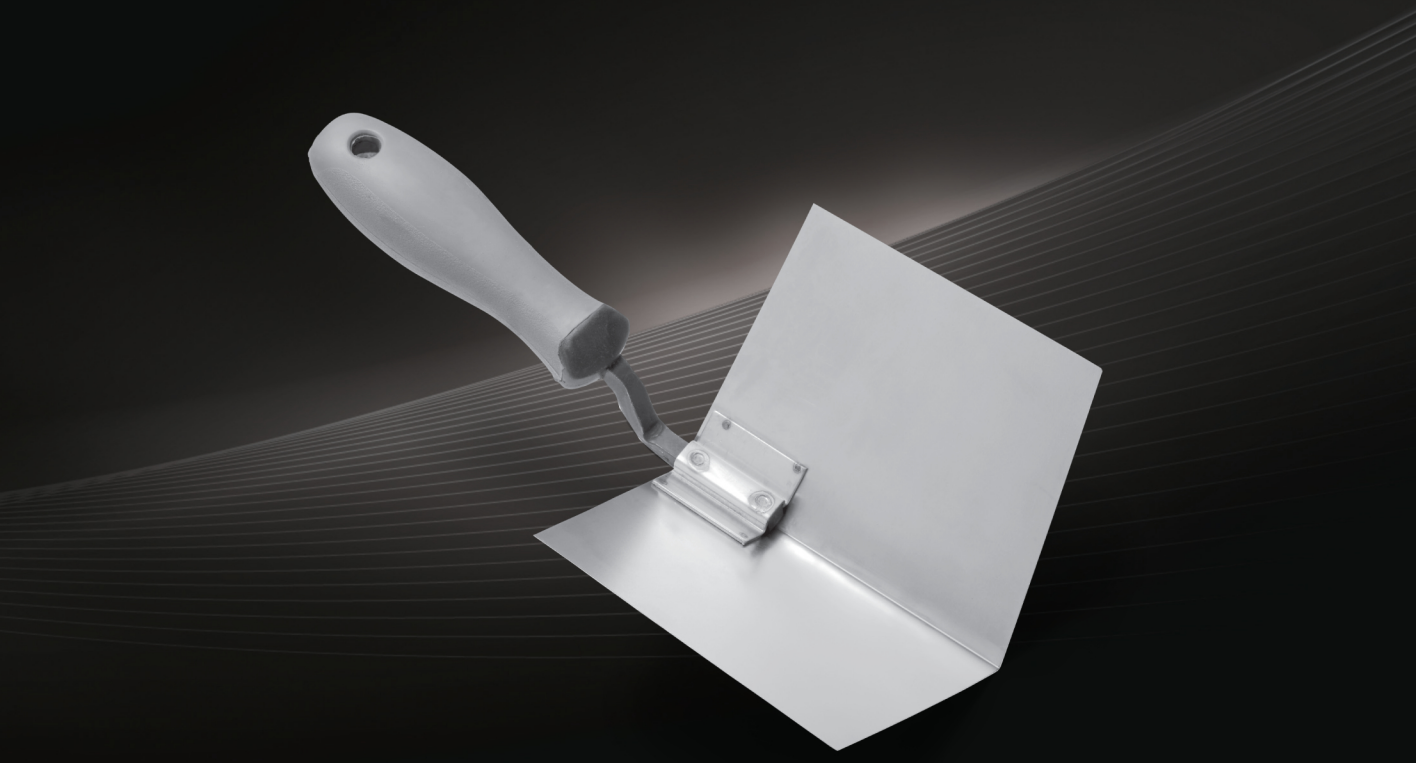
Microsoft

C++ AMP

Accelerated Massive Parallelism with Microsoft® Visual C++®



Kate Gregory
Ade Miller

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

Your guide to accelerating code performance with C++ AMP

Use your knowledge of C++ to take advantage of graphics processing units (GPUs) and other data-parallel hardware—and achieve maximum performance in your applications. Led by two expert programmers, you'll learn GPU programming fundamentals with C++ AMP and get best practices for exploiting this technology.

Discover how to:

- Create applications that run faster using C++ and Microsoft Visual Studio® 2012
- Produce the most dramatic acceleration by modifying your algorithm with tiling
- Debug your parallel code with Visual Studio
- Track the performance of your code with profiling tools
- Control how your application uses accelerators to get maximum performance
- Interoperate with the Microsoft DirectX® platform

Get code samples on the web
Ready to download at
http://go.microsoft.com/FWlink/?Linkid=260980

*For **system requirements**, see the Introduction.*

About the Authors

Kate Gregory is a Microsoft MVP for Visual C++ and a Microsoft Regional Director. She won the Regional Director of the Year award in 2005 and in 2010 was designated Visual C++ MVP of the Year. An enthusiastic instructor, speaker, and author, Kate has been using C++ for more than 20 years.

Ade Miller is a Principal Architect at Microsoft Studios. 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++*.

