Chapter 12

The Final Countdown

Congratulations! We hope you've enjoyed learning about CUDA C and experimenting some with GPU computing. It's been a long trip, so let's take a moment to review where we started and how much ground we've covered. Starting with a background in C or C++ programming, we've learned how to use the CUDA runtime's angle bracket syntax to easily launch multiple copies of kernels across any number of multiprocessors. We expanded these concepts to use collections of threads *and* blocks, operating on arbitrarily large inputs. These more complex launches exploited interthread communication using the GPU's special, on-chip shared memory, and they employed dedicated synchronization primitives to ensure correct operation in an environment that supports (and encourages) thousands upon thousands of parallel threads.

Armed with basic concepts about parallel programming using CUDA C on NVIDIA's CUDA Architecture, we explored some of the more advanced concepts and APIs that NVIDIA provides. The GPU's dedicated graphics hardware proves useful for GPU computing, so we learned how to exploit texture memory to accelerate some common patterns of memory access. Because many users add GPU computing to their interactive graphics applications, we explored the interoperation of CUDA C kernels with industry-standard graphics APIs such as OpenGL and DirectX. Atomic operations on both global and shared memory allowed safe,

multithreaded access to common memory locations. Moving steadily into more and more advanced topics, streams enabled us to keep our entire system as busy as possible, allowing kernels to execute simultaneously with memory copies between the host and GPU. Finally, we looked at the ways in which we could allocate and use zero-copy memory to accelerate applications on integrated GPUs. Moreover, we learned to initialize multiple devices and allocate portable pinned memory in order to write CUDA C that fully utilizes increasingly common, multi-GPU environments.

12.1 Chapter Objectives

Through the course of this chapter, you will accomplish the following:

- You will learn about some of the tools available to aid your CUDA C development.
- You will learn about additional written and code resources to take your CUDA C development to the next level.

12.2 CUDA Tools

Through the course of this book, we have relied upon several components of the CUDA C software system. The applications we wrote made heavy use of the CUDA C compiler in order to convert our CUDA C kernels into code that could be executed on NVIDIA GPUs. We also used the CUDA runtime in order to perform much of the setup and dirty work behind launching kernels and communicating with the GPU. The CUDA runtime, in turn, uses the CUDA driver to talk directly to the hardware in your system. In addition to these components that we have already used at length, NVIDIA makes available a host of other software in order to ease the development of CUDA C applications. This section does not serve well as a user's manual to these products, but rather, it aims solely to inform you of the existence and utility of these packages.

12.2.1 CUDA TOOLKIT

You almost certainly already have the CUDA Toolkit collection of software on your development machine. We can be so sure of this because the set of CUDA C compiler tools comprises one of the principal components of this package. If

you don't have the CUDA Toolkit on your machine, then it's a veritable certainty that you haven't tried to write or compile any CUDA C code. We're on to you now, sucker! Actually, this is no big deal (but it does make us wonder why you've read this entire book). On the other hand, if you *have* been working through the examples in this book, then you should possess the libraries we're about to discuss.

12.2.2 CUFFT

The CUDA Toolkit comes with two very important utility libraries if you plan to pursue GPU computing in your own applications. First, NVIDIA provides a tuned Fast Fourier Transform library known as *CUFFT*. As of release 3.0, the CUFFT library supports a number of useful features, including the following:

- One-, two-, and three-dimensional transforms of both real-valued and complex-valued input data
- Batch execution for performing multiple one-dimensional transforms in parallel
- 2D and 3D transforms with sizes ranging from 2 to 16,384 in any dimension
- 1D transforms of inputs up to 8 million elements in size
- In-place and out-of-place transforms for both real-valued and complex-valued data

NVIDIA provides the CUFFT library free of charge with an accompanying license that allows for use in any application, regardless of whether it's for personal, academic, or professional development.

12.2.3 CUBLAS

In addition to a Fast Fourier Transform library, NVIDIA also provides a library of linear algebra routines that implements the well-known package of Basic Linear Algebra Subprograms (BLAS). This library, named *CUBLAS*, is also freely available and supports a large subset of the full BLAS package. This includes versions of each routine that accept both single- and double-precision inputs as well as real- and complex-valued data. Because BLAS was originally a FORTRAN-implemented library of linear algebra routines, NVIDIA attempts to maximize compatibility with the requirements and expectations of these implementations. Specifically, the CUBLAS library uses a column-major storage layout for arrays, rather than the row-major layout natively used by C and C++. In practice, this is

not typically a concern, but it does allow for current users of BLAS to adapt their applications to exploit the GPU-accelerated CUBLAS with minimal effort. NVIDIA also distributes FORTRAN bindings to CUBLAS in order to demonstrate how to link existing FORTRAN applications to CUDA libraries.

