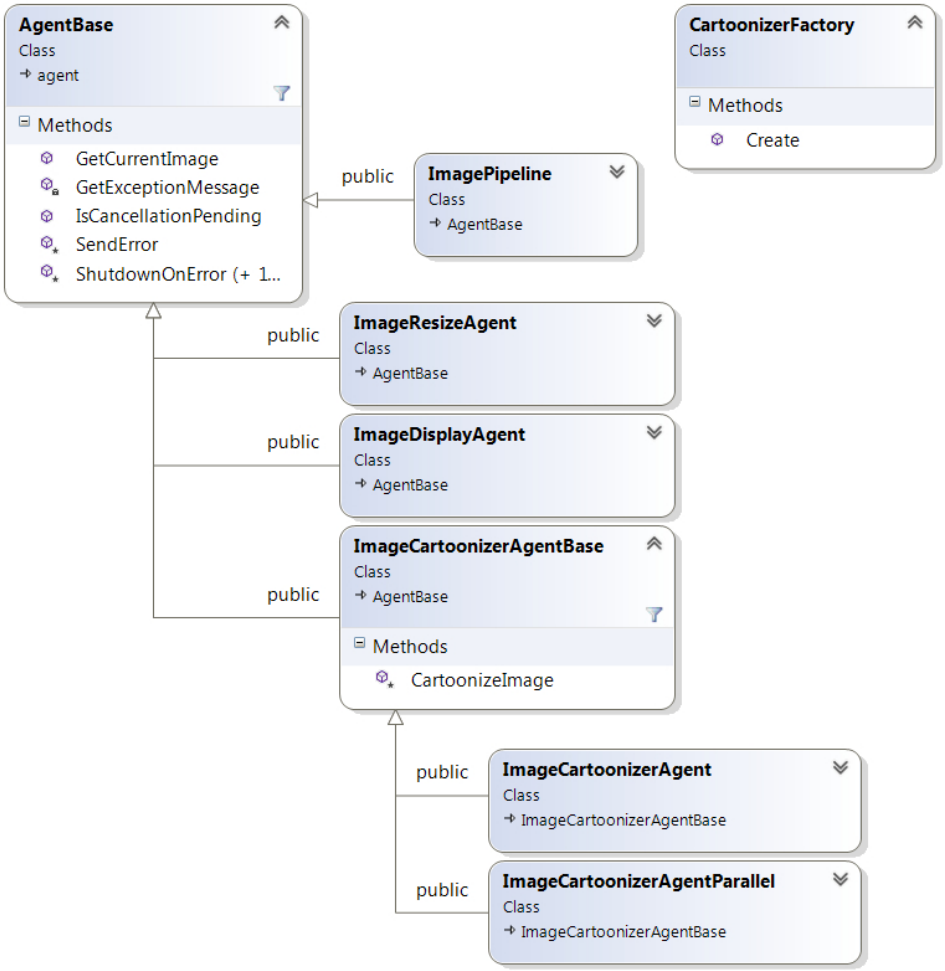
In this example, the pipeline starts an *ImageSingleFileReader*. The reader loads a single file and passes this frame down the pipeline. The file reader then sends an empty frame (*nullptr*) to signal that the pipeline stages should shut down. Each pipeline stage processes the first frame and then shuts down on receiving the empty frame. Finally, the pipeline waits until it has emptied and then waits for all the agents to indicate they are done before shutting down. The exact order of the messages can vary and does not necessarily reflect the actual ordering of events.

The Pipeline Cartoonizing Stage

The *ImagePipeline* is used to control image processing for all frame processors. This section covers the cartoonizer and frame processors used for cartoonizing on the CPU or on a single GPU.

The *ImageCartoonizerAgent* Class

The Cartoonizer project's Pipeline folder contains the definitions of the different pipeline agents in separate header files. Like the *ImagePipeline*, each agent contains a *run()* method. Each agent's *run()* method contains very similar code. The agent receives an *ImageInfoPtr* object from the input buffer *m_imageInput*, processes the image, and sends the result to the output buffer, *m_imageOutput*. The agent continues to process images until it receives a *nullptr*. Before shutting down, all agents call *agent::done()* to indicate that they have completed.



The *AgentBase* class implements all the cancellation and error-handling logic that is common to all agents. The *ImageCartoonizerAgentBase::CartoonizeImage()* method implements the functionality that actually cartoonizes the image. Both the *ImageCartoonizerAgent* and *ImageCartoonizerAgentParallel* call this method (as discussed in the section of this chapter entitled "Using Multiple C++ AMP Accelerators").

The *ImageCartoonizerAgent* calls *FrameProcessorFactory::Create()* to instantiate the correct frame processor and then calls *CartoonizeImage()* to process incoming images.

```
class ImageCartoonizerAgent : public ImageCartoonizerAgentBase
{
private:
    ISource<ImageInfoPtr>& m_imageInput;
    ITarget<ImageInfoPtr>& m_imageOutput;
    std::shared_ptr<IFrameProcessor> m_processor;

public:
```

```
    ImageCartoonizerAgent(IImagePipelineDialog* const pDialog, FrameProcessorType processorType,
        ISource<bool>& cancellationSource, ITarget<ErrorInfo>& errorTarget,
        ISource<ImageInfoPtr>& imageInput, ITarget<ImageInfoPtr>& imageOutput) :
        ImageCartoonizerAgentBase(pDialog, cancellationSource, errorTarget),
        m_imageInput(imageInput), m_imageOutput(imageOutput),
        m_processor(FrameProcessorFactory::Create(processorType))
    {
    }

    void run()
    {
        ImageInfoPtr pInfo = nullptr;
        do
        {
            pInfo = receive(m_imageInput);
            CartoonizeImage(pInfo, m_processor, m_dialogWindow->GetFilterSettings());
            asend(m_imageOutput, pInfo);
        }
        while(nullptr != pInfo);
        done();
    }
};
```

The *ImageCartoonizerAgentBase::CartoonizeImage()* method implements the cartoonizer frame processor. This method configures and locks the image *Bitmap* buffers, executes the *IFrameProcessor::ProcessImage()* method to take the bitmap data in *originalImage* and returns the cartoonized image in *processedImage*. The call to *ImageInfo::SetBitmap()* updates the *pInfo* with the processed bitmap data. The *CartoonizeImage()* method also implements performance timing and exception handling. This code has been removed for clarity.

```
class ImageCartoonizerAgentBase : public AgentBase
{
public:
    ImageCartoonizerAgentBase(IImagePipelineDialog* const pDialog,
        ISource<bool>& cancellationSource, ITarget<ErrorInfo>& errorTarget) :
        AgentBase(pDialog, cancellationSource, errorTarget)
    {
    }

protected:
    void CartoonizeImage(const ImageInfoPtr& pInfo,
        std::shared_ptr<IFrameProcessor>& processor,
        const FilterSettings& settings) const
    {
        try
        {
            if (IsCancellationPending() || (nullptr == pInfo))
                return;
```

```
            BitmapPtr inBitmap = pInfo->GetBitmapPtr();
            BitmapPtr outBitmap = BitmapPtr(inBitmap->Clone(0, 0, inBitmap->GetWidth(),
                                                inBitmap->GetHeight(), PixelFormat32bppARGB));

            Gdiplus::Rect rect(0, 0, inBitmap->GetWidth(), inBitmap->GetHeight());
            Gdiplus::BitmapData originalImage;
            inBitmap->LockBits(&rect, Gdiplus::ImageLockModeWrite,
                               PixelFormat32bppARGB, &originalImage);
            Gdiplus::BitmapData processedImage;
            outBitmap->LockBits(&rect, Gdiplus::ImageLockModeWrite,
                                PixelFormat32bppARGB, &processedImage);

            processor->ProcessImage(originalImage, processedImage,
                                    GetPhases(settings), GetNeighborWindow(settings));
            pInfo->SetBitmap(outBitmap);

            inBitmap->UnlockBits(&originalImage);
            outBitmap->UnlockBits(&processedImage);
        }
        catch ( ... ) { ... }
    }
};
```

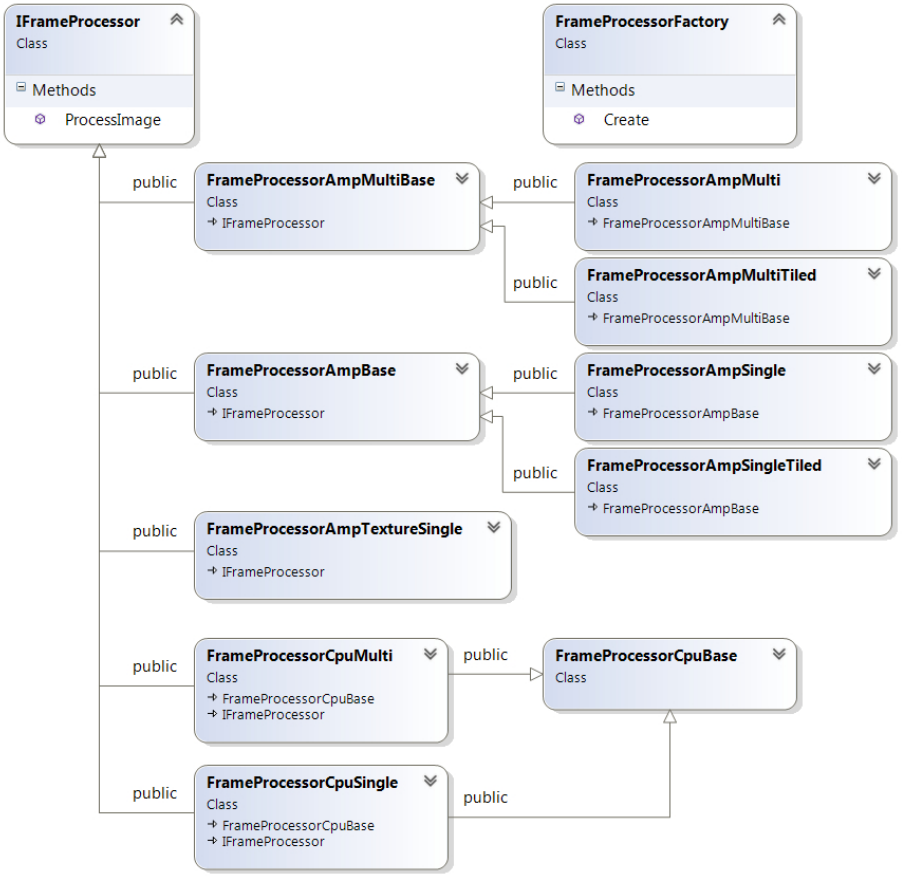
The code is factored in this way so that the *ImageCartoonizerAgent* class and *ImageCartoonizerAgentParallel* class (discussed in the section of this chapter entitled "Using Multiple C++ AMP Accelerators") can both use the *CartoonizeImage* code.

The *IFrameProcessor* Implementations

The cartoonizer pipeline supports several frame processors that implement the cartoonizing algorithm on the CPU, a single GPU, and multiple GPUs. Each frame processor implements the *IFrameProcessor* interface, which defines one pure virtual method, *ProcessImage()*.

```
class IFrameProcessor
{
public:
    virtual void ProcessImage(const Gdiplus::BitmapData& srcFrame,
        Gdiplus::BitmapData& destFrame, UINT phases,
        UINT neighborWindow) = 0;
};
```

The following class diagram shows the relationships between the various frame processor implementations.

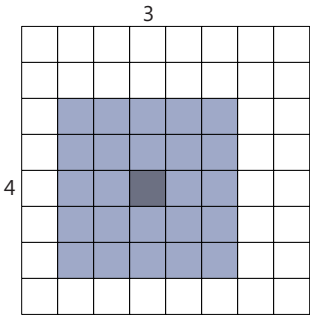


The *FrameProcessorFactory* class creates the correct *IFrameProcessor* implementation based on the *FrameProcessorType* enum.

```
class FrameProcessorFactory
{
public:
    static std::shared_ptr<IFrameProcessor> Create(FrameProcessorType processorType,
        const accelerator&  accel = accelerator(accelerator::default_accelerator))
    {
        switch (processorType)
        {
        // ... Additional processorType cases.
        case kAmpTiled:
            return std::make_shared<FrameProcessorAmpSingleTiled>(accel);
            break;
        case kAmpSimple:
            return std::make_shared<FrameProcessorAmpSingle>(accel);
            break;
```

```
case kCpuSimple:
    return std::make_shared<FrameProcessorCpuSingle>();
    break;
case kCpuSingle:
    return std::make_shared<FrameProcessorCpuMulti>();
    break;
// ... Additional processorType cases.
```

The actual mechanics of the algorithms used by the color simplification and edge detection code are outside the scope of this book. The key feature of both algorithms is that they take an array of pixels and for each pixel calculate a new value based on the surrounding pixels. These surrounding pixels are referred to as the stencil. In the case of color simplification, the user can modify the number of stencil pixels, while the edge detection algorithm always uses the eight pixels surrounding the pixel being updated. As shown in the figure, to calculate the new simplified color value for pixel [4, 3] with a neighbor window of 2, the algorithm reads values from the surrounding 24 pixels and writes the new value to the pixel at [4, 3]. It does this for each pixel in the image. Because other threads on either the CPU or GPU might be writing values to adjacent threads, the algorithm uses two copies of the image. All threads read existing values from the current image frame and write updated values to the next frame. After each phase in the pipeline the current and next images are swapped. The section of Chapter 9 entitled "Using More Than One GPU" covered a similar algorithm in some detail.



For a detailed explanation of the edge detection algorithm used by the Cartoonizer, see the Wikipedia "Canny edge detector" article at *http://en.wikipedia.org/wiki/Canny_edge_detector*.

The *FrameProcessorCpu* and *FrameProcessorCpuMulti* Classes

The *FrameProcessorCpu* class executes on a single CPU core, while the *FrameProcessorCpuMulti* uses the PPL to execute frame processing on all the available CPU cores. Both implement the *IFrameProcessor::ProcessImage()* method.

Three steps are needed to process a single image. First, the *ConfigureFrameBuffers()* method initializes an *std::array* of two frame buffers, *m_frames*. These are used to store copies of the input (*current*) and output (*next*) images for each step as *Gdiplus::BitmapData* instances. The second step, and first stage of the cartoonizing algorithm, is the color simplifier. The simplifier function, *ApplyColorSimplifierSingle()*, iteratively simplifies the colors for *phases* iterations. Each iteration reads from *m_frames[current]*, writes the result to *m_frames[next]*, and then swaps the *current* and *next* indices. The final step, edge detection, is implemented by *ApplyEdgeDetectionSingle()* and runs only

once. It detects edges by comparing the color-simplified image, *m_frames[current]*, and the original image, *srcFrame*, writing the output to *destFrame*.

The single-core implementation is shown here. The multicore implementation uses the same inheritance pattern.

```
class FrameProcessorCpuSingle : public FrameProcessorCpuBase, public IFrameProcessor
{
    void ProcessImage(const Gdiplus::BitmapData& srcFrame, Gdiplus::BitmapData& destFrame,
        UINT phases, UINT neighborWindow)
    {
        ConfigureFrameBuffers(srcFrame);

        //  Process the image. After each step swap the frame buffer indices.

        int current = kCurrent;
        int next = kNext;
        UINT shift = neighborWindow / 2;

        for (UINT i = 0; i < phases; ++i)
        {
            ApplyColorSimplifierSingle(*m_frames[current].get(), *m_frames[next].get(),
                neighborWindow, shift, shift, (srcFrame.Width - shift),
                (srcFrame.Height - shift));
            std::swap(current, next);
        }

        ++shift;
        ApplyEdgeDetectionSingle(*m_frames[current].get(), destFrame, srcFrame,
            shift, shift, (srcFrame.Width - shift), (srcFrame.Height - shift));

        //  Copy the resulting image into the destination frame.
        for (int i = 0; i < kBufSize; ++i)
            m_bitmaps[i]->UnlockBits(m_frames[i].get());
    }
};
```

Although the performance of the CPU implementations of the cartoonizer is impressive, they are not responsive enough for more interactive applications, such as cartoonizing a video stream. The remaining frame processors described in this chapter use C++ AMP to improve the performance of the cartoonizing stage.

The *FrameProcessorAmpSingle* Class

The *FrameProcessorAmpSingle* class uses C++ AMP to execute frame processing on a single AMP accelerator with support for both simple and tiled models. It's very similar to the *FrameProcessorCpuSingle* class, but it requires some additional code to deal with moving and converting image data. Rather than three steps there are now five, with two additional steps to convert and copy the image data to and from the C++ AMP accelerator.

The *FrameProcessorAmpSingle* class inherits from *FrameProcessorAmpBase*, a base class that declares two virtual methods to allow the subclass to customize the image processing behavior.

```
class FrameProcessorAmpBase : public IFrameProcessor
{
private:
    accelerator m_accelerator;
    std::array<std::shared_ptr<array<ArgbPackedPixel, 2>>, 3> m_frames;
    UINT m_height;
    UINT m_width;

public:
    // ...

    virtual inline void ApplyEdgeDetection(accelerator& acc,
        const array<ArgbPackedPixel, 2>& srcFrame, array<ArgbPackedPixel, 2>& destFrame,
        const array<ArgbPackedPixel, 2>& orgFrame, UINT simplifierNeighborWindow) = 0;
    virtual inline void ApplyColorSimplifier(accelerator& acc,
        const array<ArgbPackedPixel, 2>& srcFrame, array<ArgbPackedPixel, 2>& destFrame,
        UINT neighborWindow) = 0;

    void ProcessImage(const Gdiplus::BitmapData& srcFrame,
        Gdiplus::BitmapData& destFrame,
        UINT phases, UINT simplifierNeighborWindow)
    {
        ConfigureFrameBuffers(srcFrame);

        int current = kCurrent;
        int next = kNext;
        CopyIn(srcFrame, *m_frames[current].get());
        m_frames[current]->copy_to(*m_frames[kOriginal].get());
        for (UINT i = 0; i < phases; ++i)
        {
            ApplyColorSimplifier(*m_frames[current].get(), *m_frames[next].get(),
                simplifierNeighborWindow);
            std::swap(current, next);
        }

        ApplyEdgeDetection(*m_frames[current].get(), *m_frames[next].get(),
            *m_frames[kOriginal].get(), simplifierNeighborWindow);
        std::swap(current, next);
        CopyOut(*m_frames[current].get(), destFrame);
    }
    // ...
};
```

The *FrameProcessorAmpSingle* class simply overrides the virtual functions and calls the appropriate helper functions. This allows the different *IFrameProcessor* implementations to share the same *ProcessImage()* method code.

```
class FrameProcessorAmpSingle : public FrameProcessorAmpBase
{
public:
    FrameProcessorAmpSingle(const accelerator& accel) : FrameProcessorAmpBase(accel) { }

    virtual inline void ApplyColorSimplifier(const array<ArgbPackedPixel, 2>& srcFrame,
        array<ArgbPackedPixel, 2>& destFrame, UINT neighborWindow)
    {
        ::ApplyColorSimplifierHelper(srcFrame, destFrame, neighborWindow);
```

```
    }

    virtual inline void ApplyEdgeDetection(
        const array<ArgbPackedPixel, 2>& srcFrame, array<ArgbPackedPixel, 2>& destFrame,
        const array<ArgbPackedPixel, 2>& orgFrame, UINT simplifierNeighborWindow)
    {
        ::ApplyEdgeDetectionHelper(srcFrame, destFrame, orgFrame, simplifierNeighborWindow);
    }
};
```

The *ProcessImage()* method calls the *ConfigureFrameBuffers()* method, which creates a *std::array<std::shared_ptr<array<ArgbPackedPixel, 2>>, 3>*, *m_frames*, to store the input (*current*), output (*next*) images and original (*kOriginal*) images. Although this is similar to the CPU implementation, a third frame is needed to store a copy of the original image because the accelerator can't directly access *srcFrame* and requires the original image data for edge detection.

Next, the *CopyIn()* method copies the *srcFrame* data to the GPU as *ArgbPackedPixel* data stored in *m_frames*. The next stages of the cartoonizing algorithm follow the same pattern as the CPU implementation, iteratively simplifying the colors for *phases* iterations and then applying edge detection. Finally, the *CopyOut()* method moves the *ArgbPackedPixel* on the GPU back into pixel data in the *destFrame* bitmap.

The actual image processing is done by the *ApplyColorSimplifierHelper()* and *ApplyEdgeDetectionHelper()* methods for the simple implementation and the *ApplyColorSimplifierTiledHelper()* and *ApplyEdgeDetectionTiledHelper()* methods for the tiled implementation. For simplicity, only the simple version of *ApplyEdgeDetectionHelper()* is shown here. The concepts for the tiled implementations are no different from those discussed in Chapter 4, "Tiling," and in Chapter 5, "Case Study—Tiled NBody."

```
void ApplyEdgeDetectionHelper(const array<ArgbPackedPixel, 2>& srcFrame,
    array<ArgbPackedPixel, 2>& destFrame, const array<ArgbPackedPixel, 2>& orgFrame,
    UINT simplifierNeighborWindow)
{
    const float_3 W(ImageUtils::W);
    extent<2> ext(srcFrame.extent -
        extent<2>(simplifierNeighborWindow, simplifierNeighborWindow));

    extent<2> computeDomain(ext -
        extent<2>(FrameProcessorAmp::EdgeBorderWidth, FrameProcessorAmp::EdgeBorderWidth));
    parallel_for_each(computeDomain,
        [=, &srcFrame, &destFrame, &orgFrame](index<2> idx) restrict(amp)
    {
        DetectEdge(idx, srcFrame, destFrame, orgFrame, simplifierNeighborWindow, W);
    });
}
```

The edge detection algorithm calculates the new value for each pixel by combining the *CalculateSobel()* method result for both the original image and the color-simplified image.

```
void DetectEdge(index<2> idx, const array<ArgbPackedPixel, 2>& srcFrame,
    array<ArgbPackedPixel, 2>& destFrame, const array<ArgbPackedPixel, 2>& orgFrame,
    UINT simplifierNeighborWindow, const float_3& W) restrict(amp)
```

```
{
    const float alpha = 0.3f;
    const float beta = 0.8f;
    const float s0 = 0.054f;
    const float s1 = 0.064f;
    const float a0 = 0.3f;
    const float a1 = 0.7f;
    const int neighborWindow = 2;
    const int offset = (simplifierNeighborWindow + neighborWindow) / 2;

    index<2> idc(idx[0] + offset, idx[1] + offset);  // Corrected index for border offset.
    float Sy, Su, Sv;
    float Ay, Au, Av;
    Sy = Su = Sv = 0.0f;
    Ay = Au = Av = 0.0f;
    CalculateSobel(srcFrame, idc, Sy, Su, Sv, W);
    CalculateSobel(orgFrame, idc, Ay, Au, Av, W);

    const float edgeS = (1 - alpha) * Sy + alpha * (Su + Sv) / 2;
    const float edgeA = (1 - alpha) * Ay + alpha * (Au + Av) / 2;
    const float i = (1 - beta) * smoothstep(s0, s1, edgeS) + beta * smoothstep(a0, a1, edgeA);

    const RgbPixel srcClr = UnpackPixel(srcFrame[idc]);
    RgbPixel destClr;
    const float oneMinusi = 1 - i;
    destClr.r = static_cast<UINT>(srcClr.r * oneMinusi);
    destClr.g = static_cast<UINT>(srcClr.g * oneMinusi);
    destClr.b = static_cast<UINT>(srcClr.b * oneMinusi);
    destFrame[idc] = PackPixel(destClr);
}
```

Here, *CalculateSobel()* is the C++ AMP function that calculates a new value based on the pixel's neighbors.

```
void CalculateSobel(const array<ArgbPackedPixel, 2>& srcFrame, index<2> idx,
                        float& dy, float& du, float& dv,  const float_3& W) restrict(amp)
{
    const int gx[3][3] = { { -1, 0, 1 }, { -2, 0, 2 }, { -1,  0,  1 } };
    const int gy[3][3] = { {  1, 2, 1 }, {  0, 0, 0 }, { -1, -2, -1 } };
    float new_yX = 0, new_yY = 0, new_uX = 0, new_uY = 0, new_vX = 0, new_vY = 0;
    for (int y = -1; y <= 1; y++)
        for (int x = -1; x <= 1; x++)
        {
            const int gX = gx[x + 1][y + 1];
            const int gY = gy[x + 1][y + 1];
            float clrY, clrU, clrV;
            index<2> idxNew(idx[0] + x, idx[1] + y);
            ImageUtils::RGBToYUV(UnpackPixel(srcFrame[idxNew]), clrY, clrU, clrV, W);
            new_yX += gX * clrY;
            new_yY += gY * clrY;
            new_uX += gX * clrU;
            new_uY += gY * clrU;
            new_vX += gX * clrV;
            new_vY += gY * clrV;
        }
```

```
    dy = fast_math::sqrt((new_yX * new_yX) + (new_yY * new_yY));
    du = fast_math::sqrt((new_uX * new_uX) + (new_uY * new_uY));
    dv = fast_math::sqrt((new_vX * new_vX) + (new_vY * new_vY));
}
```

Here you can see where the *ArgbPackedPixel* gets unpacked and packed into *RgbPixel* data so that the calculation can be performed.