Microsoft® Most Valuable Professional

DEVELOPER ROADMAP

Start Here!
- Beginner-level instruction
- Easy to follow explanations and examples
- Exercises to build your first projects

Step by Step
- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook

Developer Reference
- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples

Focused Topics
- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples

Microsoft®

ISBN: 978-0-7356-6473-9

microsoft.com/mspress

U.S.A. $36.99
Canada $38.99
[*Recommended*]

Programming/C++

Microsoft®

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

Kate Gregory
Ade Miller

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

Contents

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://www.microsoftpressstore.com/title/9780735664739

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:

http://www.microsoftpressstore.com/title/ 9780735664739

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?

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.

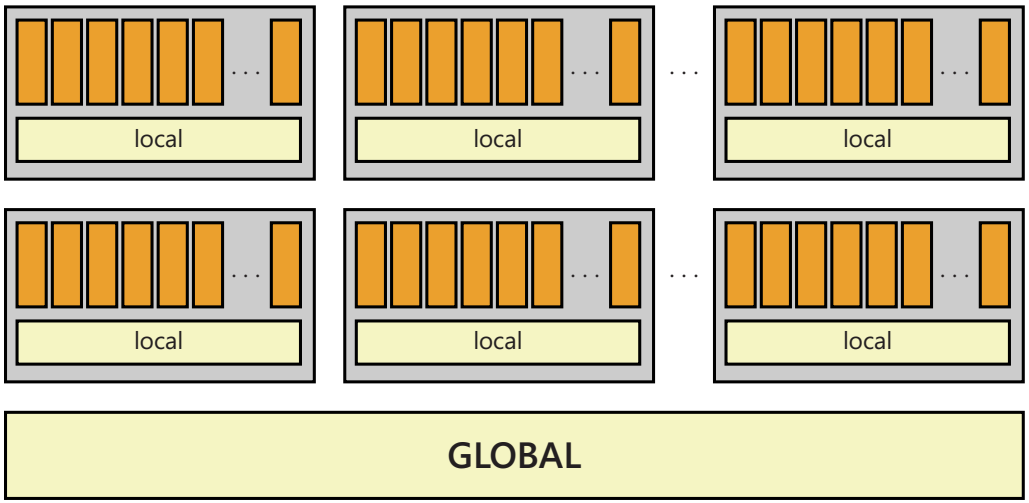
Then there is the matter of the memory accessed by these cores. Cache size can outweigh clock speed when it comes to compute speed, so the CPU has a large cache to make sure that there is always data ready to be processed, and the core will rarely have to wait while data is fetched. It's normal for CPU 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 9

Working with Multiple Accelerators

In this chapter:

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()*.

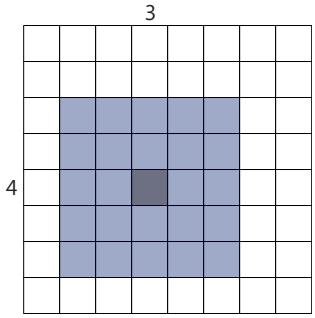
```
const int rows = 2000, cols = 2000; shift = 60;
std::vector<float> vA(rows * cols);
std::vector<float> vC(rows * cols);
std::iota(vA.begin(), vA.end(), 0.0f);

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();
```

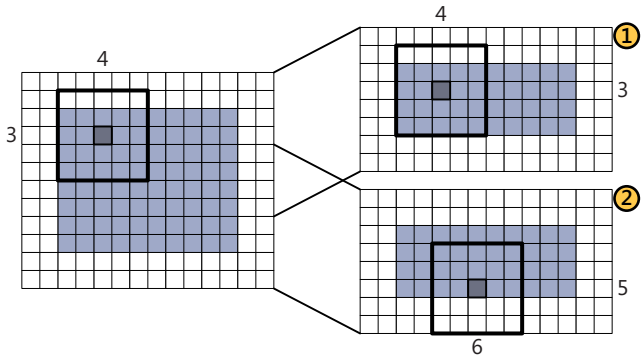
The *WeightedAverage()* function simply calculates an average using the weighted sum of the surrounding pixels, and the parameter *shift* specifies the size of the surrounding pixel window. This actual function isn't that important; for the purposes of the example, it is just work being done on the

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

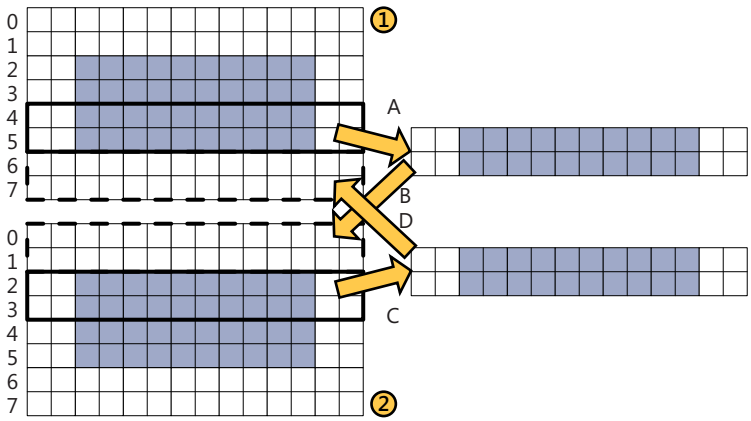
elements stored on other GPUs will need to refresh updated elements before the next iteration step can proceed.

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.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

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.