12.2.4 NVIDIA GPU COMPUTING SDK

Available separately from the NVIDIA drivers and CUDA Toolkit, the optional *GPU Computing SDK* download contains a package of dozens and dozens of sample GPU computing applications. We mentioned this SDK earlier in the book because its samples serve as an excellent complement to the material we've covered in the first 11 chapters. But if you haven't taken a look yet, NVIDIA has geared these samples toward varying levels of CUDA C competency as well as spreading them over a broad spectrum of subject material. The samples are roughly categorized into the following sections:

CUDA Basic Topics

CUDA Advanced Topics

CUDA Systems Integration

Data-Parallel Algorithms

Graphics Interoperability

Texture

Performance Strategies

Linear Algebra

Image/Video Processing

Computational Finance

Data Compression

Physically-Based Simulation

The examples work on any platform that CUDA C works on and can serve as excellent jumping-off points for your own applications. For readers who have considerable experience in some of these areas, we warn you against expecting to see state-of-the-art implementations of your favorite algorithms in the NVIDIA

GPU Computing SDK. These code samples should not be treated as production-worthy library code but rather as educational illustrations of functioning CUDA C programs, not unlike the examples in this book.

12.2.5 NVIDIA PERFORMANCE PRIMITIVES

In addition to the routines offered in the CUFFT and CUBLAS libraries, NVIDIA also maintains a library of functions for performing CUDA-accelerated data processing known as the NVIDIA Performance Primitives (NPP). Currently, NPP's initial set of functionality focuses specifically on imaging and video processing and is widely applicable for developers in these areas. NVIDIA intends for NPP to evolve over time to address a greater number of computing tasks in a wider range of domains. If you have an interest in high-performance imaging or video applications, you should make it a priority to look into NPP, available as a free download at www.nvidia.com/object/npp.html (or accessible from your favorite web search engine).

12.2.6 DEBUGGING CUDA C

We have heard from a variety of sources that, in rare instances, computer software does not work exactly as intended when first executed. Some code computes incorrect values, some fails to terminate execution, and some code even puts the computer into a state that only a flip of the power switch can remedy. Although having clearly *never* written code like this personally, the authors of this book recognize that some software engineers may desire resources to debug their CUDA C kernels. Fortunately, NVIDIA provides tools to make this painful process significantly less troublesome.

CUDA-GDB

A tool known as *CUDA-GDB* is one of the most useful CUDA downloads available to CUDA C programmers who develop their code on Linux-based systems. NVIDIA extended the open source GNU debugger (`gdb`) to transparently support debugging device code in real time while maintaining the familiar interface of `gdb`. Prior to CUDA-GDB, there existed no good way to debug device code outside of using the CPU to simulate the way in which it was expected to run. This method yielded extremely slow debugging, and in fact, it was frequently a very poor approximation of the exact GPU execution of the kernel. NVIDIA's CUDA-GDB enables programmers to debug their kernels directly on the GPU, affording them all of

the control that they've grown accustomed to with CPU debuggers. Some of the highlights of CUDA-GDB include the following:

- Viewing CUDA state, such as information regarding installed GPUs and their capabilities
- Setting breakpoints in CUDA C source code
- Inspecting GPU memory, including all global and shared memory
- Inspecting the blocks and threads currently resident on the GPU
- Single-stepping a warp of threads
- Breaking into currently running applications, including hung or deadlocked applications

Along with the debugger, NVIDIA provides the CUDA Memory Checker whose functionality can be accessed through CUDA-GDB or the stand-alone tool, `cuda-memcheck`. Because the CUDA Architecture includes a sophisticated memory management unit built directly into the hardware, all illegal memory accesses will be detected and prevented by the hardware. As a result of a memory violation, your program will cease functioning as expected, so you will certainly want visibility into these types of errors. When enabled, the CUDA Memory Checker will detect any global memory violations or misaligned global memory accesses that your kernel attempts to make, reporting them to you in a far more helpful and verbose manner than previously possible.