Finally, the data must be copied back to the CPU. The *CopyOut()* function uses the *copy_async()* and *accelerator_view::wait()* pattern discussed in the "Swapping Data between Accelerators" section of Chapter 9, "Working With Multiple Accelerators," to minimize the length of time that the GPU-to-CPU copy operation will hold a process-wide lock on the DirectX kernel, preventing other work from being submitted to GPUs.

```
void CopyOut(array<ArgbPackedPixel, 2>& currentImg, Gdiplus::BitmapData& destFrame)
{
    auto iter = stdext::make_checked_array_iterator<ArgbPackedPixel*>(
        static_cast<ArgbPackedPixel*>(destFrame.Scan0), destFrame.Height * destFrame.Width);

    completion_future f = copy_async(currentImg.section(0, 0,
        destFrame.Height, destFrame.Width), iter);
    currentImg.accelerator_view.wait();
    f.get();
}
```

This code is required only when running with multiple accelerators on Windows 7. On Windows 8 or when running on a single GPU, a single *copy()* call would be sufficient. The *FrameProcessorAmpSingle* class is reused by the *ImageCartoonizerAgentParallel* multi-GPU implementation (discussed later in this chapter), so it requires this approach.

The other stages of the C++ AMP implementation of the cartoonizing algorithm use a similar stencil-based approach to calculating new pixel values based on the surrounding pixels. You can find the full source code in the FrameProcessorAmp.h and FrameProcessorAmp.cpp files.

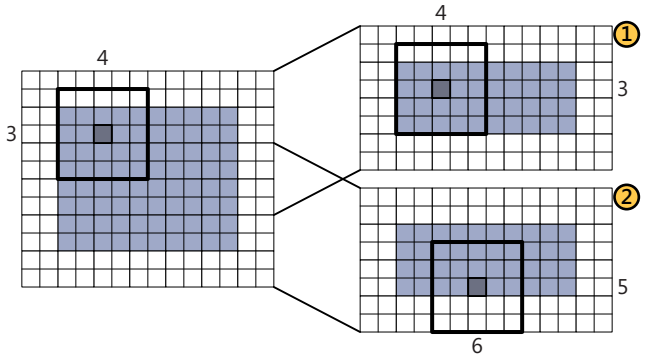
Using Multiple C++ AMP Accelerators

So far the pipeline supports only a single C++ AMP accelerator. This section describes two ways to modify the sample to support more than one accelerator. Their performance is compared in the "Cartoonizer Performance" section later in the chapter.

The *FrameProcessorAmpMulti* Class

One strategy for using more than one C++ AMP accelerator to cartoonize an image is to split the image up into blocks and process a block on each accelerator before syncing the data at the end of each calculation step. This is the approach described in the section of Chapter 9 entitled "Using More Than One GPU." The N-body case study's *NBodyAmpMultiTiled* class also uses a similar approach.

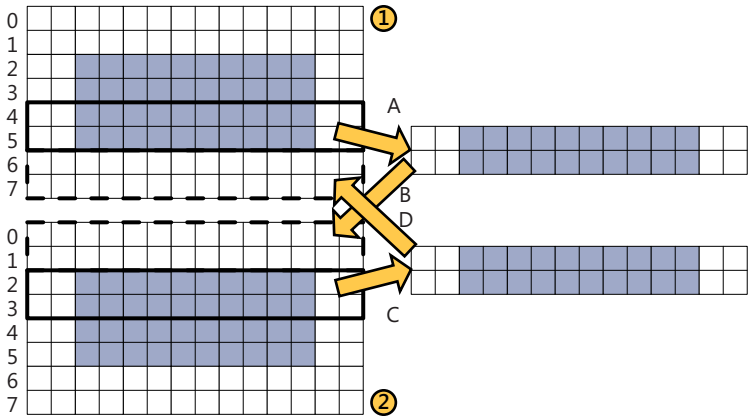
It divides the particles up across the available GPUs, updating them and then copying the updates across all the GPUs before starting the next iteration step.



The advantage of this approach is that no change is required to the *ImagePipeline* class and the ordering of images is maintained. The disadvantage is that the code for managing the work between different accelerators is more complicated and doesn't scale as well as other approaches. Looking at the *FrameProcessorAmpMultiBase::ProcessImage()* method shows the problem.

```
const UINT borderHeight = simplifierNeighborWindow / 2;
// ...
for (UINT i = 0; i < phases; ++i)
{
    std::for_each(m_frameData.begin(), m_frameData.end(), [=](TaskData& d)
    {
        ::ApplyColorSimplifierHelper(*d.frames[current].get(), *d.frames[next].get(),
            simplifierNeighborWindow);
    });

    for (UINT d = 0; d < m_frameData.size() - 1; ++d)
    {
        SwapEdges(m_frameData[d].frames[next].get(), m_frameData[d+1].frames[next].get(),
            borderHeight);
    }
    std::swap(current, next);
}
```

After each color simplifier step, the new results from the edges of each block must be exchanged with the neighboring accelerators. This involves copying the data from one accelerator to another through host memory. While this is happening, no further calculation can take place. The larger the neighbor window, the more data is transferred after each step of the calculation.



The diagram shows an example with a small 16 × 14 image with a neighbor window of two. After each color simplifier step, the new results from rows 4 and 5 on accelerator 1 must be copied into the halo cells in rows 0 and 1 of accelerator 2. This occurs in two steps: first, the data on accelerator 1 is copied to a temporary buffer on the CPU (operation A on the diagram) and then back to accelerator 2 (operation B). Similarly, the new values from accelerator 2 must be copied into the halo of accelerator 1 (operations C and D).

The *SwapEdges()* method uses staging buffers on the CPU *m_swapDataTop* and *m_swapDataBottom* of type *array<ArgbPackedPixel, 2>* to move the data between GPUs. These are initialized in the *ConfigureFrameBuffers()* method.

```
array<ArgbPackedPixel, 2> m_swapDataTop;
array<ArgbPackedPixel, 2> m_swapDataBottom;
array_view<ArgbPackedPixel, 2> m_swapViewTop;
array_view<ArgbPackedPixel, 2> m_swapViewBottom;
// ...
void ConfigureFrameBuffers(std::vector<TaskData>& taskData,
    const Gdiplus::BitmapData& srcFrame, UINT neighborWindow)
{
    bool neighborWindowChanged = m_neighborWindow != neighborWindow;
    bool widthChanged = m_width != srcFrame.Width;
    bool heightChanged = m_height != srcFrame.Height;
    m_height = srcFrame.Height;
    m_width = srcFrame.Width;
    m_neighborWindow = neighborWindow;

    if (neighborWindowChanged || widthChanged)
    {
        const UINT borderHeight = (neighborWindow - FrameProcessorAmp::EdgeBorderWidth) / 2;
        m_swapDataTop = array<ArgbPackedPixel, 2>(extent<2>(borderHeight, m_width),
            accelerator(accelerator::cpu_accelerator).default_view);
        m_swapViewTop = array_view<ArgbPackedPixel, 2>(m_swapDataTop);
        m_swapDataBottom = array<ArgbPackedPixel, 2>(extent<2>(borderHeight, m_width),
            accelerator(accelerator::cpu_accelerator).default_view);
        m_swapViewBottom = array_view<ArgbPackedPixel, 2>(m_swapDataBottom);
    }
    // ...
```

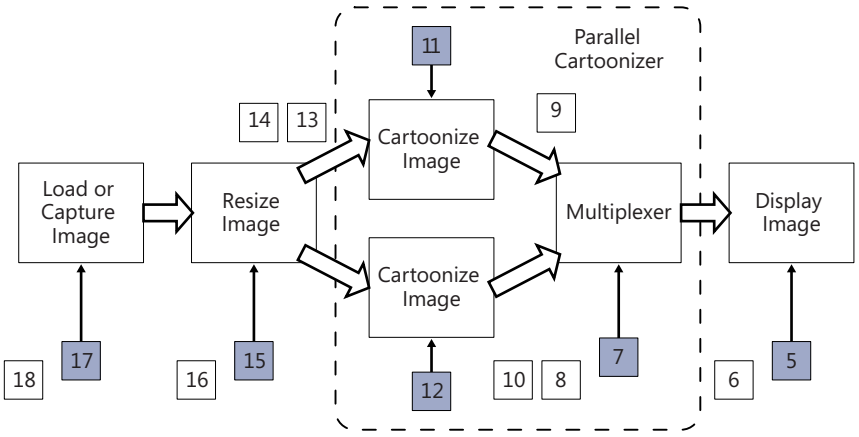
The *SwapEdges()* method uses the staging buffers and synchronous copies to swap the data.

```
void SwapEdges(array<ArgbPackedPixel, 2>* const top,
    array<ArgbPackedPixel, 2>* const bottom, UINT borderHeight)
{
    const UINT topHeight = top->extent[0];
    std::array<completion_future, 2> copyResults;

    copyResults[0] = copy_async(
        top->section(topHeight - borderHeight * 2, 0, borderHeight, m_width),
        m_swapViewTop);
    copyResults[1] = copy_async(
        bottom->section(borderHeight, 0, borderHeight, m_width), m_swapViewBottom);
    parallel_for_each(copyResults.begin(), copyResults.end(), [](completion_future& f)
        { f.get(); });

    copyResults[0] = copy_async(m_swapViewTop,
        bottom->section(0, 0, borderHeight, m_width));
    copyResults[1] = copy_async(m_swapViewBottom,
        top->section(topHeight - borderHeight, 0, borderHeight, m_width));
    parallel_for_each(copyResults.begin(), copyResults.end(), [](completion_future& f)
        { f.get(); });
}
```

The *SwapEdges()* method uses *async_copy()* to improve copying performance, just like the simpler example in the "Swapping Data Between Accelerators" section of Chapter 9. You can review the complete code for the *FrameProcessorAmpMulti* implementation in the FrameprocessorAmpMulti.h header file.

The Forked Pipeline

An alternative to dividing each image up and processing different portions on different GPUs is to process whole images on different GPUs while ensuring that their original ordering is maintained.



The diagram here shows two cartoonizing processors, one for each GPU. Each cartoonizer reads from a shared queue of images. This split-pipeline approach allows two images to be processed at the

same time. While the upper cartoonizer is processing image 11, the lower one is processing image 12. This approach supports a degree of dynamic load balancing, as discussed in the section of Chapter 9 entitled "Dynamic Load Balancing."

Each cartoonizer stage reads from a common queue of ordered images, but differences in the exact times taken to process an image might result in their being returned out of sequence. It's important to maintain the original ordering of the images so that each cartoonizer stage feeds its output into a multiplexer that ensures that the images are sent to the display stage in the correct order.

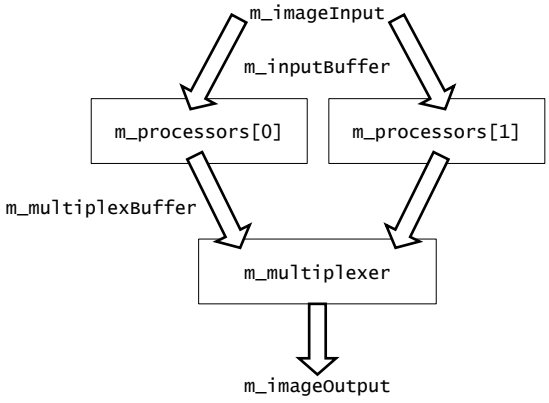
The *ImageCartoonizerAgentParallel* Class

The *ImageCartoonizerAgentParallel* class, defined in Pipeline\ImageCartoonizerAgentParallel.h, replaces this existing cartoonizer agent and implements the forked pipeline with one fork for each C++ AMP-capable accelerator.

```
class ImageCartoonizerAgentParallel : public ImageCartoonizerAgentBase
{
private:
    ISource<ImageInfoPtr>& m_imageInput;
    ITarget<ImageInfoPtr>& m_imageOutput;
    std::vector<std::shared_ptr<IFrameProcessor>> m_processors;
    unbounded_buffer<ImageInfoPtr> m_inputBuffer;
    int m_multiplexSequence;
    std::unique_ptr<call<ImageInfoPtr>> m_multiplexer;
    unbounded_buffer<ImageInfoPtr> m_multiplexBuffer;

    struct CompareImageInfoPtr
    {
        bool operator()(const ImageInfoPtr lhs, const ImageInfoPtr rhs) const
        {
            return (lhs->GetSequence() > rhs->GetSequence());
        }
    };

    std::priority_queue<ImageInfoPtr, std::vector<ImageInfoPtr>, CompareImageInfoPtr>
        m_multiplexQueue;

public:
    ImageCartoonizerAgentParallel(IImagePipelineDialog* const pDialog,
        FrameProcessorType processorType,
        ISource<bool>& cancellationSource, ITarget<ErrorInfo>& errorTarget,
        ISource<ImageInfoPtr>& imageInput, ITarget<ImageInfoPtr>& imageOutput) :
        ImageCartoonizerAgentBase(pDialog, cancellationSource, errorTarget),
        m_multiplexSequence(kFirstImage), m_processors(),
        m_imageInput(imageInput), m_imageOutput(imageOutput)
    {
        Initialize(processorType);
        m_imageInput.link_target(&m_inputBuffer);
    }
    // ...
```

```
private:
    void Initialize(FrameProcessorType processorType)
    {
        std::vector<accelerator> accels = AmpUtils::GetAccelerators();
        m_processors.resize(accels.size());
        std::transform(accels.cbegin(), accels.cend(), m_processors.begin(),
            [=](const accelerator& acc)->std::shared_ptr<IFrameProcessor>
        {
            return FrameProcessorFactory::Create(processorType, acc);
        });

        m_multiplexer = std::unique_ptr<call<ImageInfoPtr>>(new call<ImageInfoPtr>(
            [this](ImageInfoPtr pInfo)
            {
                if (pInfo == nullptr)
                {
                    asend<ImageInfoPtr>(m_imageOutput, nullptr);
                    return;
                }
                m_multiplexQueue.push(pInfo);
                while (m_multiplexQueue.empty() &&
                        (m_multiplexQueue.top()->GetSequence() == m_multiplexSequence))
                {
                    asend(m_imageOutput, m_multiplexQueue.top());
                    m_multiplexQueue.pop();
                    ++m_multiplexSequence;
                }
            }
        ));
        m_multiplexBuffer.link_target(m_multiplexer.get());
    }
```

The *Initialize()* method configures the additional pipeline elements required to fork the pipeline and adds a cartoonizer for each C++ AMP-capable GPU. This involves connecting the *m_imageInput* and *m_imageOutput* source and targets to an additional message block *m_inputBuffer* and adding another multiplexer stage to the pipeline, *m_multiplexer*.



The multiplexer itself uses a *std::priority_queue* of *ImageInfoPtr* with a *CompareImageInfoPtr* function object that orders the items by their sequence number. Regardless of the input order, images always leave the multiplexer ordered by their original sequence numbering.

The *run()* method is remarkably similar to the *ImageCartoonizerAgent::run()* method. The biggest change is that a PPL *parallel_for_each* is used to start up the *std::vector* of *IFrameProcessor* processors configured by the *Initialize()* method. Each frame processor is associated with a specific accelerator to ensure that it executes against that accelerator's *default_view*.

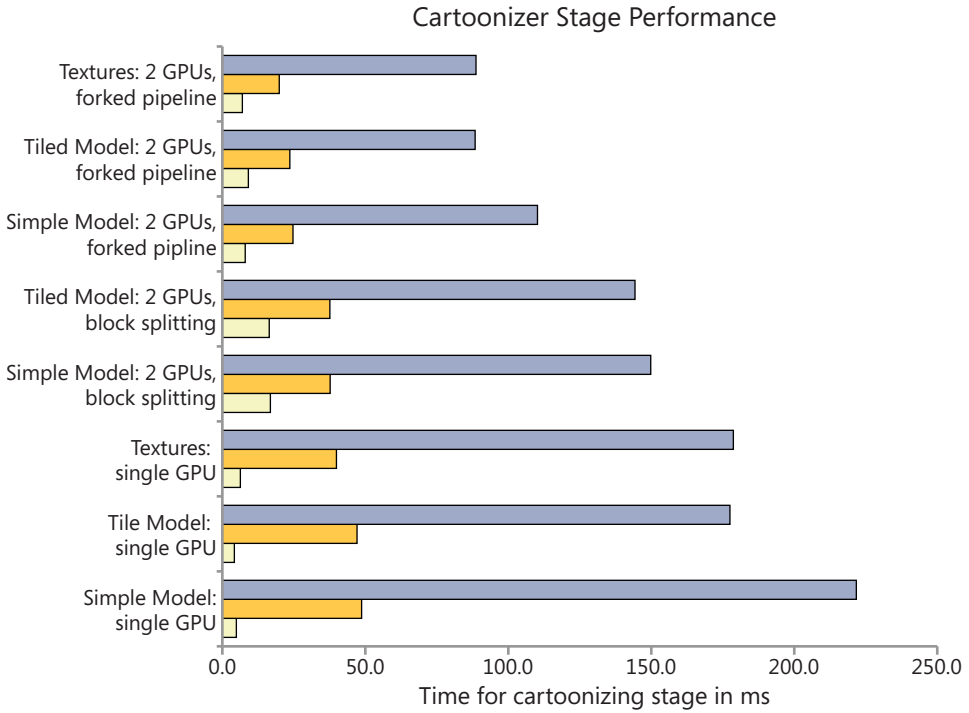
```
void run()
{
    parallel_for_each(m_processors.begin(), m_processors.end(),
        [=](std::shared_ptr<IFrameProcessor>& p)
    {
        ImageInfoPtr pInfo = nullptr;
        do
        {
            pInfo = receive(m_inputBuffer);
            CartoonizeImage(pInfo, p, m_dialogWindow->GetFilterSettings());
            send((pInfo == nullptr) ? m_inputBuffer : m_multiplexBuffer, pInfo);
        }
        while (nullptr != pInfo);
    });
    asend<ImageInfoPtr>(m_multiplexBuffer, nullptr);
    done();
}
```

All the *IFrameProcessor* instances are connected to the same *m_inputBuffer* and read items out of it on a first-come basis. This is the work-stealing approach described in the section of Chapter 9 entitled "Dynamic Load Balancing," and it supports load balancing across several GPUs.

The only other change is on shutdown. The pipeline is shut down by sending a single *nullptr* through the pipeline into *m_inputBuffer*. As each frame processor shuts down, it sends another *nullptr* into the *m_inputBuffer* to ensure that the next frame processor shuts down. Finally, the *parallel_for_each* completes and the agent sends its *nullptr* to the next pipeline stage before shutting down.

Cartoonizer Performance

The following graph and table show the performance of each frame processor when processing images using the minimum (1 simplifier phase and a border width of 1), the default (11 simplifier phases and a border width of six), and the maximum (32 simplifier phases and a border width of eight) frame processor settings. These measurements were taken on a Windows 8–based machine.



The texture-based frame processors listed here are discussed in detail in the "Using Textures and Short Vectors" section of Chapter 11, "Graphics Interop."

The table below shows the raw data for the cartoonizing stage, with additional columns showing the overall time/image for each frame processor.

	Minimum		Default		Max	
Frame processor	**Cartoonize**	**Time/ image**	**Cartoonize**	**Time/ image**	**Cartoonize**	**Time/ image**
CPU Single Core	291.1 ms	291.6 ms	39.2 s	39.2 s	189.9 s	189.9 s
CPU Multi Core	66.5 ms	66.7 ms	7.9 s	7.9 s	35.4 s	35.4 s
Simple Model: WARP	46.3 ms	46.9 ms	1.8 s	1.8 s	8.7 s	8.7 s
Tiled Model: WARP	48.5 ms	48.9 ms	1.7 s	1.7 s	8.1 s	8.1 s
Simple Model: single GPU	4.9 ms	30.1 ms	48.7 ms	49.0 ms	221.7 ms	222.1 ms
Tiled Model: single GPU	4.2 ms	30.3 ms	47.1 ms	47.4 ms	177.5 ms	177.9 ms
Textures: single GPU	6.3 ms	28.6 ms	39.9 ms	40.2 ms	178.7 ms	179.2 ms
Simple Model: 2 GPUs, block splitting	16.8 ms	30.5 ms	37.7 ms	37.9 ms	149.8 ms	150.5 ms
Tiled Model: 2 GPUs, block splitting	16.4 ms	31.2 ms	37.6 ms	37.8 ms	144.3 ms	144.5 ms

	Minimum		Default		Max	
Frame processor	**Cartoonize**	**Time/ image**	**Cartoonize**	**Time/ image**	**Cartoonize**	**Time/ image**
Simple Model: 2 GPUs, forked pipeline	8.0 ms	26.2 ms	24.7 ms	25.7 ms	110.2 ms	110.4 ms
Tiled Model: 2 GPUs, forked pipeline	9.1 ms	27.5 ms	23.6 ms	23.9 ms	88.4 ms	89.1 ms
Textures: 2 GPUs, forked pipeline	7.0 ms	25.6 ms	19.9 ms	20.1 ms	88.7 ms	89.7 ms

A number of observations can be made about the results in the table. Remember that the timing results will vary depending on your computer's hardware configuration.

The C++ AMP implementations of the cartoonizer algorithm show significant improvements over the original CPU implementations. The CPU implementations have not been heavily optimized, so you shouldn't make direct comparisons between GPU and CPU performance.

The throughput of a pipeline is limited by the time taken by the slowest stage, so the display stage of the pipeline limits the time/image to approximately 20 ms. As the cartoonizer stage's performance improves, the display stage becomes the limiting factor. You can see this in the results for the default settings; the time/image never falls below 20 ms regardless of the time taken to cartoonize the image. A large portion of the time taken to display the image is in the call to the GDI *BitBlt()* function in the *CartoonizerDlg::OnPaint()* method.

One possible approach to improving the display stage performance would be to use DirectX graphics to draw the image directly from GPU memory, which would also remove the need to copy the image data back into CPU memory after cartoonizing the image with C++ AMP. This isn't shown here because it adds further complexity to the code. Chapter 11 uses the NBody case study as an example of how to use Direct3D in conjunction with C++ AMP to render the results of your computations. It also discusses how to reimplement the cartoonizing frame processor using the C++ AMP *texture* type instead of *array*.

For the minimum frame processor settings (1 simplifier phase and a border width of 1), there is insufficient shared memory access to take advantage of tiled memory. This is clearly shown by comparing the times taken by the cartoonizing stage for the C++ AMP simple model (4.9 ms) and the tiled model (4.2 ms) running on a single GPU. You would expect the tiled implementation to execute more quickly, but it's comparable. For the default and maximum frame processor settings, tiled memory becomes more beneficial and the tiled model processors execute faster than the simple model ones.

The *FrameProcessorAmpMulti* implementation shows some improvement over the *FrameProcessorAmpSingle* implementation. The multi-GPU cartoonizer has additional overhead for splitting the work of processing a single image across multiple GPUs and coordinating updates of halo data after each step of the cartoonizing process. In contrast, the *ImageCartoonizerAgentParallel* implementation shows a nearly 100 percent parallelization efficiency when running on two GPUs. The code is also much simpler; the parallel cartoonizer uses the same cartoonizing code as the single-GPU version with a modified pipeline.

Summary

The Cartoonizer sample shows how to leverage the power of both the CPU and available C++ AMP-capable GPUs to get the best performance from all of the available hardware. This braided parallelism approach allows the application to get the most out of the available hardware by using the CPU to control the image processing pipeline and by using the CPU and available GPUs to execute the individual pipeline stages. Cartoonizer demonstrates several other features:

- It allows the user to choose the hardware configuration to use at run time to achieve the best performance. Your application can do this automatically at startup.
- The tiled image processing functions use padding to support images whose dimensions are not an exact multiple of the tile size. Using padding to handle compute domains that do not exactly map to the tile size is discussed in Chapter 12.
- It includes a texture-based frame processor implementation. This is explained in detail in Chapter 11.
- If your application has more than one GPU available to it, it shows different load balancing strategies.
- The forked pipeline implements dynamic load balancing across the available GPUs, which will allow the pipeline to deal with variability in performance due to other processes running other work on the GPUs or differences in image size.

The CPU and GPU implementations of the cartoonizing algorithm are quite similar. The main difference is that the GPU requires some additional type conversion to pack and unpack the data as it's moved to and from the GPU. This reduces the amount of data that must be copied and therefore improves overall performance. It also moves the inherently data-parallel work of converting the data for each pixel in the image to the GPU, allowing the application to get further performance gains over doing this work on the CPU.

Chapter 11 discusses an additional cartoonizing frame processor that uses the C++ *texture* type to take advantage of GPU hardware support for 2D spatial locality and automatic packing and unpacking of data.

CHAPTER 11

Graphics Interop

In this chapter:

Fundamentals 257
Using Textures and Short Vectors 271
HLSL Intrinsic Functions 274
DirectX Interop 275
Summary 282

So far, this book has not discussed graphics or DirectX in any detail. One of the powerful features of C++ AMP is that the application programming interface (API) removes the need for programmers to think about their programs in terms of graphics primitives. This has two advantages: it allows you to write code that reflects your problem domain rather than the underlying hardware, and it supports portability by not tying the implementation directly to a particular run time. Currently, C++ AMP is implemented only on DirectCompute, but other vendors might choose to support it on other run times.

However, sometimes it's useful to interact with the GPU more directly—for example, feeding results of a C++AMP calculation directly into the DirectX graphics pipeline or taking advantage of specific GPU features like texture memory or intrinsic functions. This chapter covers these additional features of C++ AMP.

Fundamentals

C++ AMP provides several types specifically for exposing features that are supported by the Direct3D High Level Shader Language (HLSL) or are commonly used when programming graphics-related applications. These additional types are defined in the *concurrency::graphics* and *concurrency::direct3d* namespaces, declared in amp_graphics.h. They were not covered in Chapter 3, "C++ Fundamentals." That chapter described only the types defined in the *concurrency* namespace. This section describes each of these graphics-related types and describes how the Cartoonizer case study was extended to take advantage of these features.

norm and *unorm*

The *norm* and *unorm* classes are wrappers over *float* that implement clamping behavior, similar to the HLSL *snorm float* and *unorm float* scalar types. They can be used in both amp-restricted and cpu-restricted code. The *norm* type limits the range of float values to [-1.0, 1.0], while *unorm* limits the range to [0.0, 1.0]. The *norm* and *unorm* types are typically used for image processing, and several pixel formats are based on them—for example, *R8G8B8A8_UNORM* and *R8G8B8A8_SNORM*. Your program would more likely use these types as part of a preprocessing or postprocessing phase in a computation that fed its output directly into the Direct3D pipeline.

For more details on the supported DirectX types that make use of the corresponding HLSL types *float unorm* and *float snorm*, see the "DXGI_FORMAT enumeration" topic on MSDN: *http://msdn.microsoft.com/en-us/library/windows/desktop/bb173059(v=vs.85).aspx*.

Although *norm* and *unorm* are wrappers over *float*, unlike a *float,* their default constructor initializes them to 0.0. The following declaration is an example:

```
unorm val1;
```

This will result in *val1* being initialized to 0.0. As you would expect, initializing a *norm* or *unorm* to a value outside of its range results in the value being clamped. For example, here *val2* is initialized to 1.0 and *val3* to 0.0:

```
norm val2(2.0f);
unorm val3(-2.0f);
```

You can also construct a *norm* or *unorm* from a nonfloat value. In this case, it is first cast to a *float* and then clamped. Here val4 is initialized to 1.0:

```
unorm val4(2u);
```

Both *norm* and *unorm* provide the same operators and copy constructors as *float*. Implicit casts also exist for *norm* and *unorm* to *float* and for *unorm* to *norm*. The implementation performs the operation as a *float* and then clamps the result. Here *val5* is set to 1.0:

```
float val5 = norm(0.25f) + unorm(1.5f);
```

In the case above, the *unorm* is implicitly converted to a *norm*, then the two *norm* values are added, and finally the resulting *norm* is cast to a *float*.

Note that *norm* supports the negation operator but *unorm* does not. Here, both *val6* and *val7* are initialized to -0.25 but *val6* is of type *norm*, whereas val7 is a *float* because *unorm* is promoted to a *float* and then negated:

```
auto val6 = -norm(0.25f);
auto val7 = -unorm(0.25f);
```

You can find the declarations of *norm* and *unorm* in amp_short_vectors.h, which is included by amp_graphics.h. The same header also contains some useful macro definitions for the zero and clamped minimum and maximum values of *norm* and *unorm*.

Macro	Value
NORM_ZERO	norm(0.0f)
NORM_MIN	norm(–1.0f)
NORM_MAX	norm(1.0f)
UNORM_ZERO	unorm(0.0f)
UNORM_MIN	unorm(0.0f)
UNORM_MAX	unorm(1.0f)

Short Vector Types

C++ AMP also provides types that implement similar behavior to the HLSL vector types. These are primarily provided for interop with Direct3D pipeline, but they are useful in their own right if your computation requires vector math. For example, the NBody case study uses *float_3* to store position and velocity data for each particle.

The following types are declared in the *concurrency::graphics* namespace and follow the naming convention *ScalarType_Rank*. Like *norm* and *unorm*, they can be used in both amp-restricted and cpu-restricted code.

uint_2	int_2	float_2	double_2	unorm_2	norm_2
uint_3	int_3	float_3	double_3	unorm_3	norm_3
uint_4	int_4	float_4	double_4	unorm_4	norm_4

The graphics namespace also adds a typedef *uint* for unsigned int. Remember that support for *double_N* types is dependent on the accelerator hardware. See the "Double-Precision Support" section in Chapter 12, "Tips, Tricks, and Best Practices," for more details.

C++ AMP also declares typedefs in the *concurrency::graphics::direct3d* namespace for each type without the underscore: *float3*, *int4*, and so on. These alternative type names are provided for graphics programmers who are used to this naming convention in HLSL. If you just want to use the vector types for computation, use the types in the *concurrency::graphics* namespace.

Each short vector type supports the following operators:

Operators	uint_N	int_N	float_N	Double_N	unorm_N	norm_N
Arithmetic operators: +, -, *, /	×	×	×	×	×	×
Bitwise operators: %, ^, |, &, <<, >>, ~	×	×				
Compound arithmetic assignment operators: +=, -=, *=, /=	×	×	×	×	×	×
Compound bitwise assignment operators: %=, ^=, |=, &=, <<=, >>=	×	×				
Equality operators: ==, !=	×	×	×	×	×	×

Operators	uint_N	int_N	float_N	Double_N	unorm_N	norm_N
Increment and decrement operators: ++, -- (prefix and postfix)	×	×	×	×	×	×
Unary negation operator: -		×	×	×		×

All operations are applied to each component of the vector. So *uint_2(1, 2) == uint_2(1, 2)* evaluates to *true* and *++uint_2(1, 2)* evaluates to *uint_2(2, 3)*.

For more information on HLSL's vector types, see the "Data Types" section in the HLSL Reference on MSDN: *http://msdn.microsoft.com/en-us/library/windows/desktop/bb509587(v=vs.85).aspx*.

Accessing Vector Components

Rather than accessing individual vector components using the subscript operator, short vector types use named properties of the vector. Two sets of properties are provided: one for 4D spatial vectors and a second for colors.

Component	Spatial properties	RGBA color properties
0	x	r
1	y	g
2	z	b
3	w	a

Your program can access individual members of a vector using either the spatial or color space accessors. You can also collapse a vector from N to M dimensions by selecting individual components. In the following example, an *int_4* vector is converted to an *int_2* and its *r* (first) and *b* (third) components are also reordered. The result is *int_2(3, 1)*.

```
int_4 vec4(1, 2, 3, 4);
int_2 vec5 = vec4.br;
```

In computer graphics, this form of vector component access and reordering is referred to as swizzling.

Template Metaprogramming

Each short vector exposes its rank and type as two static properties, *size* and *value_type*, respectively. For example, *uint_2* is declared in amp_short_vectors.h.

```
class uint_2
{
public:
    typedef uint value_type;
    static const int size = 2;
    // ...
};
```

These are also exposed on a separate *short_vector_traits* template declared for each short vector type and for the corresponding scalar types. For example, here are the traits for *uint_2* and *unsigned int*:

```
template<>
struct short_vector_traits<uint_2>
{
    typedef unsigned int value_type;
    static int const size = 2;
};

template<>
struct short_vector_traits<unsigned int>
{
    typedef unsigned int value_type;
    static int const size = 1;
};
```

A class, *short_vector*, is also provided for generically programming short vectors. Each template instance declares a single typedef, *type*.

```
template<>
struct short_vector<unsigned int, 2>
{
    typedef uint_2 type;
};
```

You can use these to create generic methods for all short vectors. For example, the following uses *short_vector_traits<T>* and *short_vector<T, 2>* to declare a vector *length()* function that can be applied to all short vectors:

```
//  The length() function for N > 1.

template<typename T>
inline static typename std::enable_if<(short_vector_traits<typename T>::size > 0), float>::type
    length(const T& vec) restrict(cpu, amp)
{
    return length_helper<short_vector_traits<typename T>::value_type,
        short_vector_traits<typename T>::size>::length(vec);
}

//  Template specializations for ScalarType_N short vectors.

template<typename ScalarType, int N>
class length_helper
{
public:
    inline static float length(const typename short_vector<ScalarType, N>::type& vec)
        restrict(cpu, amp)
    {
        static_assert(false, "length() is not supported for this type.");
    }
};
```

```
template<typename ScalarType>
class length_helper<ScalarType, 1>
{
public:
    inline static float length(const typename short_vector<ScalarType, 1>::type& vec)
        restrict(cpu, amp)
    {
        return static_cast<float>(vec);
    }
};

template<typename ScalarType>
class length_helper<ScalarType, 2>
{
public:
    inline static float length(const typename short_vector<ScalarType, 2>::type& vec)
        restrict(cpu, amp)
    {
        return fast_math::sqrtf(static_cast<float>(vec.x * vec.x + vec.y * vec.y));
    }
};
// ...
```

The full implementation of the *length()* function can be found in Chapter11\AmpVectorUtils.h in the Chapter 11 sample, Chapter11\Chapter11.sln. This also includes additional overloads of the *length()* function for vectors of length three and four and for *double_N* types that return a *double* rather than casting it to a *float* with the corresponding loss of precision. The AmpStreamUtils.h header also includes a further example showing how to provide a *wostream* output operator for short vector types.

texture<T, N>

GPUs were originally designed to support rendering graphics, so it should come as no surprise that their hardware supports many features that were designed for that purpose. Textures are one of these features, and C++ AMP implements a *texture<T, N>* type to allow your application to take advantage of them. It is declared in amp_graphics.h. This section explains the *texture<T, N>* type and how to use it.

Data Storage

Textures are N-dimensional containers of individual texture elements called texels of type T. N is limited to 1, 2, or 3. A texel is composed of between one and four components (also called scalar elements). Texels can hold a limited number of scalar and short vector types.

- The texture's texel can hold only the following scalar types: *int*, *uint*, *float*, *double*, *norm*, and *unorm* as a single component texel.

- The texture's texel can hold short vector types that have two or four components—for example, *int_2*, *int_4* and *float_4*. Short vectors of double are the exception; only *double_2* is supported. A texel can't hold four *double* values. Although Direct3D supports some three-component texel types, C++ AMP does not.

Texels store their data as packed bit fields, and the GPU hardware automatically handles extracting the packed data into the required type. The packing is described in terms of the number of bits per scalar element in the texel. The bits per scalar element property describes the number of bits used for each of the stored values. For example, an *int_4* stored with 8 bits per scalar element would mean a 32-bit texel holding four int values, each using 8 bits. Only certain values for bits per scalar element make sense. The following table lists the default and supported *bits_per_scalar_element* values for each texel type:

Texel type	Allowed values for bits per scalar element	Default bits per scalar element
int, uint, int_2,uint_2, int_4,uint_4	8, 16, 32	32
float, float_2, float_4	16, 32	32
double, double_2	64	64
norm, unorm, norm_2,unorm_2, norm_4,unorm_4	8, 16	No default; must be specified.

It's possible to set the bits per scalar element by setting the *_Bits_per_scalar_element* parameter on the texture constructor. The *texture<T, N>::bits_per_scalar_element* read-only property holds the bits per scalar element for the texture instance. If your program attempts to create a texture with an unsupported combination of texel type and bits per scalar element value, a *runtime_exception* will be thrown.

Creating a texture with a lower bits per scalar element value than is usually used to store the texture's scalar element type results in a lower range of values being stored or a lower precision. For example, creating a texture of *int* with 16 bits per scalar element results in a value that is still retrieved from the texture as an *int*. However, the *int* holds the range of values of a *short*. For *float* values, a lower level of precision is used. For example, storing a *float* with a bits per scalar element of 16 stores the value as a half precision floating-point number defined by IEEE-754-2008 as *binary16* format.

The *norm* and *unorm* types, which were introduced in the previous section, store their values within the texture as fixed-point numbers (see *http://en.wikipedia.org/wiki/Fixed-point_arithmetic*). For these types, the bits per scalar element affects the number of discrete values representing the range stored by the type. For example, take the hypothetical case where a *unorm* value is stored with four bits per scalar element. The binary values are mapped to unique, evenly spaced floating-point values within the range [0.0, 1.0].

Binary value	0000	0001	0010	0011	0100	0101	0110	0111
Float value	0.0	1/15	2/15	3/15	4/15	5/15	6/15	7/15

Binary value	1000	1001	1010	1011	1100	1101	1110	1111
Float value	8/15	9/15	10/15	11/15	12/15	13/15	14/15	1.0

This shows the loss of precision that would result if *norm(0.500f)* were stored in this four bits per element type. The value would be rounded accordingly to 8/15, or $0.5\overline{3333}$ with the corresponding loss of precision.

You can use the flexibility provided by being able to set the bits per scalar element of a texture to reduce the amount of memory required to store your program's data when lower levels of precision are acceptable. It also allows you to handle data types that are not natively supported by C++ AMP; this chapter includes examples of kernels that manipulate byte arrays and 32-bit RGBA data.

Copying Data to and from Textures

Your program can use the various *texture* constructor overloads to initialize the instance data. C++ AMP supports two approaches, each with several overloads for different extent ranks: iterator-based initialization and pointer-based initialization.

Your program can provide a pair of iterators to specify the initialization data.

```
const int cols = 32;
const int rows = 64;
std::vector<uint> uintData((rows * cols), 1);
texture<uint, 2> text1(rows, cols, uintData.cbegin(), uintData.cend());
```

If iterators are used, the default value for the bits per scalar element for the texture type is used and can't be set as a constructor parameter. In addition, it's not possible to construct textures of *norm*, *unorm* or *norm_N*, and *unorm_N* short vectors in this way because these types do not have a default *bits_per_scalar_element* value.

There are also overloads that allow you to pass a void pointer, the size of the data, and the bits per scalar element.

```
uint bitsPerScalarElement = 8u;
uint dataSize = rows * cols;
std::vector<char> byteData((rows * cols), 1);
texture<uint, 2> text2(rows, cols, byteData.data(), dataSize, bitsPerScalarElement);
```

These constructors allow you more flexibility when loading data, especially mapping types like *char* that are not natively supported.

Constructing an uninitialized texture associated with a specific *accelerator_view* is also straightforward:

```
accelerator acc = // ...
const texture<int, 2> text0(rows, cols, acc.default_view);
```

The example could also have specified the bits per scalar element. For example, 8 bits:

```
const texture<int, 2> text0(rows, cols, 8u, acc.default_view);
```

Using 8 bits per scalar element means that although the values stored in the texture are retrieved as *int*, they are stored as 8-bit values. So they are limited to the range [-128, 127].

The texture type supports numerous other constructor overloads. For full details, see the "Texture Class" topic on MSDN: *http://msdn.microsoft.com/en-us/library/hh537953(v=vs.110)*.

C++ AMP also provides copy methods for moving data from CPU memory to textures. For example, in the previous code, the texture data could have been copied in a separate step.

```
texture<uint, 2> text3(rows, cols, bitsPerScalarElement);
copy(uintData.data(), dataSize, text3);
```

The example could also be rewritten wrapping *text3* in a *writeonly_texture_view* and then that data could be copied into the view. As the name suggests, the *writeonly_texture_view* is similar to the *array_view* for *array* but supports only write access. It is covered in more detail later in the chapter.

```
writeonly_texture_view<uint, 2> textVw3(text3);
copy(uintData.data(), dataSize, textVw3);
```

Additional overloads also exist for copying texture data back into CPU memory. For example, the data in *text3* can be copied back as follows:

```
copy(text3, byteData.data(), dataSize);
```

There is no *copy()* method for copying a *writeonly_texture_view* because it can only be written to. In this case, you should copy the underlying *texture* data.

In addition to the *copy()* methods, corresponding *copy_async()* methods are also implemented. These provide asynchronous functionality and return a *completion_future* that can be waited on or associated with a continuation. For example, the previous copy example could have been written with *copy_async()*.

```
completion_future f = copy_async(text3, byteData.data(), dataSize);

// Do other work...

f.then([=](){ std::wcout << "Copy complete" << std::endl; });
```

The *completion_future::then()* method can be used to specify a task that gets executed on completion of the asynchronous method. This task is scheduled on another thread, so the calling thread is not blocked. Alternatively, *completion_future::get()* could have been used to wait for the task to complete and execute subsequent work on the calling thread.

```
// Do other work...

f.get();
std::wcout << "Copy complete" << std::endl;
```

For a complete list of all the available *copy()* and *copy_async()* methods, see the "Concurrency Namespace (C++ AMP)" topic on MSDN: *http://msdn.microsoft.com/en-us/library/hh305267(v=vs.110)*.

In addition to *copy()* and *copy_async()*, the *texture<T, N>::copy_to()* method also provides two overloads for copying a *texture* to another *texture* or a *writeonly_texture_view*. For example, you could copy *text3* to a new *texture*, *text4*.

```
texture<uint, 2> text4(rows, cols, bitsPerScalarElement);
text3.copy_to(text4);
```

The source and target must have the same *bits_per_scalar_element* and *extent* but do not have to be on the same accelerator view.

Reading from Textures

Here's the simplest possible program using *texture*. It initializes a two-dimensional *texture* of *int* from data in a *std::vector<int>*and uses a *parallel_for_each* to copy the *texture* data into an *array_view<int>*. It also displays the *texture* properties. The *parallel_for_each* captures the *texture* by reference, just like an *array*.

```
const int cols = 32;
const int rows = 64;
std::vector<int> input((rows * cols), 1);

const texture<int, 2> inputTx(rows, cols, input.cbegin(), input.cend());
std::vector<int> output((rows * cols), 0);
array_view<int, 2> outputAv(rows, cols, output);
outputAv.discard_data();

parallel_for_each(outputAv.extent, [&inputTx, outputAv](index<2> idx) restrict(amp)
{
    outputAv[idx] = inputTx[idx];             // subscript [index<2>] operator
});
std::wcout << "extent:      (" << inputTx.extent[0] << ", " << inputTx.extent[1] << ")"
    << std::endl;
std::wcout << "size:        " << inputTx.data_length << std::endl;
std::wcout << "BPSE:        " << inputTx.bits_per_scalar_element << std::endl;
std::wcout << "accelerator: " << inputTx.accelerator_view.accelerator.description
    << std::endl;
```

This program displays the following output:

```
extent:      (64, 32)
size:        8192
BPSE:        32
accelerator: NVIDIA GeForce GTX 570
```

Like *array<T, N>*, *texture<T, N>* has an extent and is associated with an *accelerator_view*. These can be accessed through the read-only *extent* and *accelerator_view* properties. Note that the bits per scalar element of the texture is set to the default value for *int*, which is 32.

Textures also support an additional *texture<T, N>::get()* method. In the example, the assignment instruction could have been replaced by one of the following:

```
outputAv[idx] = inputTx(idx);                  // function (index<2>) operator
outputAv[idx] = inputTx.get(idx);              // get(index<2>) method
outputAv[idx] = inputTx(idx[0], idx[1]);       // function (int, int) operator
```

The function and subscript operators and the *get()* method are all amp-restricted, so you can't use them to access *texture* data from the CPU. Your application can also use the flexibility provided by a configurable *bits_per_scalar_element* property to access data that would otherwise not be easily loaded onto an accelerator because the underlying data type is not supported. In the following example, a vector of *char* is loaded into a *texture* with 8 bits per scalar element and then each individual 8-bit *char* value is accessed as an *int*:

```
const UINT bitsPerScalarElement = 8u;
const int size = 1024;
std::vector<char> input(size, 'a');

const texture<int, 1> inputTx(size, input.data(), size, bitsPerScalarElement);
std::vector<int> output(size, 0);
array_view<int, 1> outputAv(size, output);
outputAv.discard_data();

parallel_for_each(outputAv.extent, [&inputTx, outputAv](index<1> idx) restrict(amp)
{
    int element = inputTx[idx];
    outputAv[idx] = element; // Calculate using 8-bit element value holding a char value
});
```

The Cartoonizer case study, discussed in the section of this chapter entitled "Using Textures and Short Vectors," shows further examples of using the texel components to access individual RGBA elements of an image.

Writing to Textures

The previous example showed only how to read from textures. Unlike *array*, which has [] subscript and () function operators that return a reference to the element, the same operators on *texture* return a value. This means that you can't write by means of the [] operator. The *texture::set()* method is used for setting *texture* values. The following example reads data from an *array_view* and writes it to a *texture*:

```
const int cols = 32;
const int rows = 64;
std::vector<int> input((rows * cols), 1);

texture<int, 2> outputTx(rows, cols, input.cbegin(), input.cend());
array_view<int, 2> inputAv(rows, cols, input);

parallel_for_each(outputTx.extent, [inputAv, &outputTx](index<2> idx) restrict(amp)
{
    outputTx.set(idx, inputAv[idx]);
});
```

The *set()* method is amp-restricted, so you can use it only to update texture data within a C++ AMP kernel.

Given that a texture might have a *bits_per_scalar_element* setting that is insufficient for storing the value, the accelerator hardware implements the following clamping behavior when writing data:

- Integer values are clamped to a value between the [min, max] values that can fit in the available storage bits. This differs from the behavior in CPU code where integer overflows result in a reduction modulo a power of two, causing the number to wrap around. For example, attempting to set 130 on an *int* texel with 8 bits per scalar element results in 127 being stored. In contrast, the same operation on a char value on the host would result in a value of -126 due to wrapping.
- Float values are clamped to a value between the [min, max] values that can fit in the available storage bits. Values that have already overflowed or underflowed are stored as overflowing or underflowing values; they are not clamped. For example, setting a float texel with 16 bits per scalar element to *FLT_MAX* (max *float* value defined in float.h) results in clamping and in 65504.0, the max value for an IEEE *binary16* value being stored. When storing a value that has already overflowed, NaN, the overflow is stored and no clamping occurs.
- The *norm* and *unorm* types are clamped to their appropriate ranges. Assigning 2.0f to an 8 bits per scalar element *norm* results in a 1.0f value being stored.

Read-Write Textures

In most cases, a *texture<T, N>* is read-only, or write-only if accessed through a *writeonly_texture_view*. Read-write textures are supported only in the following cases:

- The texel type, T, is; *int*, *uint*, or *float*. This is checked at compile time.
- This *bits_per_scalar_element* is 32. This is checked at run time and causes an *unsupported_feature* exception to be thrown if your code both reads and writes to a *texture* with *bits_per_scalar_element != 32*.

If these criteria aren't met, the *texture* is read-only.

The *texture* in the previous example satisfies these requirements, so the *parallel_for_each* can be modified to read and write each element, incrementing them.

```
parallel_for_each(outputTx.extent, [&outputTx](index<2> idx) restrict(amp)
{
    outputTx.set(idx, outputTx[idx] + 1);
});
```

However, if the example had used *texture<int_2>*, the criteria above would not be satisfied.

writeonly_texture_view<T, N>

The previous section explained the limitations of both reading and writing to textures. C++ AMP provides the *write_only_texture_view<T, N>* type to allow writes and make this limitation explicit. In the following example, it's not possible to write to *text1* directly because it contains elements type *int_2*. In this case, wrapping the texture in a *writeonly_texture_view* allows your application to write to it.

```
const int cols = 32;
const int rows = 64;

texture<int_2, 2> text1(rows, cols);
writeonly_texture_view<int_2, 2> textVw(text1);
parallel_for_each(textVw.extent, [textVw] (index<2> idx) restrict(amp)
{
    textVw.set(idx, int_2(1, 1));
});
```

It's still not possible to both read and write to *text1*, but the use of *writeonly_texture_view* supports writes and makes it clear in the code that reads are not possible.