NVIDIA PARALLEL NSIGHT

Although CUDA-GDB is a mature and fantastic tool for debugging your CUDA C kernels on hardware in real time, NVIDIA recognizes that not every developer is over the moon about Linux. So, unless Windows users are hedging their bets by saving up to open their own pet stores, they need a way to debug their applications, too. Toward the end of 2009, NVIDIA introduced NVIDIA Parallel Nsight (originally code-named Nexus), the first integrated GPU/CPU debugger for Microsoft Visual Studio. Like CUDA-GDB, Parallel Nsight supports debugging CUDA applications with thousands of threads. Users can place breakpoints anywhere in their CUDA C source code, including breakpoints that trigger on writes to arbitrary memory locations. They can inspect GPU memory directly from the Visual Studio Memory window and check for out-of-bounds memory accesses. This tool has been made publicly available in a beta program as of press time, and the final version should be released shortly.

12.2.7 CUDA VISUAL PROFILER

We often tout the CUDA Architecture as a wonderful foundation for high-performance computing applications. Unfortunately, the reality is that after ferreting out all the bugs from your applications, even the most well-meaning "high-performance computing" applications are more accurately referred to as simply "computing" applications. We have often been in the position where we wonder, "Why in the Sam Hill is my code performing so poorly?" In situations like this, it helps to be able to execute the kernels in question under the watchful gaze of a profiling tool. NVIDIA provides just such a tool, available as a separate download on the CUDA Zone website. Figure 12.1 shows the Visual Profiler being used to compare two implementations of a matrix transpose operation. Despite not looking at a line of code, it becomes quite easy to determine that both memory and instruction throughput of the `transpose()` kernel outstrip that of the `transpose_naive()` kernel. (But then again, it would be unfair to expect much more from a function with *naive* in the name.)

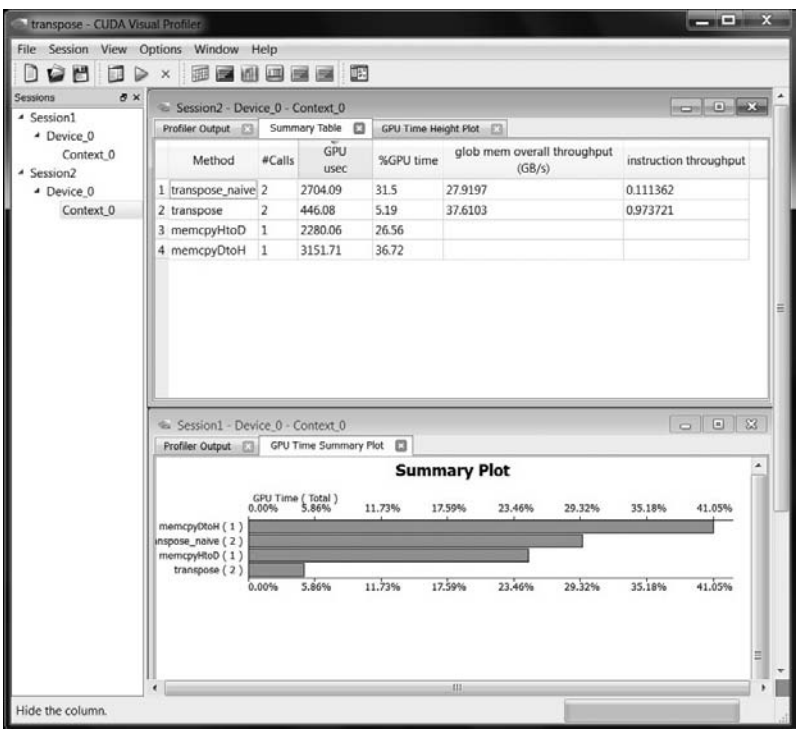

Figure 12.1 The CUDA Visual Profiler being used to profile a matrix transpose application

The CUDA Visual Profiler will execute your application, examining special performance counters built into the GPU. After execution, the profiler can compile data based on these counters and present you with reports based on what it observed. It can verify how long your application spends executing each kernel as well as determine the number of blocks launched, whether your kernel's memory accesses are coalesced, the number of divergent branches the warps in your code execute, and so on. We encourage you to look into the CUDA Visual Profiler if you have some subtle performance problems in need of resolution.

12.3 Written Resources

If you haven't already grown queasy from all the prose in this book, then it's possible you might actually be interested in reading more. We know that some of you are more likely to want to play with code in order to continue your learning, but for the rest of you, there are additional written resources to maintain your growth as a CUDA C coder.