Note that a *writeonly_texture_view* is captured by value, just like an *array_view*. It's also possible to create a *writeonly_texture_view* within amp-restricted code, provided that the texture it wraps contains types that meet the requirements for a read-write texture. The example in the previous section could have been rewritten as:

```
parallel_for_each(outputTx.extent, [&outputTx](index<2> idx) restrict(amp)
{
    // outputTx.set(idx, outputTx[idx] + 1);
    writeonly_texture_view<int, 2> outputTxVw(outputTx);
    outputTxVw.set(idx, outputTx[idx] + 1);
});
```

Note that the following is not possible and results in a *runtime_exception* being thrown:

```
std::vector<int> input((rows * cols), 1);
texture<int, 2> text2(rows, cols, input.data(), input.size() * sizeof(int), 32u);
writeonly_texture_view<int, 2> outputTxVw(text2);
parallel_for_each(outputTxVw.extent, [outputTxVw, &text2](index<2> idx) restrict(amp)
{
   outputTxVw.set(idx, text2[idx] +1);
});
```

The kernel fails because texture aliasing is not supported; *text2* and *outputTxVw* refer to the same texture instance and would require aliasing. Aliasing is discussed in detail in Chapter 7, "Optimization."

Textures vs. Arrays

In some respects, the *texture<T, N>* type is similar to the familiar *array<T, N>* introduced in Chapter 3. However, the following are the important differences:

- A texture's rank, N, is limited to values one, two, or three.
- A texture can hold a limited number of types.
- Although the *texture* constructor allows you to pass an optional *accelerator_view* parameter, textures can't be instantiated on the *cpu_accelerator*.
- Textures are limited in their size based on their rank.

Rank	Maximum size in each dimension
texture<T, 1>	16,384
texture<T, 2>	16,384
texture<T, 3>	2,048

  Attempting to create a texture that exceeds these limits will result in a *runtime_exception*.

- Textures are not always readable and writable. In most cases, your C++ AMP kernel can either read data from, or write data to, a *texture* instance but not do both.
- Textures can store scalar elements with different numbers of bits per scalar element. These values are automatically packed and unpacked by GPU hardware.
- Textures have additional *data_length* and *bits_per_scalar_element* properties that return the size in bytes of the data stored by the texture and the number of bits per scalar element.

When should you use an *array* and when should you use a *texture*? A texture might be the obvious choice when interop is required or for subword packing of data. However, in more general compute scenarios, the answer is less clear-cut and the relative performance of the two approaches is very application-specific and hardware-specific. The following guidelines might be helpful:

- If your algorithm makes good use of coalesced global memory or can take advantage of *tile_static* memory—or both—then using textures is unlikely to result in any major performance gains.
- Most GPU hardware has a texture cache optimized for 2D spatial locality. If your algorithm can make use of this 2D locality and is unable to use coalesced memory accesses, then you might see some performance improvements in moving to textures.
- Textures are unlikely to improve performance if your algorithm does not reuse the data read from global memory into the texture.
- When experimenting with a texture implementation, you should try to compare it with an array-based implementation to see the relative performance. The following section shows an example of this using the Cartoonizer case study introduced in Chapter 10, "Cartoonizer."

Using Textures and Short Vectors

The simple examples shown so far in this chapter do not show the full power and flexibility of textures and short vectors. The Cartoonizer case study in Chapter 10 processes bitmaps stored in a RGBA32 format. Each pixel's red, green, blue, and alpha values are stored as 8-bit values packed into a single 32-bit value, *ArgbPackedPixel*:

```
typedef unsigned long ArgbPackedPixel;
```

The individual 8-bit pixel values must be unpacked and packed into this 32-bit value using additional C++ AMP code.

This section explains how the Cartoonizer case study sample can be extended to use *texture<uint_4, 2>* to store the data with eight bits per scalar element. In this case, the result brings some performance improvements and significantly less code because the texture takes care of the data transformations.

In the *array*-based implementation, the image processing methods must first unpack each pixel in the *srcFrame* Bitmap image into an *RgbAmp* structure. This can then be used for image processing. The *RgbPixel* type is declared in RgbPixel.h along with the packing and unpacking functions.

```
struct RgbPixel
{
    unsigned int r;
    unsigned int g;
    unsigned int b;
};

const int fixedAlpha = 0xFF;

inline ArgbPackedPixel PackPixel(const RgbPixel& rgb) restrict(amp)
{
    return (rgb.b | (rgb.g << 8)  | (rgb.r << 16) | (fixedAlpha << 24));
}

inline RgbPixel UnpackPixel(const ArgbPackedPixel& packedArgb) restrict(amp)
{
    RgbPixel rgb;
    rgb.b = packedArgb & 0xFF
    rgb.g = (packedArgb & 0xFF00) >> 8;
    rgb.r = (packedArgb & 0xFF0000) >> 16;
    return rgb;
}
```

These additional *Unpack()* and *Pack()* functions are required to convert the *ArgbPackedPixel* elements in *srcFrame* into *RgbPixel* elements in *currentImg* and *originalImg*. This code was discussed in the section of Chapter 10 entitled "The FrameProcessorAmpSingle Class." You can compare the two implementations by referring back to the code in that section.

Reimplementing the existing algorithm to use textures is relatively simple. The new *FrameProcessorAmpTextureSingle* frame processor class stores its data as textures and also declares a *writeonly_texture_view* to allow new image data to be written into an existing texture on each new frame processing request, rather than creating a new texture from scratch.

```
std::array<std::shared_ptr<texture<uint_4, 2>>, 3> m_frames;
std::unique_ptr<writeonly_texture_view<uint_4, 2>> m_originalFrameView;
// ...

void FrameProcessorAmpTextureSingle::ConfigureFrameBuffers(const Gdiplus::BitmapData& srcFrame)
{
    if ((m_height == srcFrame.Height) && (m_width == srcFrame.Width))
        return;
    m_height = srcFrame.Height;
    m_width = srcFrame.Width;

    std::generate(m_frames.begin(), m_frames.end(), [=]()->std::shared_ptr<texture<uint_4, 2>>
    {
        return std::make_shared<texture<uint_4, 2>>(int(m_height), int(m_width), 8u,
            m_accelerator.default_view);
    });
    m_originalFrameView = std::unique_ptr<writeonly_texture_view<uint_4, 2>>(
        new writeonly_texture_view<uint_4, 2>(*m_frames[kOriginal].get()));
}
```

The new *ProcessImage()* implementation uses *copy()* and *copy_to()* to move the data to and from the GPU. The *texture<uint_4, 2>* data stored in *m_frames* is initialized directly from the bitmap data. The texture hardware automatically handles any packing.

```
void FrameProcessorAmpTextureSingle::ProcessImage(const Gdiplus::BitmapData& srcFrame,
    Gdiplus::BitmapData& destFrame, UINT phases, UINT simplifierNeighborWindow)
{
    assert(simplifierNeighborWindow % 2 == 0);
    assert(phases > 0);

    ConfigureFrameBuffers(srcFrame);

    int current = kCurrent;
    int next = kNext;
    const UINT frame_size = srcFrame.Stride * m_height;
    copy(srcFrame.Scan0, frame_size, *m_originalFrameView.get());
    m_frames[kOriginal]->copy_to(*m_frames[current].get());
    for (UINT i = 0; i < phases; ++i)
    {
        ApplyColorSimplifier(m_accelerator, *m_frames[current].get(), *m_frames[next].get(),
            simplifierNeighborWindow);
        std::swap(current, next);
    }
```

```
    ApplyEdgeDetection(m_accelerator,
        *m_frames[current].get(), *m_frames[next].get(), *m_frames[kOriginal].get(),
        simplifierNeighborWindow);
    std::swap(current, next);

    copy(*m_frames[current].get(), destFrame.Scan0, frame_size);
}
```

The real benefit can be seen in the image processing functions themselves. For example, the edge detection code no longer contains calls to the explicit packing and unpacking functions as it did in the array-based implementation. The GPU hardware does the packing and unpacking.

```
void DetectEdge(index<2> idx, const texture<uint_4, 2>& srcFrame,
    const writeonly_texture_view<uint_4, 2>& destFrame, const texture<uint_4, 2>& orgFrame,
    UINT simplifierNeighborWindow, const float_3& W) restrict(amp)
{
   // ...

    const uint_4 srcClr = srcFrame[idc];
    uint_4 destClr;
    const float oneMinusi = 1 - i;
    destClr.r = static_cast<uint>(srcClr.r * oneMinusi);
    destClr.g = static_cast<uint>(srcClr.g * oneMinusi);
    destClr.b = static_cast<uint>(srcClr.b * oneMinusi);
    destClr.a = 0xFF;
    destFrame.set(idc, destClr);
}
```

Writing this using *texture<uint_4, 2>* instead of *array_view<ArgbPackedPixel , 2>* to store the original bitmap data results in much simpler code because there is no need for any unpacking or packing at all! The *ProcessImage()* method (defined in FrameProcessorAmpTextureSingle.cpp) can use the *texture* data directly and make use of the GPU's texture hardware to extract the individual RGB components as *uint* values holding an 8-bit color [0, 255] value.

The cartoonizing image-processing algorithms use a stencil to access the surrounding pixels to calculate new values, so they are a good candidate for a *texture*-based implementation. GPU texture memory is designed to be most efficient when accessing 2D data. The color simplifier and edge detection algorithms require threads to access pixel values on different rows within the image. This means that the memory accesses are not coalesced, so using texture memory might provide some modest performance improvements.

Storing the data as *uint_4* with 8 bits per scalar element means that the kernel also reads and writes less data per pixel because each pixel is stored as 4 bytes, rather than the 12 bytes required for each pixel stored as an *RgbPixel*. This is another area where the application might see gains in performance.

New implementations of *ApplyColorSimplifier()* and *ApplyEdgeDetection()* use *texture<uint_4, 2>* data directly. These separate implementations can be found in FrameProcessorAmpTextureSingle.cpp. They differ in only minor ways from the original versions defined in FrameProcessorAmp.cpp.

Image Processor Settings	Minimum		Default		Max	
Frame processor	**Cartoonize**	**Time/image**	**Cartoonize**	**Time/image**	**Cartoonize**	**Time/image**
C++ AMP Simple Model: single GPU	4.9 ms	30.1 ms	48.7 ms	49.0 ms	221.7 ms	222.1 ms
C++ AMP Tiled Model: single GPU	4.2 ms	30.3 ms	47.1 ms	47.4 ms	177.5 ms	177.9 ms
C++ AMP Textures: single GPU	6.3 ms	28.6 ms	39.9 ms	40.2 ms	178.7 ms	179.2 ms

The texture-based frame processor has roughly equivalent performance to the tiled processor, but the code is simpler and easier to read and maintain. This will vary depending on your application and the algorithms it uses. However, if you think that your algorithm can be implemented with textures and will be able to take advantage of some of their unique features or is hard to express efficiently using *arrays*, then you might want to consider investigating a *texture*-based implementation.

From the numbers above, it is clear that in some cases the cartoonizing stage is no longer the limiting factor on the pipeline. The time per image is actually limited by the display phase of the pipeline. The "DirectX Interop" section later in this chapter shows how to further optimize rendering of result data. For further discussion of the overall performance of all the frame processors implemented by the Cartoonizer case study, see the "Cartoonizer Performance" section of Chapter 10.

HLSL Intrinsic Functions

The High Level Shader Language (HLSL) supports many intrinsic functions. C++ AMP exposes a subset of these within the *concurrency::direct3d* namespace. They can be called only from amp-restricted code.

Function	Description
int abs(int _X)	Returns the absolute value of _X.
float clamp(float _X, float _Min, float _Max)	Clamps the specified value to the specified minimum and maximum range.
int clamp(int _X, int _Min, int _Max)	Clamps the specified value to the specified minimum and maximum range.
unsigned int countbits(unsigned int _X)	Counts the number of bits in the input integer.
int firstbithigh(int _X)	Gets the location of the first set bit starting from the highest order bit and working downward.
int firstbitlow(int _X)	Returns the location of the first set bit starting from the lowest order bit and working upward.
int imax(int _X, int _Y)	Selects the greater of _X and _Y.
int imin(int _X, int _Y)	Selects the lesser of _X and _Y.
float mad(float _X, float _Y, float _Z)	Performs an arithmetic multiply/add operation on three values. Returns (_X * _Y) + _Z.

Function	Description
double mad(double _X, double _Y, double _Z)	Performs an arithmetic multiply/add operation on three values. Returns (_X * _Y) + _Z.
int mad(int _X, int _Y, int _Z)	Performs an arithmetic multiply/add operation on three values. Returns (_X * _Y) + _Z.
unsigned int mad(unsigned int _X, unsigned int _Y, unsigned int _Z)	Performs an arithmetic multiply/add operation on three values. Returns (_X * _Y) + _Z.
float noise(float _X)	Generates a random value within a range between -1.0 and 1.0 using the Perlin noise algorithm.
float radians(float _X)	Converts the specified value, _X, from degrees to radians.
float rcp(float _X)	Calculates a fast, approximate reciprocal.
unsigned int reversebits(unsigned int _X)	Reverses the order of the bits.
float saturate(float _X)	Clamps the specified value within the range of 0.0 to 1.0.
int sign(int _X)	Returns -1 if _X is less than zero, 0 if _X equals zero, and 1 if _X is greater than zero.
float smoothstep(float _Min, float _Max, float _X)	Returns a smooth Hermite interpolation between 0.0 and 1.0 if _X is in the range [_Min, _Max].
float step(float _Y, float _X)	Compares two values, returning 1.0 if the _X parameter is greater than or equal to the _Y parameter; otherwise, 0.0.

Further details on these functions can be found on the "Intrinsic Functions" page, which is part of the HLSL reference on MSDN: *http://msdn.microsoft.com/en-us/library/windows/desktop/ff471376(v=vs.85).aspx*. You can also look at the code comments in the amp.h header. The Cartoonizer uses the *clamp()* and *smoothstep()* functions in FrameProcessorAmp.cpp and FrameProcessorAmpTextureSingle.cpp.

DirectX Interop

If your application performs calculations on a GPU and also renders the results using the GPU, it makes sense to move the result data directly into the DirectX graphics pipeline. The alternative is to copy the data back to the CPU and then load it into a DirectX buffer so that DirectX can copy it back to the GPU and render it. This results in two additional and unnecessary copy operations on your data.

C++ AMP includes some additional functions to allow your program to associate a C++ AMP *accelerator* with a Direct3D device: a C++ AMP *array* with a Direct3D buffer and a C++ AMP *texture* with a Direct3D texture resource. This allows your program to access the same underlying resource using the API that's most suitable for the current problem domain. For example, your computation can use a C++ AMP *array* while your Direct3D rendering code can use a Direct3D buffer. These functions allow you to write computation-focused C++ AMP code and render-focused Direct3D code without having to incur the overhead of additional copies to and from the CPU. It also considerably improves the readability of your program.

Accelerator View and Direct3D Device Interop

C++ AMP's *concurrency::direct3d* namespace contains functions for creating an *accelerator_view* associated with an existing Direct3D device and getting the device associated with an existing *accelerator_view*.

```
accelerator_view create_accelerator_view(IUnknown* device);
```

The Direct3D device must implement the *ID3D11Device* interface with feature level *D3D_FEATURE_LEVEL_11_0* or above. C++ AMP accelerator_views can be accessed from multiple threads, so the device should not be created with the *D3D11_CREATE_DEVICE_FLAG D3D11_CREATE_DEVICE_SINGLETHREADED* set. How to create an appropriate device is shown in the following example:

```
    HRESULT hr = S_OK;
    UINT createDeviceFlags = 0;
#ifdef _DEBUG
    createDeviceFlags |= D3D11_CREATE_DEVICE_DEBUG;
#endif
    std::array<D3D_FEATURE_LEVEL, 1> featureLevels = { D3D_FEATURE_LEVEL_11_0 };

    CComPtr<ID3D11Device> device;
    D3D_FEATURE_LEVEL featureLevel;
    CComPtr<ID3D11DeviceContext> immediateContext;

    hr = D3D11CreateDevice(nullptr /* default adapter */,
        D3D_DRIVER_TYPE_HARDWARE,
        nullptr /* No software rasterizer */,
        createDeviceFlags,
        featureLevels.data(),
        UINT(featureLevels.size()),
        D3D11_SDK_VERSION,
        &device,
        &featureLevel,
        &immediateContext);
    assert(SUCCEEDED(hr));

    accelerator_view dxView = create_accelerator_view(device);
    std::wcout << "Created accelerator_view on "
        << dxView.accelerator.description << std::endl;
```

The *dxView* now represents the *ID3D11Device*, *device*, and its Direct3D immediate context. Note that we use *CComPtr<>* as a smart pointer wrapper around a COM interface. It is part of the ATL and is declared in atlcomcli.h. For a full explanation of how to create a Direct3D device, see the MSDN topic "D3D11CreateDevice function" at *http://msdn.microsoft.com/en-us/library/windows/desktop/ff476082(v=vs.85).aspx*.

It is also possible to obtain the Direct3D device associated with an *accelerator_view*. The following code shows how to get an *IUnknown* interface and query it for the *ID3D11Device*:

```
    HRESULT hr = S_OK;
    CComPtr<ID3D11Device> device;
    IUnknown* unkDev = get_device(accelerator(accelerator::default_accelerator).default_view);
    hr = unkDev->QueryInterface(__uuidof(ID3D11Device), reinterpret_cast<LPVOID*>(&device));
```

Array and Direct3D Buffer Interop

Similarly, your program can create an *array* associated with an existing Direct3D buffer using the *make_array()* function.

```
template<typename T, int N>
array<T,N> make_array(const extent& ext, IUnknown* buffer);
```

The Direct3D buffer must implement the *ID3D11Buffer* interface. It must support raw views (*D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS)* and allow *SHADER_RESOURCE* and *UNORDERED_ACCESS* binding. The buffer itself must be of the correct size, the size of the extent multiplied by the size of the buffer type. The following code uses *make_array* to create an array using the *accelerator_view*, *dxView*, which was created in the previous section:

```
HRESULT hr = S_OK;
UINT bufferSize = 1024;
D3D11_BUFFER_DESC bufferDesc =
{
    bufferSize * sizeof(float),
    D3D11_USAGE_DEFAULT,
    D3D11_BIND_VERTEX_BUFFER | D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_UNORDERED_ACCESS,
    0 /* no CPU access */,
    D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS /* misc flags */,
    sizeof(float)
};
D3D11_SUBRESOURCE_DATA resourceData;
ZeroMemory(&resourceData, sizeof(D3D11_SUBRESOURCE_DATA));

std::vector<float> vertices(bufferSize, 1.0f);

resourceData.pSysMem = &vertices[0];
CComPtr<ID3D11Buffer> buffer;
hr = device->CreateBuffer(&bufferDesc, &resourceData, &buffer);
assert(SUCCEEDED(hr));

array<float, 1> arr = make_array<float, 1>(extent<1>(bufferSize), dxView, buffer);
std::wcout << "Created array<float,1> on "
    << arr.accelerator_view.accelerator.description << std::endl;
```

The *get_buffer()* function supports the reverse operation, getting a Direct3D buffer interface from an *array*. As when getting a device, the returned *IUnknown* must be queried for the desired interface.

```
HRESULT hr = S_OK;
array<int, 1> arr(1024);
CComPtr<ID3D11Buffer> buffer;
IUnknown* unkBuf = get_buffer(arr);
hr = unkBuf->QueryInterface(__uuidof(ID3D11Buffer), reinterpret_cast<LPVOID*>(&buffer));
```

Texture and Direct3D Texture Resource Interop

Your program can also create a *texture* associated with an existing Direct3D texture resource using the *make_texture()* function.

```
template<typename T, int N>
texture<T, N> make_texture(const Concurrency::accelerator_view& view, IUnknown* res)
```

Unlike the previous two interop examples, the type of the *IUnknown* instance returned by *make_texture()* depends on the rank of the *texture*. The following table shows the mapping between the C++ AMP and Direct3D types:

C++ AMP texture	Direct3D interface
texture<T, 1>	ID3D11Texture1D
texture<T, 2>	ID3D11Texture2D
texture<T, 2>	ID3D11Texture3D

Similarly, there is a mapping between the scalar or short vector type stored in a *texture* created in C++ AMP and the Direct3D texture resource's corresponding pixel format. The following table shows only the valid bits per scalar element values for each value type, as per the discussion of valid texel types in the texture<T, N> section:

Value Type	Bits per scalar element	Direct3D DXGI_FORMAT
int	8	DXGI_FORMAT_R8_SINT
	16	DXGI_FORMAT_R16_SINT
	32	DXGI_FORMAT_R32_SINT
int_2	8	DXGI_FORMAT_R8G8_SINT
	16	DXGI_FORMAT_R16G16_SINT
	32	DXGI_FORMAT_R32G32_SINT
int_4	8	DXGI_FORMAT_R8G8B8A8_SINT
	16	DXGI_FORMAT_R16G16B16A16_SINT
	32	DXGI_FORMAT_R32G32B32A32_SINT
uint	8	DXGI_FORMAT_R8_UINT
	16	DXGI_FORMAT_R16_UINT
	32	DXGI_FORMAT_R32_UINT
uint_2	8	DXGI_FORMAT_R8G8_UINT
	16	DXGI_FORMAT_R16G16_UINT
	32	DXGI_FORMAT_R32G32_UINT
uint_4	8	DXGI_FORMAT_R8G8B8A8_UINT
	16	DXGI_FORMAT_R16G16B16A16_UINT
	32	DXGI_FORMAT_R32G32B32A32_UINT
float_2	16	DXGI_FORMAT_R16_FLOAT
	32	DXGI_FORMAT_R32_FLOAT
float_2	16	DXGI_FORMAT_R16G16_FLOAT

Value Type	Bits per scalar element	Direct3D DXGI_FORMAT
	32	DXGI_FORMAT_R32G32_FLOAT
float_4	16	DXGI_FORMAT_R16G16B16A16_FLOAT
	32	DXGI_FORMAT_R32G32B32A32_FLOAT
double	64	DXGI_FORMAT_R32G32_UINT
double_2	64	DXGI_FORMAT_R32G32B32A32_UINT
unorm	8	DXGI_FORMAT_R8_UNORM
	16	DXGI_FORMAT_R16_UNORM
unorm_2	8	DXGI_FORMAT_R8G8_UNORM
	16	DXGI_FORMAT_R16G16_UNORM
unorm_4	8	DXGI_FORMAT_R8G8B8A8_UNORM
	16	DXGI_FORMAT_R16G16B16A16_UNORM
norm	8	DXGI_FORMAT_R8_SNORM
	16	DXGI_FORMAT_R16_SNORM
norm_2	8	DXGI_FORMAT_R8G8_SNORM
	16	DXGI_FORMAT_R16G16_SNORM
norm_4	8	DXGI_FORMAT_R8G8B8A8_SNORM
	16	DXGI_FORMAT_R16G16B16A16_SNORM

These mappings do not apply if the C++ AMP *texture* is created through interop from the Direct3D texture resource. In this case, C++ AMP does not attempt to set the *bits_per_scalar_element* property or check that the Direct3D texture format can be mapped onto the *texture* value type. The C++ AMP run time surfaces any errors logged by the DirectX run time by means of exceptions. In debug mode, the DirectX debug layer is on and extensive error checks are performed, making it easier to spot incorrect mappings. In release mode, some of the errors will not be detected and will lead to undefined behavior.

The following example shows how to use this code to create an *ID3D11Texture2D* buffer containing *DXGI_FORMAT_R8G8B8A8_UINT* pixels and make a C++ AMP *texture<uint4, 2>* that uses the same resource:

```
const int height = 100;
const int width = 100;

D3D11_TEXTURE2D_DESC desc;
ZeroMemory(&desc, sizeof(desc));
desc.Height = height;
desc.Width = width;
desc.MipLevels = 1;
desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_R8G8B8A8_UINT;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
```

```
desc.Usage = D3D11_USAGE_DEFAULT;
desc.BindFlags = D3D11_BIND_UNORDERED_ACCESS | D3D11_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = 0;
desc.MiscFlags = 0;

CComPtr<ID3D11Texture2D> dxTexture = nullptr;
hr = device->CreateTexture2D(&desc, nullptr, &dxTexture);
assert(SUCCEEDED(hr));

texture<uint4, 2> ampTexture = make_texture<uint4, 2>(dxView, dxTexture);
```

The resulting texture will have an extent of [100, 100], the same height and width of the Direct3D texture resource. Here, the number of MIP map levels is set to one. If the texture resource has more than one MIP level, C++ AMP can access only the first MIP map level. Here, the texture has the *D3D11_BIND_SHADER_RESOURCE* bind flag set to allow reading and the *D3D11_BIND_UNORDERED_ACCESS* flag set to allow writing.

You can find more information on the texture description structure on MSDN in the "D3D11_TEXTURE2D_DESC structure" topic: *http://msdn.microsoft.com/en-us/library/windows/desktop/ff476253(v=vs.85).aspx*.

The *get_texture()* function supports the reverse operation, getting a Direct3D resource from a *texture*. As in the above example, the texture and Direct3D resource ranks must be equal, and the Direct3D format and the C++ AMP *texture* value type must be compatible.

```
texture<int, 2> text(100, 100);
CComPtr<ID3D11Texture2D> texture;
IUnknown* unkRes = get_texture(text);
hr = unkRes>QueryInterface(__uuidof(ID3D11Texture2D), reinterpret_cast<LPVOID*>(&texture));
assert(SUCCEEDED(hr));
```

Using Graphics Interop

The previous section discussed C++ AMP's support for interop in terms of the methods it implements to allow your application to associate *arrays* with Direct3D buffers and Direct3D devices with *accelerator_views*. This section reviews how the NBody case study introduced in Chapter 2, "NBody Case Study," makes use of interop.

The NBody sample uses one of the available GPUs to handle the rendering of the particles but uses all available C++ AMP-capable GPUs for calculations. By convention, the application uses the first available accelerator to handle rendering. The *CreateParticlePosBuffer()* method in NBodyGravityAmp.cpp creates the appropriate *ID3D11Buffer* buffers for rendering and associates them with corresponding *array<float_3, 1>* arrays containing the positions of the particles. These are held in *g_deviceData[0]->DataOld->pos* and *g_deviceData[0]->DataNew->pos* respectively.

```
CComPtr<ID3D11Buffer> g_pParticlePosOld;
CComPtr<ID3D11Buffer> g_pParticlePosNew;
std::vector<std::shared_ptr<TaskData>> g_deviceData;
```

```
// ...

HRESULT CreateParticlePosBuffer(ID3D11Device* pd3dDevice)
{
    HRESULT hr = S_OK;
    accelerator_view renderView =
        concurrency::direct3d::create_accelerator_view(reinterpret_cast<IUnknown*>(pd3dDevice));
    g_deviceData = CreateTasks(g_maxParticles, renderView);
    LoadParticles();

    //  Particles from GPU zero are the ones synced with the graphics buffers.
    //  Attach AMP array of positions to D3D buffer.

    g_pParticlePosOld = nullptr;
    g_pParticlePosNew = nullptr;

    hr = concurrency::direct3d::get_buffer(
            g_deviceData[0]->DataOld->pos)->QueryInterface(__uuidof(ID3D11Buffer),
            reinterpret_cast<LPVOID*>(&g_pParticlePosOld));
    V_RETURN(hr);
    hr =
        concurrency::direct3d::get_buffer(
            g_deviceData[0]->DataNew->pos)->QueryInterface(__uuidof(ID3D11Buffer),
            reinterpret_cast<LPVOID*>(&g_pParticlePosNew));
    V_RETURN(hr)
```

For each buffer, the corresponding resource and unordered access views are created.

```
    hr = pd3dDevice->CreateShaderResourceView(g_pParticlePosOld, &resourceDesc,
        &g_pParticlePosRvOld);
    // ...
    hr = pd3dDevice->CreateUnorderedAccessView(g_pParticlePosOld, &viewDesc,
        &g_pParticlePosUavOld);
```

The *CreateTasks()* method, defined in NBodyAmp.h, adds a *TaskData* instance for each available GPU accelerator and makes sure that the *tasks[0]* is associated with the *renderView*. All other *TaskData* instances are associated with their corresponding accelerator's default view.

```
std::vector<std::shared_ptr<TaskData>> CreateTasks(int numParticles,
    accelerator_view renderView)
{
    std::vector<accelerator> gpuAccelerators = AmpUtils::GetGpuAccelerators();
    std::vector<std::shared_ptr<TaskData>> tasks;
    tasks.reserve(gpuAccelerators.size());

    if (!gpuAccelerators.empty())
    {
        //  Create first accelerator attached to main view. This will attach the C++ AMP
        //  array<float_3> to the D3D buffer on the first GPU.
        tasks.push_back(std::make_shared<TaskData>(numParticles, renderView,
            gpuAccelerators[0]));

        //  All other GPUs are associated with their default view.
        std::for_each(gpuAccelerators.cbegin() + 1, gpuAccelerators.cend(),
            [=, &tasks](const accelerator& d)
```

```
        {
            tasks.push_back(std::make_shared<TaskData>(numParticles, d.default_view, d));
        });
    }
```

At the end of each integration step, the *RenderParticles()* method uses the data in the shader resource view, *g_pParticlePosRvOld*, which points to the data in *g_deviceData[0]->DataOld->pos*.

```
pd3dImmediateContext->IASetVertexBuffers(0, 1, &g_pParticleBuffer.p, &stride, &offset);
// ...
pd3dImmediateContext->VSSetShaderResources(0, 1, &g_pParticlePosRvOld.p);
// ...
pd3dImmediateContext->Draw(g_numParticles, 0);
```

Two separate old and new buffers are required because each integration step reads data from *TaskData::DataOld* and writes it to *TaskData::DataNew*. After each step, the old and new data are swapped, as are the corresponding DirectX buffers. The following code is in the *OnFrameMove()* method, which is responsible for updating the particles prior to rendering.

```
g_pNBody->Integrate(g_deviceData, g_numParticles);

std::for_each(g_deviceData.begin(), g_deviceData.end(), [](std::shared_ptr<TaskData>& t)
{
    std::swap(t->DataOld, t->DataNew);
});
std::swap(g_pParticlePosOld, g_pParticlePosNew);
std::swap(g_pParticlePosRvOld, g_pParticlePosRvNew);
std::swap(g_pParticlePosUavOld, g_pParticlePosUavNew);
```

The NBody example gets significant performance gains from this approach because it avoids ever having to copy data back to the CPU. Data is copied onto the GPU only once after the initial particle data is generated on the CPU. This saves copying the all the particle position data from the GPU and back again for each iteration timestep.

Summary

C++ AMP provides additional features to allow your computations to make the best use of the additional GPU hardware. Short vector types and textures allow your application to make use of the texture-specific hardware present on most GPUs, while the intrinsic functions provide access to some HLSL functionality that might help with your application's performance. The Cartoonizer case study shows how you can use these features to write more efficient code when the algorithms align well with *texture*-based data storage and accesses.

Your applications can also use the Direct3D interop features of C++ AMP to move result data directly from an *array* into a Direct3D buffer and the Direct3D rendering pipeline. This is far more efficient than copying it back to the CPU only to copy it into a Direct3D buffer and back to the GPU. In some cases, your application might never need to copy data back to the CPU. The NBody case study is a good example of this.

CHAPTER 12

Tips, Tricks, and Best Practices

In this chapter:

Dealing with Tile Size Mismatches 283
Initializing Arrays 290
Function Objects vs. Lambdas 291
Atomic Operations 292
Additional C++ AMP Features on Windows 8 295
Time-Out Detection and Recovery 296
Double-Precision Support 299
Debugging on Windows 7 300
Additional Debugging Functions 302
Deployment 303
C++ AMP and Windows 8 Windows Store Apps 306
Using C++ AMP from Managed Code 306
Summary 307

This final chapter covers some best practices as well as a few tips and tricks that will help you get the most out of C++ AMP. It also includes some more advanced topics that were not covered in the previous chapters of the book.

Dealing with Tile Size Mismatches

Although C++ AMP tiles are a fixed size that is determined at compile time, the data your application tries to process usually varies based on user input at run time. The NBody and Cartoonizer case studies both show examples of this, as they allow the user to choose the size of the input data. In some cases, the user is restricted to picking a data set size that is a multiple of the tile size. The NBody case study takes this approach; the number of particles is restricted to multiples of 512, which is an exact multiple of the available tile sizes. In the case of the Cartoonizer application, the tile size is fixed

and users are free to use whatever image sizes they want. The application ensures that images with dimensions that are not exactly divisible by the tile size are dealt with correctly.

The following sections look at several implementations to the two possible approaches to this problem: either padding the compute domain so that its dimensions fit to an exact number of tiles or truncating the compute domain to match an exact number of tiles and then calculating the results for the remaining data elements.

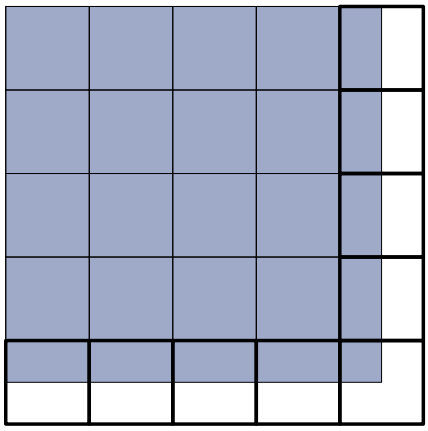
A tiled matrix transpose implementation is used because it's a simple example that benefits from tiling. A matrix transpose simply swaps the contents of a square matrix along the leading diagonal. The tiled matrix transpose is also discussed from a performance perspective in Chapter 7, "Optimization." Here is the code for a tiled matrix transpose. You can run this example by loading the Chapter12\Chapter12.sln solution. Build the sample in Release configuration and run it by pressing Ctrl+F5 to start it without the debugger attached.

```
static const int tileSize = 16;

void TransposeExample(int matrixSize)
{
    if (matrixSize % tileSize != 0)
        throw std::exception("matrix is not a multiple of tile size.");

    std::vector<unsigned int> inData(matrixSize * matrixSize);
    std::vector<unsigned int> outData(matrixSize * matrixSize, 0u);
    std::iota(inData.begin(), inData.end(), 0u);

    array_view<const unsigned int, 2> inDataView(matrixSize, matrixSize, inData);
    array_view<unsigned int, 2> outDataView(matrixSize, matrixSize, outData);
    outDataView.discard_data();

 tiled_extent<tileSize, tileSize> computeDomain =
     inDataView.extent.tile<tileSize, tileSize>();
    parallel_for_each(computeDomain, [=](tiled_index<tileSize, tileSize> tidx) restrict(amp)
    {
        tile_static unsigned int localData[tileSize][tileSize];
        localData[tidx.local[1]][tidx.local[0]] = inDataView[tidx.global];

        tidx.barrier.wait();

        index<2> outIdx(index<2>(tidx.tile_origin[1], tidx.tile_origin[0]) + tidx.local);
        outDataView[outIdx] = localData[tidx.local[0]][tidx.local[1]];
    });
    // ...
```

The *TransposeExample()* function shown here can correctly execute only for *matrixSize* values that are exactly divisible by the *tileSize*. The following sections show how to modify this code to handle matrices of any size. Matrix transpose operations can be applied only to square matrices, but the padding and truncation approaches apply equally well to data with unequal extents. The example uses two-dimensional data, but padding and truncation can be applied to data of any dimension. You can run these examples on your own hardware by running the example code in Chapter12\Chapter12.sln.

Padding Tiles

One approach to improving the matrix transpose example shown in the previous section is to increase the size of the compute domain so that its dimensions match an exact number of tiles. C++ AMP refers to this as padding. The diagram shows how the matrix can be divided up into tiles containing the matrix elements (shaded) and some additional elements (unshaded) used to pad the compute domain to an exact number of tiles. The boxes on the diagram represent tiles, each containing many individual matrix elements.



You can see that the rightmost and bottommost tiles (with heavy borders) contain real matrix elements (shaded) and additional padded elements (unshaded). Adding code to handle these padding elements is the key to implementing a padded solution to this matrix transpose program.

Instead of reading the matrix elements directly, two new methods, *PaddedRead()* and *PaddedWrite()*, ensure that the padded compute domain elements are dealt with correctly. The *PaddedRead()* method returns the default value for the type *T* if the requested *index* falls outside the true *extent* of the matrix. C++ AMP provides the *extent::contains()* method to test if an *index* is within an *extent*.

```
template <typename T, unsigned int Rank>
T PaddedRead(const array_view<const T, Rank>& A, const index<Rank>& idx) restrict(amp)
{
    return A.extent.contains(idx) ? A[idx] : T();
}
```

Similarly, *PaddedWrite()* checks that the *index* falls within the *extent* of the matrix and only writes values to elements within the matrix. These two methods guarantee that the additional padded matrix elements are ignored. You can consider them as virtual elements. They pad the compute domain and no storage is allocated for these additional elements.

```
    template <typename T, unsigned int Rank>
void PaddedWrite(const array_view<T, Rank>& A, const index<Rank>& idx, const T& val)
        restrict(amp)
    {
        if (A.extent.contains(idx))
            A[idx] = val;
    }
```

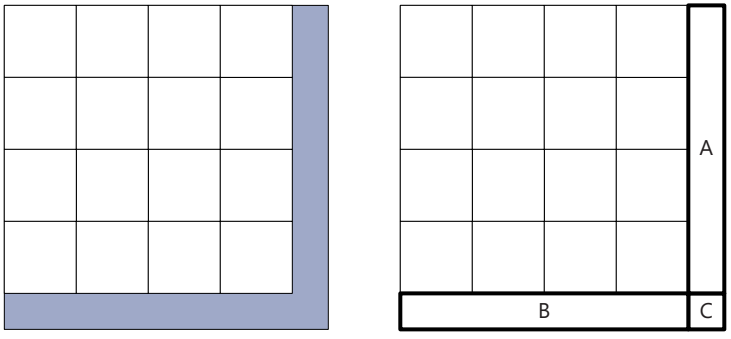
Although the range checking in these new functions adds some additional overhead, it's relatively insignificant compared to the cost of the global memory access. Modifying the original sample code to support padding is now very easy. The original *computeDomain* is padded using the *tile_extent::pad()* method, and the read and write operations to global memory are replaced with calls to *PaddedRead()* and *PaddedWrite()*.

```
tiled_extent<tileSize, tileSize> computeDomain =
        inDataView.extent.tile<tileSize, tileSize>();
    computeDomain = computeDomain.pad();
    parallel_for_each(view, computeDomain,
        [=](tiled_index<tileSize, tileSize> tidx) restrict(amp)
    {
        tile_static unsigned int localData[tileSize][tileSize];
        localData[tidx.local[1]][tidx.local[0]] = PaddedRead(inDataView, tidx.global);

        tidx.barrier.wait();

        index<2> outIdx(index<2>(tidx.tile_origin[1], tidx.tile_origin[0]) + tidx.local);
        PaddedWrite(outDataView, outIdx, localData[tidx.local[0]][tidx.local[1]]);
    });
```

This padding approach results in very simple code, but it requires your application to run more threads than are actually needed to solve the problem. The next section describes an alternative approach.

Truncating Tiles

Rather than padding the compute domain as shown in the previous section, it's also possible to take the opposite approach. Truncate the compute domain into an area that fits an exact number of tiles and then deal with the edges separately. The diagram shows how the tiles (unshaded) cover only some of the matrix elements. The remaining elements (shaded) lie outside of the tiled elements, but they must still be transposed if the program is going to calculate the correct result. This section describes two approaches to handling these additional elements.

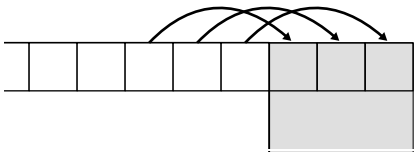


The matrix elements now fall into four areas:

- The elements that lie within the tiles
- The elements to the right of the rightmost tile (area A)
- The elements below the bottommost tile (area B)
- The remaining elements below and left of the bottom-rightmost tile (area C)

Handling Truncated Elements with Edge Threads

This first approach uses all threads within the rightmost and bottommost margins to calculate the values for the truncated region. The following diagram shows this approach. It shows the top-right corner of the previous diagram. The arrows show how threads close to the rightmost edge each handle their own element and a truncated element.



The code shows how this is implemented. If a thread lies within the *rightMargin* or *bottomMargin,* it transposes not only the element within the tile but also one additional element for the truncated area of the row or column. Threads lying within both *rightMargin* and *bottomMargin* perform a further calculation to update an additional element in Area C.

```
const int rightMargin = inDataView.extent[1] - computeDomain[1];
const int bottomMargin = inDataView.extent[0] - computeDomain[0];
parallel_for_each(view, computeDomain,
    [=](tiled_index<tileSize, tileSize> tidx) restrict(amp)
{
    tile_static unsigned int localData[tileSize][tileSize];
    localData[tidx.local[1]][tidx.local[0]] = inDataView[tidx.global];
    tidx.barrier.wait();
    index<2> outIdx(index<2>(tidx.tile_origin[1], tidx.tile_origin[0]) + tidx.local);
    outDataView[outIdx] = localData[tidx.local[0]][tidx.local[1]];

    bool isRightMost = tidx.global[1] >= computeDomain[1] - rightMargin;
    bool isBottomMost = tidx.global[0] >= computeDomain[0] - bottomMargin;
    if (isRightMost | isBottomMost)
    {
        int idx0, idx1;
        if (isRightMost)                          // Area A.
        {
            idx0 = tidx.global[0];
            idx1 = tidx.global[1] + rightMargin;
            outDataView(idx1, idx0) = inDataView(idx0, idx1);
        }
        if (isBottomMost)                         // Area B.
        {
            idx1 = tidx.global[1];
```

```
            idx0 = tidx.global[0] + bottomMargin;
            outDataView(idx1, idx0) = inDataView(idx0, idx1);
        }
        if (isRightMost & isBottomMost)          // Area C.
        {
            idx0 = tidx.global[0] + bottomMargin;
            idx1 = tidx.global[1] + rightMargin;
            outDataView(idx1, idx0) = inDataView(idx0, idx1);
        }
    }
});
```

This approach results in slightly more code than the padding implementation, but there are no idle threads. Each thread transposes a single element, and the threads within the margin transpose at most three additional elements to ensure that all the truncated elements are also correctly transposed.

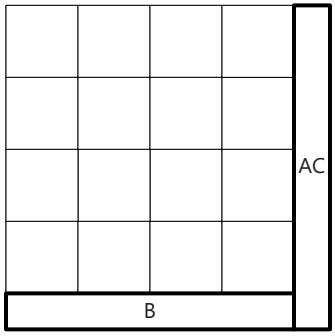
Handling Truncated Elements with Sections

In the previous section on truncation, you saw how the matrix could be considered as four separate areas: one made up of an exact number of tiles and three others (areas A, B, and C) that could not be tiled. It's also possible to consider these as two separate problems—tiled and simple matrix transpose operations—and write two functions to compute them.

```
void SimpleTranspose(const array_view<const unsigned int, 2>& inDataView,
    const array_view<unsigned int, 2>& outDataView)
{
    outDataView.discard_data();
    parallel_for_each(outDataView.extent, [=] (index<2> idx) restrict(amp)
    {
        outDataView(idx[0], idx[1]) = inDataView(idx[1], idx[0]);
    });
}

template <int TileSize>
void TiledTranspose(const array_view<const unsigned int, 2>& inDataView,
    const array_view<unsigned int, 2>& outDataView)
{
    outDataView.discard_data();
    parallel_for_each(outDataView.extent.tile<TileSize, TileSize>(),
        [=] (tiled_index<TileSize, TileSize> tidx) restrict(amp)
    {
        tile_static unsigned int localData[tileSize][tileSize];
        localData[tidx.local[1]][tidx.local[0]] = inDataView[tidx.global];
        tidx.barrier.wait();
        index<2> outIdx(index<2>(tidx.tile_origin[1], tidx.tile_origin[0]) + tidx.local);
        outDataView[outIdx] = localData[tidx.local[0]][tidx.local[1]];
    });
}
```

The *SimpleTranspose()* and *TiledTranspose()* methods take *array_view* parameters. This means that they can take subsections of the original *array* as inputs by using an *array_view::section* to partition the original matrix. Now it's possible to rewrite the truncated algorithm as three separate kernels. One

handles the main area using *TiledTranspose(),* and the two others handle the truncated areas B and AC with *SimpleTranspose()*.



The following code shows how this is implemented:

```
tiled_extent<tileSize, tileSize> computeDomain
    = inDataView.extent.tile<tileSize, tileSize>();
tiled_extent<tileSize, tileSize> truncatedDomain = computeDomain.truncate();
bool isBottomTruncated = truncatedDomain[0] < computeDomain[0];
bool isRightTruncated = truncatedDomain[1] < computeDomain[1];
array_view<const unsigned int, 2> fromData  =
    inDataView.section(index<2>(0, 0), truncatedDomain);
array_view<unsigned int, 2> toData =
    outDataView.section(index<2>(0, 0), extent<2>(truncatedDomain[1], truncatedDomain[0]));
TiledTranspose<tileSize>(fromData, toData);

if (isBottomTruncated)                  // Area B.
{
    index<2> offset(truncatedDomain[0], 0);
    extent<2> ext(inDataView.extent[0] - truncatedDomain[0], truncatedDomain[1]);
    fromData  = inDataView.section(offset, ext);
    toData = outDataView.section(index<2>(offset[1], offset[0]), extent<2>(ext[1], ext[0]));
    SimpleTranspose(fromData, toData);
 outDataView.synchronize();
}
if (isRightTruncated)                  // Area AC.
{
    index<2> offset(0, truncatedDomain[1]);
    fromData  = inDataView.section(offset);
    toData = outDataView.section(index<2>(offset[1], offset[0]));
    SimpleTranspose(fromData, toData);
}
```

Although this implementation is more complex than the previous ones, it does highlight the degree to which *array_view* and *array_view::section* can be used to compose computations across several kernel invocations.

Comparing Approaches

Of the three different implementations described, which one is most appropriate? The transpose example doesn't show a clear winner in terms of performance of the padding approach versus truncating and using a margin of threads or sections to calculate the truncated values. The following are some general guidelines on which approach to choose:

- Padding usually results in simpler code that is easier to write and maintain at the expense of using threads that do not contribute to the computation.
- Truncating requires more complex code but uses the available threads more efficiently.
- Using sections and multiple kernels further adds to the complexity but might lead to more code reuse if the individual kernels are useful in other parts of your codebase.

However, a matrix transpose is a relatively simple example. Your application might see performance improvements from using one or another approach. Given the additional work required to implement truncating, it's reasonable to start off with a padding-based solution and consider using truncation if benchmarking and performance investigations suggest that your application might benefit from a more complex but potentially faster implementation.

Initializing Arrays

The STL *vector<T>* class supports constructor overloads to set all the values in the vector on initialization. The following code initializes all elements in *theData* to 1.5:

```
std::vector<float> theData(10000, 1.5f);
```

The C++ AMP array class does not support a similar constructor overload to initialize the elements. However, it's quite straightforward to write a *Fill()* function that implements this.

```
template<typename T, int Rank>
void Fill(array<T, Rank>& arr, T value)
{
    parallel_for_each(arr.extent, [&arr, value](index<Rank> idx) restrict(amp)
    {
        arr[idx] = value;
    });
}
```

Using the *Fill()* function enables you to set the elements in an array.

```
array<float, 2> theData(100, 100);
Fill(theData, 1.5f);
```

C++ templates make it easy to extend C++ AMP. The *Fill()* utility function is just one example of this.

Function Objects vs. Lambdas

Most of the code in this book uses lambdas rather than function objects, or functors. Lambdas are now part of the C++11 standard. They result in more readable code because the code being executed appears where it executes, rather than being defined in a class declared somewhere else in the code.

The following example from Chapter 4, "Tiling," shows a simple implementation of matrix multiply:

```
void MatrixMultiply(std::vector<float>& vC,
    const std::vector<float>& vA,
    const std::vector<float>& vB, int M, int N, int W)
{
    array_view<const float,2> a(M, W, vA);
    array_view<const float,2> b(W, N, vB);
    array_view<float,2> c(M, N, vC);
    c.discard_data();
    parallel_for_each(c.extent, [=](index<2> idx) restrict(amp)
    {
        int row = idx[0];
        int col = idx[1];
        float sum = 0.0f;
        for(int i = 0; i < W; i++)
            sum += a(row, i) * b(i, col);
        c[idx] = sum;
    });
    c.synchronize();
}
```

It is possible to rewrite this code using a functor. Instead of capturing the variables in the lambda, these are passed in to the functor's constructor and stored as member variables. The member variables must keep the pass-by-value and pass-by-reference semantics intact. In this case *mA*, *mB*, *mC,* and *W* are implicitly captured by value.

The class's call operator should take the same parameters and return the same result. In this example, the lambda takes an *index<2>* and returns *void*. It should also have the same restrict specifiers as the lambda. Here, this is *restrict(amp)*. The functor must also obey the additional rules associated with C++ AMP code. For example, it can't use unsupported types.

```
class Multiply
{
private:
    array_view<const float, 2> m_mA;
    array_view<const float, 2> m_mB;
    array_view<float, 2> m_mC;
    int m_W;

public:
    Multiply(const array_view<const float, 2>& a,
             const array_view<const float, 2>& b,
             const array_view<float, 2>& c,
             int w) : m_mA(a), m_mB(b), m_mC(c), m_W(w)
    {}
```

```
    void operator()(index<2> idx) const restrict(amp)
    {
        int row = idx[0]; int col = idx[1];
        float sum = 0.0f;
        for(int i = 0; i < m_W; i++)
            sum += m_mA(row, i) * m_mB(i, col);
        m_mC[idx] = sum;
    }
};
```

The *parallel_for_each* now becomes a single line of code.

```
parallel_for_each(defaultView, extent<2>(eC), Multiply(mA, mB, mC, W));
```

The functor is passed into the *parallel_for_each* by blitting. This means that the copy constructor is not invoked. Any further copies are also blitted so that copy and move constructors and destructors will not be called.

There are some cases where functors might be more appropriate. A functor can make use of templates and inheritance. Functors can also be reused in different *parallel_for_each* calls. In most cases, it's a matter of personal preference as to the choice of functors or lambdas.

For a general discussion of lambdas and functors, see the "Lambda Expressions in C++" topic on MSDN: *http://msdn.microsoft.com/en-us/library/dd293608(v=vs.110).aspx*.

Atomic Operations

Atomic operations are operations, or a set of operations, that are guaranteed to be isolated from other concurrent processes and to appear to other concurrent processes to have occurred instantaneously. Atomic operations either successfully change the system state or have no visible effects. For example, an atomic increment operation to increment **dest* by *val* loads the current value at the memory location pointed to by *dest*, adds *val* to it, and stores the result in **dest*. This set of operations appears to other concurrent processes as a single operation. See the Wikipedia article "Linearizability" for an introduction to atomic operations: *http://en.wikipedia.org/wiki/Linearizability*.

C++ AMP threads within the same tile can share data using *tile_static* memory and the *tiled_index* barrier methods, such as *tiled_index::wait()*, to synchronize execution of threads in a tile and their memory accesses. To share data safely between threads in a nontiled *parallel_for_each* or between threads in a tiled *parallel_for_each* that are in different tiles, your program must use global memory and atomic operations.

C++ AMP defines the following atomic operations; they are defined in amp.h.

```
int atomic_fetch_add(int* dest, int val) restrict(amp)
unsigned int atomic_fetch_add(unsigned int* dest, unsigned int val) restrict(amp)
```

Adds *val* to **dest* and returns the original value of **dest*.

```
int atomic_fetch_sub(int* dest, int val) restrict(amp)
unsigned int atomic_fetch_sub(unsigned int* dest, unsigned int val) restrict(amp)
```

Subtracts *val* from **dest* and returns the original value of **dest*.

```
int atomic_fetch_inc(int* dest) restrict(amp)
unsigned int atomic_fetch_inc(unsigned int* dest) restrict(amp)
```

Increments **dest* and returns the original value of **dest*.

```
int atomic_fetch_dec(int* dest) restrict(amp)
unsigned int atomic_fetch_dec(unsigned int* dest) restrict(amp)
```

Decrements **dest* and returns the original value of **dest*.

```
int atomic_exchange(int* dest, int val) restrict(amp)
unsigned int atomic_exchange(unsigned int* dest, unsigned int val) restrict(amp)
float atomic_exchange(float * dest, float val) restrict(amp)
```

Sets the value of **dest* to *val* as an atomic operation and returns the original value of **dest*.

```
bool atomic_compare_exchange(int* dest, int* expected, int val) restrict(amp)
bool atomic_compare_exchange(unsigned int* dest, unsigned int* expected, unsigned int val)
    restrict(amp)
```

Compares **dest* and **expected* for equality. The result of this comparison dictates the functions behavior:

true - The function returns *true* and sets **dest* to *val*; **expected* is unchanged.

false - The function returns *false* and sets **expected* to *dest; *dest is unchanged.

```
int atomic_fetch_max(int* dest, int val) restrict(amp)
unsigned int atomic_fetch_max(unsigned int* dest, unsigned int val) restrict(amp)
```

Sets the value of **dest* to the maximum of *val* and **dest* and returns the original value of **dest*.

```
int atomic_fetch_min(int* dest, int val) restrict(amp)
unsigned int atomic_fetch_min(unsigned int* dest, unsigned int val) restrict(amp)
```

Sets the value of **dest* to the minimum of *val* and **dest* and returns the original value of **dest*.

```
int atomic_fetch_and(int* dest, int val) restrict(amp)
unsigned int atomic_fetch_and(unsigned int* dest, unsigned int val) restrict(amp)
```

Performs a bitwise-and operation of *val* to **dest* and returns the original value of **dest*.

```
int atomic_fetch_or(int* dest, int val) restrict(amp)
unsigned int atomic_fetch_or(unsigned int* dest, unsigned int val) restrict(amp)
```

Performs a bitwise-XOR operation of *val* to **dest* and returns the original value of **dest*.

```
int atomic_fetch_xor(int* dest, int val) restrict(amp)
unsigned int atomic_fetch_xor(unsigned int* dest, unsigned int val) restrict(amp)
```

Performs a bitwise-XOR operation of *val* to **dest* and returns the original value of **dest*.

These atomic operations in C++ AMP also have the following limitations:

- You should not mix atomic and normal (nonatomic) reads and writes. Normal reads might not see the results of atomic writes to the same memory location. Normal writes should not be mixed with atomic writes to the same memory location. If your program does not comply with these criteria, this will lead to an undefined result.
- Atomic operations do not imply a memory fence of any sort. Atomic operations may be reordered. This differs from the behavior of interlocked operations in C++.

In the following example, which is somewhat contrived, the program updates all the elements in an array of random floating-point numbers in the range [0.0, 1.0]. It also tracks the number of array elements that are greater than or equal to 0.999. Given the uniform distribution of random numbers, an array of 100,000 *float* values would usually contain about 10 elements that met these criteria.

```
std::random_device rd;
std::default_random_engine engine(rd());
std::uniform_real_distribution<float> randDist(0.0f, 1.0f);
std::vector<float> theData(100000);
std::generate(theData.begin(), theData.end(), [=, &engine, &randDist]()
    { return randDist(engine); });
array_view<float, 1> theDataView(int(theData.size()), theData);

int exceptionalOccurrences = 0;
array_view<int> count(1, &exceptionalOccurrences);
parallel_for_each(theDataView.extent, [=] (index<1> idx) restrict(amp)
{
    if (theDataView[idx] >= 0.999f)  // Exceptional occurrence.
    {
        atomic_fetch_inc(&count(0));
    }
    theDataView[idx] = // Update the value...
});
count.synchronize();
std::wcout << "Calculating values for " << theData.size() << " elements " << std::endl;
std::wcout << "A total of " << exceptionalOccurrences
    << " exceptional occurrences were detected."
    << std::endl << std::endl;
```

The most important point about this sample is that the atomic operation is being used to count the number of highly uncommon occurrences across all the threads. The output of this program should look something like the following:

```
Calculating values for 100000 elements
A total of 5 exceptional occurrences were detected.
```

While one thread is executing an atomic operation, other threads are blocked until it completes. This can have a significant impact on your application's performance. For example, you could write a reduction kernel, like those described in Chapter 8, "Performance Case Study—Reduction," by using *atomic_fetch_add*. However, this would perform very poorly because each thread would have to wait its turn to update the result, effectively serializing the work across all threads. Use atomic operations with care because they can have a significant impact on the performance of your application.

Additional C++ AMP Features on Windows 8

Although C++ AMP runs on Windows 7, Windows Server 2008 R2, Windows 8, and Windows Server 2012, there are some new features available only on Windows 8 and Windows Server 2012.

- **Debugging support.** Windows 8 supports debugging using the REF accelerator. Unless your GPU hardware driver supports debugging, you must either install Windows 8 on your development machine or run it in a VM or on another machine and remotely debug your application. See the "Debugging on Windows 7" and "Additional Debugging Functions" sections for further details.
- **WARP accelerator support.** Only Windows 8 supports the WARP accelerator, or "Microsoft Basic Render Driver." Your application can't fall back to the WARP accelerator when running on Windows 7 or Windows Server 2008 R2. If you require your application to run on Windows 7 or Windows Server 2008 R2, you should provide a fallback implementation that runs on the CPU.
- **Session 0 support.** Running C++ AMP applications under Session 0 is supported on Windows 8 and Windows Server 2012 but not on Windows 7 or Windows Server 2008 R2. See the section of this chapter entitled "Running as a Service or Under Session 0" for further details.
- **True headless server support.** On Windows 7 and Windows Server 2008 R2, a GPU must offer at least one device that offers both a DirectX 11 driver and advertises itself to Windows as a display device, even if no physical display is connected. See the section of this chapter entitled "Running on True Headless Servers" for further details.
- **XPDM Devices.** On Windows 7 and Windows Server 2008 R2, the presence of graphics devices that support only the XP Driver Model (XPDM) might result in C++ AMP being unable to recognize DirectX 11 devices. Windows 8 does not have this limitation. See the section of this chapter entitled "Running with XPDM Graphics Devices Present" for further details.
- **Full double-precision support.** Windows 8 supports WDDM 1.2, which means that full double precision is available as an optional feature if your GPU's hardware driver supports it. See the section "Double-Precision Support" for further details.
- **Increased number of writable resources.** C++ AMP supports a limited number of writable resources. On Windows 7 with DirectX 11, DirectCompute supports 128 read-only buffers/

textures but only eight writable buffers/textures. On Windows 8, which supports DirectX 11.1, 64 writable buffers/textures are supported.

- **Disabling TDR time-outs.** On Windows 8, it's possible to create an *accelerator_view* that is not subject to TDR time-outs. This is not possible on Windows 7. See the section of this chapter entitled "Time-Out Detection and Recovery" for a discussion of TDR.
- **Improved TDR isolation.** On Windows 8, TDRs are typically isolated to the accelerator view which caused the TDR. On Windows 7, they are broadcast to all accelerator views on the accelerator. See the section "Time-Out Detection and Recovery" for more details.
- **Improved GPU to CPU copy performance.** On Windows 7, copy operations from GPU to CPU take a process-wide lock within the DirectX kernel before blocking. This prevents any GPU work, even on other GPUs, from being submitted by other threads. See the "Swapping Data Between Accelerators" section in Chapter 9, "Working with Multiple Accelerators," for further details of ways to work around this issue. This has been improved in Windows 8 and should result in improved performance, particularly in multi-GPU applications.

These differences might change. Consult the "Parallel Programming in Native Code" blog on MSDN for the latest updates on C++ AMP: *http://blogs.msdn.com/b/nativeconcurrency/*.

Regardless of which operating system you are using, make sure that you have the latest driver for the installed GPU and that your application is running on the GPU hardware. A common mistake is to run your application on a machine with incorrect drivers. If the C++ AMP run time can't find a C++ AMP-capable GPU, it will fall back to either the WARP accelerator (on Windows 8) or the REF accelerator (on Windows 7). Consequently, you will not see the performance you expect.

Time-Out Detection and Recovery

Time-Out Detection and Recovery (TDR) is the mechanism that Microsoft Windows uses to stop a process from overutilizing the GPU and causing the display to become unresponsive. The default behavior of TDR is to reset the accelerator view if a DMA buffer takes longer than two seconds to execute. This will terminate any running C++ AMP code and reset the PC's display if the accelerator is connected to a display.

GPU hardware drivers batch up work based on their resource usage, rather than their expected execution time. This means that batching might result in TDR being triggered if the batched work exceeds the TDR time-out. Using the immediate queuing mode on the *accelerator_view* can break up queued work into smaller batches that will be under the TDR time-out limit.

Other things might also cause a TDR:

- An unrecoverable out-of-memory exception while copying data to the accelerator.
- On Windows 7, another application might cause a TDR because TDRs are broadcast to all applications using the same accelerator (regardless of their accelerator view). On Windows 8,

the typical behavior is for the TDR to be limited to the accelerator view (and therefore application) that submitted the command causing the TDR.

- The device is physically removed from the system.

For more general information on TDR, see the MSDN topic "Timeout Detection and Recovery of GPUs through WDDM" at *http://msdn.microsoft.com/en-us/windows/hardware/gg487368*.

Avoiding TDRs

The best strategy would be to avoid TDRs entirely. For the reasons outlined above, this is not always possible because your application might not have caused the TDR but will still be required to handle it. You can still write your application to minimize the chances of a TDR and deal with the ones that do occur gracefully.

- Design your application for short kernel times, well under the default TDR time-out of two seconds.
- If the data running on your kernel comes directly or indirectly from user input, ensure that the input has been checked and falls within acceptable bounds unlikely to cause a TDR time-out.
- For problems so large that they will exceed the TDR time-out or available memory on the hardware, breaking up the problem into smaller chunks can help. Submitting each chunk as a separate piece of work to the accelerator and using immediate queuing mode can prevent TDR time-outs. The correct chunking approach to use is usually specific to the type of algorithm being implemented. The "Chunking data across multiple C++ AMP kernels" post on the "Parallel Programming in Native Code" blog contains a small case study in chunking up a matrix multiply implementation: *http://blogs.msdn.com/b/nativeconcurrency/archive/2012/03/05/chunking-data-across-multiple-c-amp-kernels.aspx*. This study includes a comparison of several different chunking strategies and their relative performance.

The following sections cover disabling TDRs on Windows 8 and how to handle TDRs that do occur.

Disabling TDR on Windows 8

On Windows 8, it's also possible to use the Direct3D 11 API to create a device with no time-out and then pass this to the *create_accelerator_view()* method to prevent TDR from resetting the accelerator while it's executing your code.

```
#include <d3d11.h>
// ...

IDXGIAdapter* pAdapter = nullptr;       // Use default adapter

unsigned int createDeviceFlags = D3D11_CREATE_DEVICE_DISABLE_GPU_TIMEOUT;
ID3D11Device *pDevice = nullptr;
ID3D11DeviceContext *pContext = nullptr;
D3D_FEATURE_LEVEL featureLevel;
HRESULT hr = D3D11CreateDevice(pAdapter,
```

```
                                        D3D_DRIVER_TYPE_UNKNOWN,
                                        NULL,
                                        createDeviceFlags,
                                        NULL,
                                        0,
                                        D3D11_SDK_VERSION,
                                        &pDevice,
                                        &featureLevel,
                                        &pContext);

    if (FAILED(hr) ||
        ((featureLevel != D3D_FEATURE_LEVEL_11_1) &&
            (featureLevel != D3D_FEATURE_LEVEL_11_0)))
    {
        std::wcerr << "Failed to create Direct3D 11 device" << std::endl;
        return;
    }

accelerator_view noTimeoutAcclView =
    concurrency::direct3d::create_accelerator_view(pDevice);
```

This will prevent TDR time-outs only on GPUs that the OS or other processes, such as the Windows Desktop Manager, are not contending for time on. TDR time-outs will still occur if the GPU is being used for display or has other processes executing code on it. Therefore, it's only worth modifying the TDR behavior of GPUs that are not connected to a display and are dedicated to running code only from your application process.

Detecting and Recovering from a TDR

C++ AMP allows your application to detect and handle a TDR by catching the *accelerator_view_removed* exception. Well-designed applications should do this, especially those that allow users to provide input data that could cause a computation or data copy to trigger a TDR.

The following example shows well-designed production code for calling a C++ AMP kernel. It catches *accelerator_view_removed* exceptions and attempts to rerun the computation on a new *accelerator_view* using immediate mode queuing before finally failing and surfacing the error to the user.

```
std::vector<float> inData(10000);
std::vector<float> outData(10000, 0.0f);
accelerator accel = accelerator();

try
{
    Compute(inData, outData, -1, accel);
}
catch (accelerator_view_removed& ex)
{
    std::wcout << "TDR exception: " << ex.what();
    std::wcout << "  Error code:" << std::hex << ex.get_error_code();
    std::wcout << "  Reason:" << std::hex << ex.get_view_removed_reason();
```

```
        std::wcout << "Retrying..." << std::endl;
        try
        {
            Compute(inData, outData, -1, accel, queuing_mode::queuing_mode_immediate);
        }
        catch (accelerator_view_removed& ex)
        {
            std::wcout << "TDR exception: " << ex.what();
            std::wcout << "  Error code:" << std::hex << ex.get_error_code();
            std::wcout << "  Reason:" << std::hex << ex.get_view_removed_reason();
            std::wcout << "FAILED." << std::endl;
        }
    }
```

The sample does not use the default *accelerator_view* because it's not possible to recover from TDRs on the default view unless your application uses a different accelerator or restarts itself. A non-default *accelerator_view* can be discarded and a new one can be created on the same accelerator.

```
void Compute(std::vector<float>& inData, std::vector<float>& outData, int start,
    accelerator& device, queuing_mode mode = queuing_mode::queuing_mode_automatic)
{
    array_view<const float, 1> inDataView(int(inData.size()), inData);
    array_view<float, 1> outDataView(int(outData.size()), outData);

    accelerator_view view = device.create_view(mode);
    parallel_for_each(view, outDataView.extent, [=](index<1> idx) restrict(amp)
    {
        // Long running kernel that triggers a TDR.
    });
}
```

Many of the samples that accompany this book do not attempt to handle TDRs. This is largely to make the code clearer and simpler to read. In your production code, you should consider trapping *accelerator_view_removed* exceptions whenever a C++ AMP kernel is called.

More details on this topic can be found in the "Handling TDRs in C++ AMP" post on the "Parallel Programming in Native Code" blog: *http://blogs.msdn.com/b/nativeconcurrency/archive/2012/03/07/handling-tdrs-in-c-amp.aspx*.

Double-Precision Support

Although the C++ AMP programming model supports both single-precision (*float*) and double precision types, double-precision support is an optional DirectX driver feature. Depending on the level of driver support provided by the GPU vendor, your application will have one of the following levels of double-precision support: full, limited, or none.

Limited Double Precision

The Windows Display Driver Model (WDDM) 1.1 has only limited double-precision support. Specifically, the following operations are not supported:

- Fused Multiply Add (FMA)
- Division
- Reciprocal
- Casting *int* or *unsigned int* to *double*
- Casting *double* to *int* or *unsigned int*

C++ AMP provides the *accelerator::supports_limited_double_precision* Boolean property to allow your application to query the available hardware. See Chapter 9, "Working with Multiple Accelerators," for more information about selecting accelerators.

Full Double Precision

Windows 8 supports the WDDM 1.2 model, which has none of the limitations of WDDM 1.1 outlined above. Double-precision support is still an optional driver feature, so your GPU requires a driver that supports it. The *accelerator::supports_double_precision* Boolean property can be used to query accelerators to see whether they support full double precision.

In the case where your application tries to execute a double-precision operation on an accelerator with no double-precision support or uses one of the unsupported operations on a card with limited double-precision support, a *concurrency::runtime_exception* will be thrown. If you encounter unexpected issues when trying to use double precision, make sure that you have the vendor's latest driver installed. Even if you are running your application on Windows 8 and the GPU hardware supports double precision, without the correct driver, the C++ AMP run time will not be able to use it.

Also remember that just like CPUs, GPU double-precision operations are much slower than single-precision ones. How much this varies depends on your hardware. You should use double-precision types only if your application really requires them.

Debugging on Windows 7

Chapter 6, "Debugging," covers how to debug applications locally when developing on Windows 8. This section describes how to set up remote debugging so that you can develop on a Windows 7 machine and debug your C++ AMP application running on a Windows 8 machine or virtual machine (VM).

This guide assumes that you have already set up the following:

- A machine or VM running Windows 8 that is on the same network as your development machine. For the purposes of these instructions, this machine is called WIN8REMOTE.

- The Windows 8 machine has a folder, C:\shared\, that is shared as \\WIN8REMOTE\shared\ and can be written to from the local machine.
- You are running a 64-bit operating system on your local machine and building a 64-bit application for debugging.

Configure the Remote Machine

The following steps provide a one-time setup process to configure your remote machine for debugging:

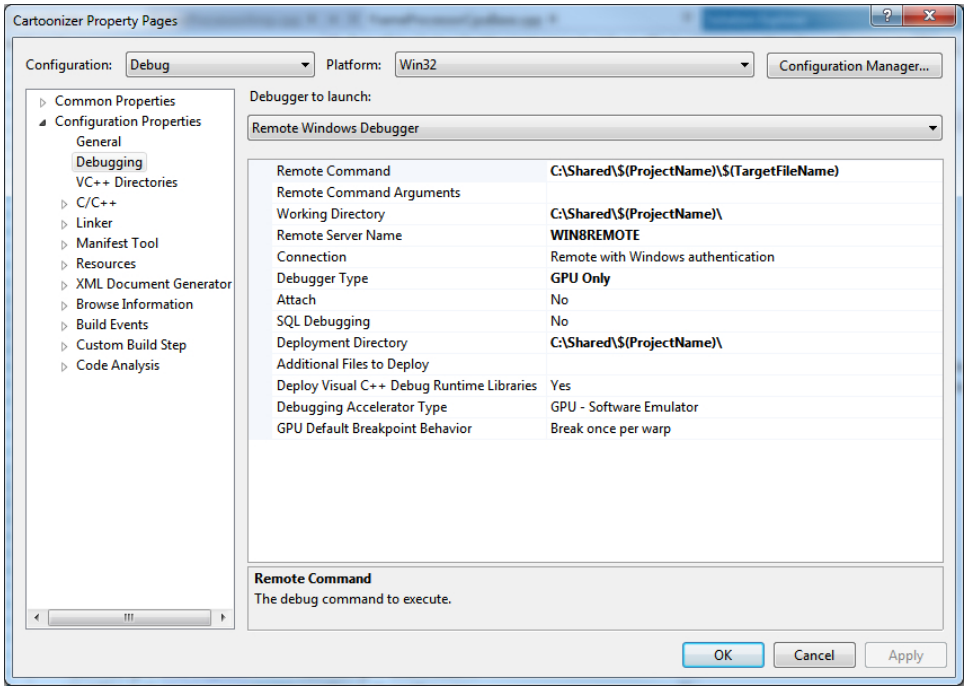
1. Install the Windows SDK on the remote machine. This is available on MSDN at *http://msdn.microsoft.com/en-us/windows/hardware/hh852363*.
2. Copy the files in "C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\Remote Debugger\x64\" on your local machine to \\WIN8REMOTE\shared\debugger. This path will change if your target application is 32-bit.
3. Log on to WIN8REMOTE and run "C:\shared\debugger\msvsmon.exe." When the Remote Debugging Configuration dialog box opens, click the Configure Remote Debugging button to configure the firewall.

Configure Your Project

Each project you want to debug remotely must be configured. Open the solution on your local machine and follow these steps:

1. Open the project properties dialog by either opening the Project | Properties menu item or right-clicking the project in the Solution Explorer and selecting the Properties menu. Make sure that you have the correct configuration and platform selected because the changes made in the following step apply per configuration and platform.
2. In the Properties dialog, click the Debugging tab and make the following changes:
    a. Select Remote Windows Debugger in the Debugger launch drop-down box.
    b. Set Remote Command to "C:\shared\$(ProjectName)\$(TargetFileName)."
    c. Set the Working Directory to "C:\shared\$(ProjectName)\."
    d. Set the Remote Server Name to "WIN8REMOTE."
    e. Set the Debugger Type to "GPU Only."
    f. Set the Deployment Directory to "C:\shared\$(ProjectName)\."

You can also set the accelerator type and GPU default breakpoint behavior. These settings were described in Chapter 6. The following image shows the dialog configured as described in this section.



Deploy and Debug Your Project

You should now be able to deploy and debug your project:

1. Right-click the project in the Solution Explorer and select Deploy. This will copy the executable and any associated binaries to the working directory on the remote machine.
2. Start the debugger by pressing F5 or by using one of the other methods described in Chapter 6.

Your application should start on the remote machine, and you can debug it as described in Chapter 6.

Additional Debugging Functions

Chapter 6 covered how to debug C++ AMP applications using the Microsoft Visual Studio 2012 debugger. This section covers some additional functions you can use when debugging.

```
void direct3d_abort() restrict(amp)
```

The *direct3d_abort* function aborts the execution of the kernel. This raises a *runtime_exception* with the error message "Reference Rasterizer: Shader abort instruction hit".

```
void direct3d_printf(const char* formatString, ...) restrict(amp)
```

The *direct3d_printf* function takes a format string and optional list of up to six parameters. The formatted output is printed to the Visual Studio output window.

```
void direct3d_errorf(char* formatString, ...) restrict(amp)
```

The *direct3d_errorf* function takes a format string and optional list of up to six parameters. The formatted output is written to the Visual Studio output window. The function also throws a *runtime_exception* with the same error message.

These functions are available only if all of the following conditions are satisfied:

- The code must be compiled in a debug configuration; the *_DEBUG* preprocessor definition is set.
- Debugging and these additional debugging functions are supported only on Windows 8.
- The kernel must be invoked on an *accelerator_view* of an *accelerator* that supports the *printf*, *error*, and *abort* functions. For the Visual Studio 2012 release, only the REF accelerator supports these functions. This might change if GPU hardware vendors add this support to their drivers.
- The maximum number of allowed parameters for *direct3d_printf* and *direct3d_error* is seven, including the *formatString* parameter. Note that C++ AMP supports neither functions with variable numbers of parameters nor the *char* type. The functions described here are implemented as compiler intrinsic functions and therefore are exempt from this limitation.
- There is no support for auto widening/narrowing type conversion of parameters passed to these functions. For example, calling *direct3d_printf("%lf", 2.0f)* results in incorrect output. Similar code written on the CPU would convert the float value to a double.

Remember that C++ AMP kernels run asynchronously, so these debug functions will be executed at some point after the kernel has been dispatched and before it has completed. This means that the code running on the CPU might have finished executing the *parallel_for_each* associated with the debug function when the output appears or an exception is raised.

Deployment

Deploying your Application

Understanding your application's dependencies is critical when trying to successfully deploy your application on other machines.

C++ AMP dynamically links vcamp110.dll; it can't be statically linked. The vcamp110.dll binary has a dependency on msvcr110.dll and msvcp110.dll; these are also dynamically linked. Debug versions of each of these dynamic-link libraries (DLLs) are also available. They have the same filenames with a 'd' appended—for example, vcamp110d.dll. In general, you should not deploy debug versions of these DLLs except for remote debugging. For Visual Studio 2012 RTM, vcamp110.dll (and its dependencies)

are part of the "Microsoft Visual C++ 2012 Redistributable Package," also known as "VCRedist." To deploy Visual C++ redistributable files, you have several choices:

- Use the Visual C++ Redistributable Package (VCRedist_x86.exe or VCRedist_x64.exe) that is included in Visual Studio.
- Include the Redistributable Merge Modules in your application's installer.
- Install the required Visual C++ DLLs by copying them into the folder that contains the executable application file.

For further details, see the MSDN topic "Deploying Desktop Applications (Visual C++)" at *http://msdn.microsoft.com/en-us/library/zebw5zk9(v=vs.110).aspx*.

The Microsoft Visual C++ 2012 Redistributable Package can be downloaded from MSDN.

Running C++ AMP on Servers

C++ AMP applications can be used in a variety of contexts. This book has focused on writing applications that run on the client, but it's also possible to run C++ AMP code on servers. This section discusses how to run in different server environments.

Consult the "Parallel Programming in Native Code" blog on MSDN for the latest updates on C++ AMP support on servers and the cloud: *http://blogs.msdn.com/b/nativeconcurrency/*.

Enumerating C++ AMP-Capable Devices

Before attempting to run your application on a new computer, it's best to check that there is at least one C++ AMP-capable device available.

Build the ShowAmpDevices sample and copy it to the new computer. Running this sample also requires that C++ AMP is installed, so the VCRedist must be installed on the machine. See the section of this chapter entitled "Deploying Your Application" for details on installing C++ AMP applications. Typical output from ShowAmpDevices is shown here:

```
Found 2 accelerator device(s) that are compatible with C++ AMP:
  1: NVIDIA GeForce GTX 580, has_display=true, is_emulated=false
```

On computers with no C++ AMP-capable devices, the message "No accelerators found that are compatible with C++ AMP" is displayed.

Running with XPDM Graphics Devices Present

Windows XP display driver model (XPDM) is the display driver model used by Windows XP, and it's still supported by Windows Vista and Windows Server 2008. WDDM is the newer display driver model introduced with Windows Vista. These two driver models can't be used together.

This might cause problems on machines with older display hardware that supports only XPDM display drivers. This is common in servers that ship with an integrated VGA adaptor, which supports only

an XPDM driver. In this case, the XPDM driver loads, preventing any other cards from loading WDDM drivers. C++ AMP will not be able to detect any DirectX 11 hardware.

For Windows 7 and Windows Server 2008 R2, the workaround for this is to remove or disable the XPDM device. Disabling an integrated graphics adapter is done by modifying the BIOS settings. The required changes vary between vendors. Typically, the BIOS contains a setting to either disable the graphics adapters or change the boot priority of the adapters present. Setting one of the WDDM-capable adapters to be first priority will effectively disable any XPDM adapters.

On Windows 8 and Windows Server 2012, this limitation has been addressed and the XPDM driver model is no longer supported. Instead, a WDDM Microsoft Basic Display Adapter replaces the XPDM Standard VGA driver. This means that C++ AMP will work without having to remove or disable any XPDM devices.

Running without a Connected Display

C++ AMP does not require a display to be connected to your DirectX 11 card in order for it to be detected as a C++ AMP accelerator. The only requirement is that its DirectX driver advertise the device as a Windows display device. Cards that have a display output advertise themselves whether or not a display is connected.

Running on True Headless Servers

True headless servers do not require a keyboard, mouse, or display or graphics adapter. In the case of C++ AMP, that would mean a server that has hardware that has a DirectX 11 driver but does not have a display connector. Such hardware does not advertise itself as a Windows display device. The vast majority of devices do not fall into this category and should work as expected with C++ AMP. However, on Windows 7 and Windows Server 2008 R2, devices that do not advertise themselves as a Windows display device can't be used by C++ AMP unless there is another WDDM device present that does advertise itself as a display device. The presence of this additional card will cause DirectX to initialize and recognize all devices, regardless of whether they advertise themselves as being display devices.

Windows 8 and Windows Server 2012, support the new WDDM v1.2 driver model, which has better support for headless servers. This means that true headless servers running Windows 8 and Windows Server 2012 fully support C++ AMP.

Running as a Service or under Session 0

Windows services run under Session 0. This is a noninteractive session that is isolated from other user sessions. On Windows 8 and Windows Server 2012, C++ AMP code can run in Session 0. This is a supported scenario.

On Windows 7 or Windows Server 2008 R2, C++ AMP code can also run under Session 0. However, this is not a supported scenario. Although it might work, Microsoft hasn't tested it. An alternative approach would be to refactor your application into a C++ AMP component that runs under an interactive session and a service running under Session 0 that calls it.

For more information on the compatibility requirements for running under Session 0, see the MSDN topic "Application Compatibility: Session 0 Isolation" at *http://msdn.microsoft.com/en-us/library/windows/desktop/bb756986.aspx*.

C++ AMP and Windows 8 Windows Store Apps

The Microsoft patterns and practices Hilo project is an example of how to develop and deploy a Windows 8 Windows Store App using C++ and XAML. The Hilo application includes C++ AMP image-processing features similar to the Cartoonizer application discussed in Chapter 10, "Cartoonizer," but with a Windows 8 Windows Store App user interface. The documentation describes how to integrate C++ AMP into a Windows 8 Windows Store App and deploy the app.

The Hilo documentation and source code can be found on CodePlex: *http://hilo.codeplex.com/*.

Using C++ AMP from Managed Code

Sometimes you might want to call C++ AMP code from managed code. C++ AMP can't be called directly from managed code, but there are several ways your managed code can take advantage of C++ AMP. These are not covered in great detail here because there are no significant differences when calling C++ AMP code from managed code beyond the general requirements for calling native code.

From a .NET Application, Windows 7 Windows Store App or Library

To call C++ AMP code from a .NET application or library, use P/invoke to call functions exported by a native DLL. These native functions can then use C++ AMP. The "How to use C++ AMP from C#" on the PFX Team blog describes this approach: *http://blogs.msdn.com/b/pfxteam/archive/2011/09/21/10214538.aspx*.

On Windows 7 or for a Windows 8 desktop application, it's also possible to use C++/CLI to wrap your native C++ library with a managed wrapper that can be referenced by a .NET project. The MSDN topic "How to: Wrap Native Class for Use by C#" contains a simple example of how to do this: *http://msdn.microsoft.com/en-us/library/ms235281.aspx*.

For Windows 8 Windows Store Apps, C++/CLI is not supported. Windows 8 Windows Store Apps written in managed code can consume Windows Runtime Components written in C++. See the MSDN topic "Creating Windows Runtime Components" for further detail: *http://msdn.microsoft.com/en-us/library/windows/apps/hh441572(v=vs.110).aspx*.

From a C++ CLR Application

C++ AMP is not supported when compiling with /clr. This section and the one that follows summarize two approaches for using C++ AMP from C++ CLR code.

You can refactor your application code into two separate projects: a C++ CLR application project and a native DLL containing the C++ AMP code. The C++ CLR project references the native project, thereby working around the limitation that code compiled with the /clr flag can't use C++ AMP.

The "How to Use C++ AMP from C++ CLR app" post on the "Parallel Programming in Native Code" blog walks through this approach: *http://blogs.msdn.com/b/nativeconcurrency/archive/2011/12/21/how-to-use-c-amp-from-c-clr-app.aspx*.

From within a C++ CLR Project

Rather than creating two separate projects to contain the managed C++/CLI code and native C++ AMP code, as described in the previous section, it's also possible to create a mixed assembly. Mixed (native and managed) assembly projects contain source files that are configured to compile as native code alongside those configured to compile as C++ CLR. The linker combines both the native and managed objects into an assembly that can be referenced by .NET projects.

The "Using C++ AMP code in a C++ CLR project" post on the "Parallel Programming in Native Code" blog walks through this approach: *http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/05/using-c-amp-code-in-a-c-clr-project.aspx*.

Summary

This chapter covered a variety of advanced topics to help you get the most out of using C++ AMP. It also covered strategies to help you deploy your C++ AMP applications in a variety of environments. C++ AMP is designed to target not only desktops but also the newest hardware featuring Windows 8 Windows Store Apps and server hardware in data centers and the cloud.

APPENDIX

Other Resources

More from the Authors

The authors blog about C++ AMP and other programming related topics on their personal blogs: *http://www.gregcons.com/KateBlog/* and *http://ademiller.com/tech*.

The authors also maintain a webpage (*http://gregcons.com/cppamp/*) containing book-related news and information. This is a good place to look for additional material, errata, and upcoming speaking engagements. The sample code for the book is also available on CodePlex at *http://ampbook.codeplex.com/*.

Microsoft Online Resources

The C++ AMP product team contributes frequently to the "Parallel Programming in Native Code" blog, and it's a good place to find the latest updates and news on C++ AMP. It includes deep dives on many topics outside the scope of this book:

http://blogs.msdn.com/b/nativeconcurrency/

The MSDN forum for Parallel Computing in C++ and Native Code is a good place to ask questions related to C++ AMP:

http://social.msdn.microsoft.com/Forums/en-US/parallelcppnative/threads

The MSDN Library also includes information on C++ AMP, including extensive API reference material not covered in this book:

http://msdn.microsoft.com/en-us/library/hh265137(v=vs.110)

Download C++ AMP Guides

The C++ AMP open specification, at well over 100 pages, is the definitive reference, and you can find a link to it from here:

http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/03/c-amp-open-spec-published.aspx.

The C++ AMP team has written a number of guides for developers who are familiar with other GPU programming models and would like to learn C++ AMP. If you are familiar with one of these approaches, you may want to read the appropriate guide in conjunction with this book as listed below:

- **C++ AMP for the CUDA Programmer**

  http://blogs.msdn.com/b/nativeconcurrency/archive/2012/04/11/c-amp-for-the-cuda-programmer.aspx

- **C++ AMP for the OpenCL Programmer**

  http://blogs.msdn.com/b/nativeconcurrency/archive/2012/04/10/c-amp-for-the-opencl-programmer.aspx

- **C++ AMP for the DirectCompute/HLSL Programmer**

  http://blogs.msdn.com/b/nativeconcurrency/archive/2012/04/09/c-amp-for-the-directcompute-programmer.aspx

Code and Support

The C++ AMP team has also delivered a number of samples that highlight different implementations of common problems and different aspects of C++ AMP. You can find a full list of these samples here:

http://blogs.msdn.com/b/nativeconcurrency/archive/2012/01/30/c-amp-sample-projects-for-download.aspx

In addition, several libraries are being developed for C++ AMP—for example, RNG, BLAS, LAPACK, FFT, and others. These are active projects; and at the time of this writing they were in the pre-Alpha stage of development. You should check their project home pages for the latest project status updates by following the links available here:

http://blogs.msdn.com/b/nativeconcurrency/archive/2012/05/19/libraries-for-c-amp.aspx

You can incorporate these libraries into your applications, and they are also a great source for examples of C++ AMP implementations of common algorithms. Use them in addition to the sample code and case studies that accompany this book.

To ask questions about C++ AMP and other aspects of parallel computing with C++, see the "Parallel Computing in C++ and Native Code" MSDN forum, where the product team provides authoritative answers:

http://social.msdn.microsoft.com/Forums/en-US/parallelcppnative/threads

Training

If you prefer learning by watching short video recordings, you'll find a growing list under the C++ AMP Channel9 tag:

http://channel9.msdn.com/Tags/c++-accelerated-massive-parallelism

At the time of writing, there is only one formal training course offered by Acceleware. For dates, location, pricing, and other details, please visit their page:

http://www.acceleware.com/cpp-amp-training

Index

A

abstract base class, in NBody case study, 30
accelerator
  about, 48
  accelerator_view and, 47, 64
  automatically synchronizing data on, 142
  constants using, 48
  creating views, 48–49
  default, 206–207
  efficient global memory access, 146–148
  functions using, 48
  in GPU vs. accelerator memory in array, 64
  REF, 205
  setting for running parallel_for_each(), 106
  supporting double precision support, 162
  WARP, 205, 218, 220
accelerators, working with multiple
  about, 203
  braided parallelism, 219
  choosing
    about, 203
    default accelerator, 206–207
    enumerating accelerators, 203
  dynamic load balancing, 216–218
  falling back to CPU, 220–221
  in Cartoonizer case study
    forked pipeline strategy, 249–252
    strategy for, 246–248
  swapping data between accelerators, 211–214
  using more than one GPU, 208–211, 212
accelerator_view
  accelerator and, 48
  arrays bounded to, 46
  default accelerator and, 207
  Direct3D device interop and, 276–277
  in arrays, 47
  omitting setting code for debugging, 108–109
algorithms
  coding in C++ AMP, 220
  CPU, 178
  designing tiled, 74
  edge detection, used by Cartoonizer, 241
  efficient global memory access and, 147
  finding bottlenecks in, 90
  modifying simple into tiled
    tile barriers and synchronization, 74–76
    tiled matrix multiplication, 76–77
    using tile_ static memory, 69
    writting simple algorithms, 68–69
  tiling n-body, 85–86
aliasing
  parallel_for_each() invocations, 138–140
  performance impact of, 141
AMD
  Compute Units, 138
  wavefronts on, 4
Amdahl's Law, 7
applications
  as candidates for parallel processing, 6
  power requirements vs. battery life in, 3
applications, deploying, 303
arithmetic operators, supported by short vector types, 259
arithmetic reduction, 179
arrays
  about template, 45–47
  accelerator memory, 64
  as read-only, 142
  captured containers as, 138
  choosing tile size and size of, 80
  constructors for, 47
  dimensions and number of, 50, 64, 81
  Direct3D buffer interop and, 277–278
  extents in, 47, 50, 79
  initializing, 290

multiplication of, 70
number of dimensions in, 45
of structs vs. structs of arrays, 38, 149–151
overhead from first using, 128
relation to array_view, 51
relation to extent and index, 41
relation to index, 48
using staging, 145–146
vs. textures, 270–271
array_view, 18
about, 51–52
accelerator memory, 64
as parameter for parallel_for_each(), 56
as read-only, 142
captured containers as, 138
extent in, 50
going out of scope, 142
in parallel_for_each() invocations, 140
methods in, 52
relation to array, 51
relation to index, 48
Simple C++ AMP algorithm with, 179
synchronizing data automatically, 60, 142
Asynchronous Agents Library, 11
asynchronous copies, using overlapping, 144–145
asynchronous programming, 220
atomic operations, 292–294
automatically synchronizing data, 60, 142
Autos window, 108
auto-vectorization and auto-parallelization of code, 9

B

bank conflicts, eliminating, 193
barriers, 74–75, 89, 110, 114, 164, 196–197
battery life vs. power requirements, in applications, 3
best practices
additional functions for debugging, 302
atomic operations, 292–294
dealing with tile size mismatches, 283–284
debugging on Windows 7, 300–302
configuring project, 301
configuring remote machine, 301
deploy and debug project, 302
deployment
C++ AMP and Windows 8, 306
C++ CLR application, 307
C++ CLR project, 307
deploying applications, 303
library, 306
.NET Application, 306
running C++ AMP on servers, 304–306
using C++ AMP from managed code, 306
Windows store application, 306
double precision support
full, 300–302
limited, 300
features in Windows 8, 295
function objects vs. lambdas, 291–292
handling truncated elements
with edge treads, 287
with sections, 288–289
in debugging GPU, 108–109
initializing arrays, 290
tiles
padding, 285–286, 290
truncating, 286–288
Timeout Detection and Recovery, 296
avoiding, 297
detecting and recovering from, 298
disabling on Windows 8, 297–298
bitwise operators, supported by short vector types, 259
bottlenecks, finding in algorithms, 90
braided parallelism, 219
breakpoints
in debugging, 102–103
stopping execution when setting parallel_for_each(), 107

C

C++11
destructors, 142
lambda expressions in, 52
cache, GPU programmable, 64
cache size, clock speed and, 3
Call Stack window, 108
C++ AMP
about, 15–19
calculations, 36–39
capable devices, enumerating, 304
interface, xv
Windows 8 and, 306
resources for, 309–311
running on servers, 304–306

using from
managed code, 306
vs. C++, 17–19
capture clauses, in lambas, 55
captured containers, 138
Cartoonizer case study
about features of Cartoonizer, 255
about prerequisites, 224–225
braided parallelism in, 219
edge detection algorithms used by Cartoonizer, 241
performance of Cartoonizer, 250
pipeline Cartoonizing stage
about, 236
IFrameProcessor implementations, 239–245
ImageCartoonizerAgent class, 236–238
pipeline implementation, 232–236
data structures in, 229–230
OnBnClickedButtonStart method, 231
running on multiple GPUs, 211
running sample for, 224–226
structure of, 228–229
using multiple accelerators in, 246–251
forked pipeline strategy, 249–252
using textures and short vectors in, 270
cascading reductions, 198–200
C++ CLR
deployment from
application, 306
project, 307
C, CPU-parallelism and, 10
C++, CPU-parallelism and, 10
clamping behavior, in writing data, 268
classes
rules for using instances of, 58
clock speed, cache size and, 3
CLR Thread Pool, 12
code and support, for C++AMP, 310
colors, short vector types and, 260
compilers, reordering execution of instructions, 165
compiling
Just-In-Time (JIT). *See* Just-In-Time (JIT)
of kernel, 141
compound bitwise assignment operators, supported by short vector types, 259
compound operation functions, 61
computation, optimizing, 161–162
avoiding divergent code, 158–161
choosing the appropriate precision, 161–162
costing mathematical operations, 163
loop unrolling, 164–165, 195–198, 200
queuing modes, 168–169
using barriers for, 164
Compute Units (CUs), 138
concurrency::fast..math-namespace, 61
concurrency::precise_math namespace, 61
Concurrency Runtime (ConcRT), PPL and, 11–12
Concurrency Visualizer
channels in window of, 134
examining memory access patterns, 149
markers in reduction performance case study, 175–176
using, 90–93, 131–135
Concurrency Visualizer SDK, 137–138
constant memory, 155–156
constants
and constraints, in reduction performance case study, 174–175
using accelerator, 48
const keyword, 18, 60, 66, 142, 142–144
constructors
for arrays, 47
passing accelerator default view to, 48
copy_async() function, using in swapping accelerators, 214–215
copy() function, 59–61
copying
between CPU and GPU, 59–61
data to and from textures, 264–266
efficiently to and from GPU, 141–146
about, 141
leaving data on GPU, 144
removing unnecessary copies, 142–144
using overlapping asynchronous copies, 144–145
using staging arrays, 145–146
CPU
accelerator, 205
algorithms, 178
architecture, 4
copying between GPU and, 59–61
debugging, 102. *See also* debugging
enabling breakpoints, 102–103
falling back to, 220–221
multicore machines and, 2
reordering execution of instructions, 165
turning debugging on, 102
vs. GPU, 2–3

CPU-parallelism
languages supported by, 10
requirements for, 14
technologies for, 8–13
ConcRT, 11–12
OpenMP, 10–11
Task Parallel Library, 12
vectorization, 8–10
WARP, 12–13
CreateTasks() function, 38–39
C++ Standard Library, Thrust and, 13
CUDA C language, 13
CUDA (Compute Device Unified Architecture)
about, 13
CUs (Compute Units), 138
C vs. C++ AMP, 16
C++ vs. C++ AMP, 16–18

D

datasets, tile size and, 79
data storage, for textures (texels), 262–264
data structures
in C++ AMP calculations, 37–38
in CPU calculations, 29–30
data types
C++ AMP-compatible function, 57–59
forbidden, 58
lambda, 57–59
Debug build, running, 218–219
debugging
about, 101
about preparing for, 123
CPU, 102
freezing and thawing threads, 121–122
GPU
basics of, 108–110
choosing, 102, 103
detecting race conditions, 110–111
familiar windows and, 108–109
stopping execution when setting parallel_for_each() breakpoints, 107
taking more control of threads, 121–122
turning on, 103
using Debug Location Toolbar, 109
using reference accelerators, 101, 106–108
using threads window, 113–114, 116
on Windows 7, 300
parallel_for_each(), 115
seeing threads, 112–116
displaying GPU Threads Window, 113–114
filtering threads, 120–122
flagging threads, 119
grouping threads, 119
turning on thread markers, 113
using Parallel Stacks window, 115–116
using Parallel Watch Window, 117–118, 121
using Run To Cursor command, 123
tips and tricks, additional functions for, 302
turning on breakpoints, 102–103
using reference accelerators, 22
Debug Location Toolbar, GPU debugging using, 109
Debug Toolbar, turning on thread-related information, 113
decrement and increment operators, supported by short vector types, 260
default accelerator, 46–49, 56, 59, 106–108, 205–207, 299
deployment
C++ AMP and Windows 8, 306
deploying applications, 303
running C++ AMP on servers, 304–306
dimensions, number of arrays and, 45, 50, 64, 81
Direct3D
about, 13
buffer interop and array, 277–278
device interop and accelerator_view and, 276–277
mapping between C++ AMP and types in, 278
resource interop and textures, 277–280
supporting driver model for platform, 12–13
direct3d_abort function, 302
direct3d_errorf function, 302
Direct3D High Level Shader Language (HLSL)
graphics applications and, 257
scalar types using HLSL, 258
direct3d_printf function, 302
DirectCompute
C++ AMP implementation on, 141
DirectCompute API, to support GPGPU, 13
DirectCompute JIT, 164
Direct Memory Access (DMA), staging buffers and, 145
DirectX
double precision support
full, 300
limited, 300
interop
about, 275

accelerator_view and Direct3D device interop, 276–277
Direct3D buffer interop and array, 277–278
texture and Direct3D resource interop, 277–280
using, 280–282
DirectX 11
areas of responsibility in sample framework, 28–29
C++ AMP accelerator and, 48
drivers for portability of executables, 19
scalar types using HLSL, 258
staging buffer, 145
support for, 22
UI code in, 28
DirectX SDK, 224
discard_data() method, 18, 142, 144
divergence, avoiding, 158–161
divergence, minimizing, 192–193
double precision support
accelerator supporting, 162
DirectX, 299
full, 300
limited, 300
dynamic load balancing, 216–218

E

edge threads
handling truncated elements with, 287
ElapsedTime() function, 177
eliminating bank conflicts
reduction performance case study
simple with array_view, 179
emulated accelerator, debugging using, 101
enumerating accelerators, 203–204
equality operators, supported by short vector types, 259
exceptions
for accelerator setting, 106
setting GPU debugging, 111
executables, portability of, 19–20
exponent functions, 61
extents
in arrays, 47, 50, 79
in array_view, 50
in choosing tile size, 80
relation to array and index, 41
tiled, 66–67, 68, 69
tile() function and, 87

F

fast_math namespace functions
about, 161–162
list of, 163
fences, memory, 166–167, 197
filtering threads, 120–122
financial tracking, simulation, and prediction, as candidate for parallel processing, 6
flagging threads, 119
for_each () function, 54
forked pipeline strategy in using multiple accelerators, 249
Fortran, CPU-parallelism and, 10
4D spatial vectors, 260
"free lunch", 2, 14
freezing threads, 121–122
full double precision, DirectX, 300
function objects vs.
lambdas, 291–292
functions
copy(), 59–61
copy_async(), using in swapping accelerators, 214–215
debugging, 302
discard_data(), 18, 142, 144
fast_math namespace, 161–162
for_each (), 54
GetGpuAccelerators(), 39
in arrays, 52
IsAmpAccelerator(), 39–40
kernel, 56–57, 60, 61
marked with restrict(amp), 57–59
Math Library, 61–62
parallel_for, 35
parallel_for_each(), 87
about, 55–56
aliasing invocations, 138–140
debugging tiled, 115
extent used in, 41, 54
lambdas used in, 41
setting accelerator for running, 106
tiled, 67–68
tile_static memory and, 89
using array_view as parameter, 56, 66–67
precise_math namespace, 161–162
section(), 144
synchronize(), 142
tile(), 87, 88
using accelerator, 48
Futures pattern, 220

G

gaming, as candidate for parallel processing, 6
GetGpuAccelerators() function, 39
GFlops, 90, 96
global memory access, efficient accelerator, 146–148
glyphs in Step Over command, 103
GPGPU (GPU programming)
  about, 3
  arrays of structs vs. structs of arrays, 149–151
  C++ AMP and, 15–16
  DirectCompute API to support, 13
  evolving of applications, 14
  structs in, 38
GPU
  about, 2
  architecture, 4
  copying, between CPU and, 59–61
  copying efficiently to and from, 141–146
    about, 141
    leaving data on GPU, 144
    removing unnecessary copies, 142–144
    using overlapping asynchronous copies, 144–145
    using staging arrays, 145–146
  debugging
    basics of, 108–110
    choosing, 102, 103
    detecting race conditions, 110–111
    familiar windows and, 108–109
    stopping execution when setting parallel_for_each() breakpoints, 107
    taking more control of threads, 121–122
    turning on, 103
    using Debug Location Toolbar, 109
    using reference accelerators, 22, 101, 106–108
    using threads window, 112–116, 116
  executing threads in kernel, 138
  performance and parallelism, 5–6
  programmable cache, 64
  speeding up time windows and, 6–7
  support of double precision, 3
  tile size and warp in arrangement of, 79
  timing in execution on, 78
  turning debugging on, 102
  vs. CPU, 2–3
  WARP and, 12–13
  working with multiple accelerators on multiple, 208–211, 212
GPU hardware, optimizing memory access performance, 147
GPU parallelism
  aware profiler for, 14
  requirements for, 14
  technologies for, 13
GPU programming (GPGPU)
  about, 3
  arrays of structs vs. structs of arrays, 149–151
  evolving of applications, 14
  structs in, 38
GPU Threads Window, displaying, 113–114
graphics interop
  Direct3D High Level Shader Language (HLSL) graphics applications and, 257
  DirectX
    about, 275
    accelerator_view and Direct3D device interop, 276–277
    Direct3D buffer interop and array, 277–278
    texture and Direct3D resource interop, 277–280
    using, 280–282
  HLSL intrinsic functions, 274
  mapping between C++ AMP and, 278
  norm and unorm, 258, 263
  short vector types
    about, 259–260
    accessing vector components, 260
    template metaprogramming, 260–262
    texel, 262–264, 278–279
    using in Cartoonizer case study, 270
  textures
    about, 262
    copying data to and from, 264–266
    Direct3D resource interop and, 277–280
    maximium size of, 270
    reading from, 266–267
    read-write, 268
    texels for data storage of, 262–264, 278–279
    using in Cartoonizer case study, 270
    vs. arrays, 270–271
    writeonly_texture_view, 269–270
    writing to, 267–268
graphics namespace, 259
grouping threads, 119

H

HaaS (Hardware as a Service), 20
Hardware as a Service (HaaS), 20
heterogeneous computing
  history of performance improvements and, 1–2
  platforms for, 2–3
heterogeneous parallel computing, xv–xvi
heterogeneous supercomputers, about, 2
High Level Shader Language (HLSL)
  about, 13, 18
  Direct3D
    graphics applications and, 257
    scalar types using HLSL, 258
  intrinsic functions, 274
  Just-In-Time and, 128
  vector types, 260
high-resolution performance timer API, 129
hints
  in C++ AMP, 18
  reminder from IntelliSense, 61
HLSL (High Level Shader Language), 13, 18

I

IaaS (Infrastructure as a Service), 20
image processing, as candidate for parallel processing, 6
image processing, in Cartoonizer case study
  about features of Cartoonizer, 255
  about prerequisites, 224–225
  edge detection algorithms used by Cartoonizer, 241
  performance of Cartoonizer, 250
  pipeline Cartoonizing stage
    about, 236
    IFrameProcessor implementations, 239–245
  pipeline implementation
    data structures in, 229–230
  running sample for, 224–226
  structure of, 228–229
  using multiple accelerators in
    forked pipeline strategy, 249–252
    strategy for, 246–248
  using textures and short vectors in, 270
increment and decrement operators, supported by short vector types, 260
index
  relation to array, 41, 50
  relation to array_view, 50
  relation to extent, 41
  tiled, 67–68, 69
Infrastructure as a Service (IaaS), 20
Integrate() function
  in NBody case study, 30, 31, 34, 40–42
  in tiling NBody case study, 85–86, 86
IntelliSense, hint reminder, 61
intrinsic functions, HLSL, 274
intrinsics, using, 9–10
IsAmpAccelerator() function, 39–40

J

Just-In-Time (JIT)
  compilation overhead, 141
  compiling of kernels, 128
  loop unrolling and, 164
  read-only resources and, 142

K

kernel
  compiling of, 128
  function, 56–57, 60, 61
  GPUs executing threads in, 138
  measuring performance of, 129–131
  structuring, 75
__kernel_stub(), 108

L

lambda(s)
  as parameter for parallel_for_each() function, 41
  calling, 108
  compatible data types, 57–59
  expressions, 54
  in arrays, 52
  recognizing pattern for, 55
  syntax, 53
  using in parallel_for_each, 54
  vs. function objects, 291–292
library
  deployment from, 306
limited double precision, DirectX, 300
load balancing, dynamic, 216–218
log functions, 61
loop unrolling, 164–165, 195–198, 200

M

managed code, using C++ AMP from, 306
manipulation functions, 61
master-worker pattern, 218
mathematical operations, costing, 163
Math Library functions, 61–62
matrix multiplication, 70–72, 75, 76–77, 291
memory access patterns
  tile static memory access. *See also* tile_static memory
memory access patterns, optimizing, 138
  about, 138
  aliasing
    parallel_for_each() invocations, 138–140
    performance impact of, 141
  array of structs vs. struct of arrays, 38, 149–151
  arrays of structs vs. structs of arrays, 149–151
  constant memory, 155–156
  copying to and from GPU efficiently, 141–146
    about, 141
    leaving data on GPU, 144
    removing unnecessary copies, 142–144
    using overlapping asynchronous copies, 144–145
    using staging arrays, 145–146
  efficient accelerator global memory access, 146–148
  occupancy and registers, 157–158
  texture memory, 156. *See also* textures
  tile static memory access, 152–155
memory fences, 166–167, 197
methods. *See* functions
Microsoft Basic Render Driver (WARP accelerator), 205
Microsoft Concurrency Runtime (ConcRT), PPL and, 11–12
Microsoft online resources, 309
Microsoft Visual C++
  precise and fast compiler flags, 163
  support of OpenMP, 10–11
  website for book on parallel programming with, 225
Microsoft Visual C++ 2012 Redistributable Package (VCRedist), 303
Microsoft Visual Studio 2012
  and C++ AMP, 16, 48
  auto-vectorization and auto-parallelization in, 9
  Concurrency Visualizer and, 90
  debugging using, 101, 102, 104, 110
  reference accelerator in, 22
  supporting C++ AMP applications, 15
  supporting vectorization in, 8
  version needed for reduction performance case study, 175
Microsoft Visual Studio Concurrency Visualizer
  channels in window of, 134
  examining memory access patterns, 149
  markers in reduction performance case study, 175–176
  using, 90–93, 131–135
Microsoft Windows 7
  debugging on, 300–302
  debugging on Windows 7
    configuring project, 301
    configuring remote machine, 301
    deploy and debug project, 302
Microsoft Windows 8
  debugging using reference accelerators,
  debugging using, 101
  disabling TDR on, 297–298
  emulator as accelerator on, 22
  features in, 292–294
  Windows Display Driver Model support on, 300
Microsoft Windows high-resolution performance timer API, 129
Microsoft Windows XP display driver model (XPDM), 304
modifying simple into tiled algorithm
  tile barriers and synchronization, 74–76
  tiled matrix multiplication, 76–77
  using tile_ static memory, 69
  writing simple algorithms, 68–69
multicore programming, 20
multiplication
  array, 70
  matrix, 70–72, 75, 76–77

N

namespace
  concurrency:fast..math, 61
  concurrency::precise_math, 61
  graphics, 259
NBody case study, 21–38
  callback functions in, 30–34
  CPU calculations in, 29–34
  falling back to CPU, 221
  prerequisites for, 21–22

running on multiple GPUs, 211
running sample for, 22–24
structure of, 28–29
using graphics interop, 280–282
NBodyGravityAMP project
NBodyAmpTiled class in, 85–86
NBodyGravityCPU project, 85–86
in tiling NBody case study, 90
.NET Application, deployment from
, 306
norm and unorm, in graphics interop, 258, 263
NVIDIA
CUDA and, 13
Streaming Multiprocessors, 138
warps on, 4

O

occupancy
choosing tile size and size of, 80
guidelines for improving, 157–158
Open GL (Open Graphics Library), 13
OpenMP (OpenMultiprocessing), 10–11
operators, supported by short vector types, 259–260
optimizing
aliasing
performance impact of, 141
array of structs vs. struct of arrays, 38, 149–151
arrays of structs vs. structs of arrays, 149–151
computation
avoiding divergent code, 158–161
choosing the appropriate precision, 161–162
costing mathematical operations, 163
loop unrolling, 164–165, 195–198, 200
queuing modes, 168–169
using barriers for, 164
constant memory, 155–156
copying to and from GPU efficiently, 141–146
about, 141
leaving data on GPU, 144
removing unnecessary copies, 142–144
using overlapping asynchronous copies, 144–145
using staging arrays, 145–146
efficient accelerator global memory access, 146–148
occupancy and registers, 157–158
performance
about, 127–128
about preparing for, 169
analyzing performance, 128
measuring performance of kernel, 129–131
using concurrency visualizer, 131–135
using Concurrency Visualizer SDK, 137–138
texture memory, 156. *See also* textures
tile static memory access, 152–155. *See also* tile_static memory

P

padding tiles, 285–286
Parallel algorithm, 179
parallel-aware vs. parallel-unaware, software, 2
parallel_for algorithm, 12
parallel_for_each()
about, 55–56
aliasing invocations, 138–140
debugging tiled, 115
extent used in, 41, 54
lambdas used in, 41, 54
setting accelerator for running, 106
tiled, 67–68
tiled_extent in, 87
tile_static memory and, 89
using array_view as parameter, 56, 66–67
parallel_for_each algorithm, 12, 18
parallel_for function, 35
parallel_for loop, retractions for, 12
parallel_invoke algorithm, 12
parallelism
candidates for, 6
performance improvement through, 5–6
requirements for, 14
speeding up time windows and, 6–7
technologies for CPU, 8–13
ConcRT, 11–12
OpenMP, 10–11
vectorization, 8–10
WARP, 12–13
technologies for GPU, 13
Parallel Patterns Library (PPL)
C++ AMP and, 17
ConcRT and, 11–12
leveraging, to use all CPU cores, 90
using multiple CPU cores, 35
"Parallel Programming in Native Code" blog, 295

Parallel Programming with Microsoft Visual C++, website for book, 225
Parallel Stacks window, 115–116
Parallel Watch Window, 117–118, 121
"Patterns of Parallel Programming" (Mattson, Sanders, and Massingill), 219
performance case study, reduction, 175–176
  about the study, 171–172
  C++ AMP algorithms
    about, 179
    cascading reductions, 198–200
    eliminating bank conflicts, 193
    loop unrolling, 195–198
    minimizing divergence, 192–193
    naively tiled, 185–186
    reducing stalled threads, 194–195
    simple, 180–182
    simple optimized, 183–185
    simple with array_view, 179
    tiled with shared memory, 187–190
  constants and constraints in, 174–175
  CPU algorithms, 178–179
  overhead in, 178
  structure of, 172–174
performance improvements, history of, 1
performance optimization
  about, 127–128
  about preparing for, 169
  analyzing performance
    about, 128
  measuring performance of kernel, 129–131
  using concurrency visualizer, 131–135
  using Concurrency Visualizer SDK, 137–141
personal computing, history of, 1–2
pipeline, implementation of
  about features of Cartoonizer, 255
  Cartoonizer performance, 250
  Cartoonizing stage
    about, 236
    IFrameProcessor implementations, 239–245
  data structures in, 229–230
  edge detection algorithms used by Cartoonizer, 241
  ImagePipeline class, 232–236
  using multiple accelerators in, 246–251
    forked pipeline strategy, 249–252
    strategy for, 246–248
platforms for heterogeneous computing, 2–3
PLINQ, 12
pointers, references and, 58
portability of executables, 19–20
power requirements vs. battery life, in applications, 3
PPL (Parallel Patterns Library)
  C++ AMP and, 17
  ConcRT and, 11–12
  leveraging, to use all CPU cores, 90
  using multiple CPU cores, 35
precise_math namespace functions
  about, 161–162
  list of, 163
precision, choosing the appropriate, 161–162
programmable memory, 65
programming, multicore, 20

Q

queuing modes, 168–169

R

race conditions, 75, 121, 144, 165
race conditions, detecting in GPU debugging, 110–111
read-only resources, 142
real time control systems, as candidate for parallel processing, 6
reduction, about, 5
reduction performance case study
  about the study, 171–172
  C++ AMP algorithms
    about, 179
    cascading reductions, 198–200
    eliminating bank conflicts, 193
    loop unrolling, 195–198
    minimizing divergence, 192–193
    naively tiled, 185–186
    reducing stalled threads, 194–195
    simple, 180–182
    simple optimized, 183–185
    simple with array_view, 179
    tiled with shared memory, 187–190
  concurrency visualizer markers, 175–176
  constants and constraints in, 174–175
  CPU algorithms, 178–179
  overhead in, 178
  structure of, 172–174
REF accelerator (Reference Rasterizer), 205

reference accelerators, debugging using, 22, 101, 106–108
references, pointers and, 58
registers, 157
remote machine
  configuring for debugging, 301
Resource Manager, 11
resources, for C++AMP, 309–311
restrict(amp)
  functions marked with, 57–59
restrict keyword, 65
root and power functions, 61
run time
  initialization of, 128
  read-only resources and, 142
Run To Cursor command, debugging using, 123

S

scalar types
  texel, 262–264
  unorm and snorm scalar types, 258
  using HLSL, 258
scientific modeling and simulation, as candidate for parallel processing, 6
section() method, 144
sequential algorithm, 178
servers, running C++ AMP on
  enumerating C++ AMP-capable devices, 304–306
  running
    as a service or under Session 0, 305
    on true headless servers, 305
    without connected display, 305
    with XPDM graphics devices present, 304
shared pointers
  declaring global, 29
short vector types
  about, 259–260
  accessing vector components, 260
  operators supported by, 259–260
  template metaprogramming, 260–262
  texel, 262–264, 278–279
  using in Cartoonizer case study, 270
ShowAmpDevices sample, 22
Show Threads In Source button, 113
SIMD (Single Instruction, Multiple Data), 8
simple algorithms, 68–69
simple code vs. tiled code in tiling NBody case study, 87
single-core CPU algorithm, 35
"Software Adapter" accelerator, 205
Software Emulator
  in GPU debugging, 106–108
SSE (Streaming SIMD Extensions) 3, code for checking support of, 8–9
SSE (Streaming SIMD Extensions), readability of
  readability of, 35
staging arrays, using, 145–146
staging buffer, 145
Standard Library
  C++ AMP and, 17
  mathematical functions from, 61
  parameters in, 61
static memory, tiled, 65
Step Over command, visible glyphs in, 103
storage specifier, tile_static as, 76
Streaming Multiprocessors, 138
Streaming SIMD Extensions (SEE) 3, code for checking support of, 8–9
Streaming SIMD Extensions (SSE), readability of
  readability of, 35
structs
  in CPU calculations, 29–30
  in GPU programming, 38
  of arrays vs. arrays of structs, 38, 149–151
swapping data between accelerators, 211–214
synchronization, tile barriers and, 74–76
synchronize() method, 142
synchronizing data automatically, 60, 142

T

Task Graph pattern, 220
Task Parallel Library, 12
Task Scheduler, 11
TDR (Timeout Detection and Recovery)
  about, 296
  avoiding, 297
  detecting and recovering from, 298
  disabling on Windows 8, 297–298
template metaprogramming, 260–262
texels
  for data storage of textures, 262–264
  value types, 278–279
texture captured by reference, captured containers as, 138

textures
about, 262
as read-only, 142
copying data to and from, 264–266
Direct3D resource interop and, 277–280
maximum size of, 270
memory in, 155–156
reading from, 266–267
read-write, 269–270
texels
for data storage of, 262–264
value types, 278–279
using in Cartoonizer case study, 270
vs. arrays, 270–271
writeonly_texture_view, 269–270
writing to, 267–268
thawing threads, 121–122
Threading Building Blocks (TBB) 3.0, compatibility with PPL, 12
threads
arranged in groups, 4
CLR Thread Pool and, 12
freezing and thawing, 121–122
GPUs executing in kernel, 138
grouping. *See* tiling
handling truncated elements with edge, 287
reducing stalled, 194–195
seeing, 112–116
displaying GPU Threads Window, 113–114
filtering threads, 120–122
flagging threads, 119
grouping threads, 119
turning on thread markers, 113
using Parallel Stacks window, 115–116
using Parallel Watch Window, 117–118, 121
using Run To Cursor command, 123
splitting work between, using OpenMP, 10–11
taking more control of, 121–122
"tournament" approach in comparing, 5
Thrust, library of parallel algorithms, 13
tile barrier, 74–75, 89, 110, 114, 196–197
omitting in GPU debugging call to, 110–111
tiled calculation
performing, 64
tiled_extent, 66–67, 68, 69, 87
tiled_index, 67–68, 69
tiled matrix multiplication code, 76–77
tile() function, 87, 88
tile_origin, in tiled index, 67
tiles
padding, 285–286
tile size
choosing, 67, 69, 79, 95–98
effects of, 77–79
mismatches, dealing with, 283–284
tile_static memory
about, 65
access, 152–155
in reduction performance case study, 185
in tiled parallel_for_each, 87–88
using, 70–73
tile_static storage, 65, 76
tiling
about, 64–65
barriers and synchronization, 74–76
formula for tiled origin, 68
matrix multiplication, 291
modifying simple into tiled algorithm
tile barriers and synchronization, 74–76
tiled matrix multiplication, 70–72, 75, 76–77
using tile_ static memory, 69
writting simple algorithms, 68–69
NBody case study
about, 83
choosing tile size, 95–98
NBodyAmpTiled class in, in NBodyGravityAMP project, 85–86
simple code vs. tiled code in, 87
tiling n-body algorithms, 85–86
using concurrency visualizer, 90–93
tiles
padding, 285–286, 290
truncating, 286–288
timing in execution on GPU, 78
writing tiled algorithms, 68–69
tiling NBody case study
about, 83
choosing tile size, 95–98
NBodyAmpTiled class in, in NBodyGravityAMP project, 85–86
simple code vs. tiled code in, 87
tiling n-body algorithms, 85–86
using concurrency visualizer, 90–93
TimeFunc(), 176
Timeout Detection and Recovery (TDR)
about, 296
avoiding, 297
detecting and recovering from, 298
disabling on Windows 8, 297–298
time windows, speeding up, 6

timing
in execution on GPU, 78
tips and tricks, 302
atomic operations, 292–294
dealing with tile size mismatches, 283–284
debugging on Windows 7, 300–302
configuring project, 301
configuring remote machine, 301
deploy and debug project, 302
deployment
C++ AMP and Windows Store, 306
C++ CLR application, 307
C++ CLR project, 307
deploying applications, 303
library, 306
.NET Application, 306
running C++ AMP on servers, 304–306
using C++ AMP from managed code, 306
Windows 8 application, 306
double precision support
full, 300
limited, 300
features in Windows 8, 295
function objects vs. lambdas, 291–292
handling truncated elements
with edge treads, 287
with sections, 288–289
in debugging GPU, 108–109
initializing arrays, 290
tiles
padding, 285–286, 290
truncating, 286–288
Timeout Detection and Recovery, 296
detecting and recovering from, 298
disabling on Windows 8, 297–298
"tournament" approach, in comparing threads, 5
TransposeExample() function, 284
trigonometry functions, 61

U

UI code, in DirectX, 28
unary negation operator, supported by short vector types, 260

V

value types, texels, 278–279
VCRedist (Visual C++ 2012 Redistributable Package), 303
vectorization, 8–10
vector, synchronizing values in the array_view to, 51
views
Concurrency Visualizer, 90–93
creating accelerator, 48–49
Visual C++
PPL in, 11
precise and fast compiler flags, 163
support of OpenMP, 10
website for book on parallel programming with, 225
Visual C++ 2012 Redistributable Package (VCRedist), 303
visualizer, concurrency. *See also* Concurrency
Visualizer SDK
channels in window of, 134
Visual Studio 2012
auto-vectorization and auto-parallelization in, 9
C++ AMP accelerator and, 48
C++ AMP implemented in, 16
Concurrency Visualizer and, 90
debugging using, 101, 102, 104, 110
reference accelerator in, 22
supporting C++ AMP applications, 15
supporting vectorization in, 8
version needed for reduction performance case study, 175
Visual Studio Concurrency Visualizer
channels in window of, 134
examining memory access patterns, 149
markers in reduction performance case study, 175–176
using, 90–93, 131–135

W

warps, 79, 104–105, 123, 138, 147, 149–152, 157–159, 192–193, 196–198
about, 4
on NVIDIA hardware, 4
WARP (Windows Advanced Rasterization Platform), 22, 39, 198, 204–206, 218–221, 253, 295
about, 12–13
accelerator, 205, 218–219, 220
C++ AMP accelerator and, 48
tile size and, 79
wavefronts. *See* warp
wavefronts, on AMD hardware, 4

Windows 7
  debugging on, 300–302
  debugging on Windows 7
    configuring project, 301
    configuring remote machine, 301
    deploy and debug project, 302
Windows 8 Store
  application, deployment from, 306
  C++ AMP and, 306
  debugging using reference accelerators,
  debugging using, 101
  disabling TDR on, 297–298
  emulator as accelerator on, 22
  features in, 292–294
  Windows Display Driver Model support on, 300
Windows Device Driver Model (WDDM), 168
Windows Display Driver Model (WDDM) 1.1, 300
Windows high-resolution performance timer
API, 129
Windows XP display driver model (XPDM), 304
writeonly keyword
  substitute for, 60
writeonly_texture_view, 138, 269–270

About the Authors

ADE MILLER is currently a Principal Software Architect at Microsoft Studios. He has had several roles at Microsoft, including working on big data platforms as Program Manager with the Windows HPC Server team and managing the patterns & practices group's agile engineering teams as their Development Lead. His primary interests are parallel and distributed computing and improving the way teams deliver software through engineering leadership.

He is one of the authors of *Parallel Programming with Microsoft .NET* and *Parallel Programming with Microsoft Visual C++*. Ade also writes and speaks about parallel computing and his experiences with agile software development at Microsoft and elsewhere.

KATE GREGORY has been using C++ for over twenty years and is well-known as an instructor, speaker, and author. Managing, mentoring, technical writing, and technical speaking occupy much of her time, but she still writes code every week. Kate is the author of over a dozen books and speaks at DevTeach, TechEd (USA, Europe, Africa), and TechDays, among others. Kate is a C++ MVP, a founding sponsor of the Toronto .NET Users Group, the founder of the East of Toronto .NET Users group, and a member of adjunct faculty at Trent University in Peterborough. Since January 2002 she has been Microsoft Regional Director for Toronto and in January 2004 she was awarded the Microsoft Most Valuable Professional designation for Visual C++. In June 2005 she won the Regional Director of the year award and in February 2011 she was designated Visual C++ MVP of the year for 2010. Her firm, Gregory Consulting Limited, is based in rural Ontario and helps clients adopt new technologies and adjust to the changing business environment.