12.3.1 PROGRAMMING MASSIVELY PARALLEL PROCESSORS: A HANDS-ON APPROACH

If you read Chapter 1, we assured you that this book was most decidedly *not* a textbook on parallel architectures. Sure, we bandied about terms such as *multiprocessor* and *warp*, but this book strives to teach the softer side of programming with CUDA C and its attendant APIs. We learned the CUDA C language within the programming model set forth in the *NVIDIA CUDA Programming Guide*, largely ignoring the way NVIDIA's hardware actually accomplishes the tasks we give it.

But to truly become an advanced, well-rounded CUDA C programmer, you will need a more intimate familiarity with the CUDA Architecture and some of the nuances of how NVIDIA GPUs work behind the scenes. To accomplish this, we recommend working your way through *Programming Massively Parallel Processors: A Hands-on Approach*. To write it, David Kirk, formerly NVIDIA's chief scientist, collaborated with Wen-mei W. Hwu, the W.J. Sanders III chairman in electrical and computer engineering at University of Illinois. You'll encounter a number of familiar terms and concepts, but you will learn about the gritty details of NVIDIA's CUDA Architecture, including thread scheduling and latency tolerance, memory bandwidth usage and efficiency, specifics on floating-point

handling, and much more. The book also addresses parallel programming in a more general sense than this book, so you will gain a better overall understanding of how to engineer parallel solutions to large, complex problems.

12.3.2 CUDA U

Some of us were unlucky enough to have attended university prior to the exciting world of GPU computing. For those who are fortunate enough to be attending college now or in the near future, about 300 universities across the world currently teach courses involving CUDA. But before you start a crash diet to fit back into your college gear, there's an alternative! On the CUDA Zone website, you will find a link for *CUDA U*, which is essentially an online university for CUDA education. Or you can navigate directly there with the URL www.nvidia.com/object/cuda_education.Although you will be able to learn quite a bit about GPU computing if you attend some of the online lectures at CUDA U, as of press time there are still no online fraternities for partying after class.

UNIVERSITY COURSE MATERIALS

Among the myriad sources of CUDA education, one of the highlights includes an entire course from the University of Illinois on programming in CUDA C. NVIDIA and the University of Illinois provide this content free of charge in the M4V video format for your iPod, iPhones, or compatible video players. We know what you're thinking: "Finally, a way to learn CUDA while I wait in line at the Department of Motor Vehicles!" You may also be wondering why we waited until the very end of this book to inform you of the existence of what is essentially a movie version of this book. We're sorry for holding out on you, but the movie is hardly ever as good as the book anyway, right? In addition to actual course materials from the University of Illinois and from the University of California Davis, you will also find materials from CUDA Training Podcasts and links to third-party training and consultancy services.

DR. DOBB'S

For more than 30 years, *Dr. Dobb's* has covered nearly every major development in computing technology, and NVIDIA's CUDA is no exception. As part of an ongoing series, *Dr. Dobb's* has published an extensive series of articles cutting a broad swath through the CUDA landscape. Entitled *CUDA, Supercomputing for the Masses*, the series starts with an introduction to GPU computing and progresses

quickly from a first kernel to other pieces of the CUDA programming model. The articles in *Dr. Dobb's* cover error handling, global memory performance, shared memory, the CUDA Visual Profiler, texture memory, CUDA-GDB, and the CUDPP library of data-parallel CUDA primitives, as well as many other topics. This series of articles is an excellent place to get additional information about some of the material we've attempted to convey in this book. Furthermore, you'll find practical information concerning some of the tools that we've only had time to glance over in this text, such as the profiling and debugging options available to you. The series of articles is linked from the CUDA Zone web page but is readily accessible through a web search for *Dr Dobbs CUDA*.

12.3.3 NVIDIA FORUMS

Even after digging around all of NVIDIA's documentation, you may find yourself with an unanswered or particularly intriguing question. Perhaps you're wondering whether anyone else has seen some funky behavior you're experiencing. Or maybe you're throwing a CUDA celebration party and wanted to assemble a group of like-minded individuals. For anything you're interested in asking, we strongly recommend the forums on NVIDIA's website. Located at http://forums.nvidia.com, the forums are a great place to ask questions of other CUDA users. In fact, after reading this book, you're in a position to potentially help others if you want! NVIDIA employees regularly prowl the forums, too, so the trickiest questions will prompt authoritative advice right from the source. We also love to get suggestions for new features and feedback on the good, bad, and ugly things that we at NVIDIA do.

12.4 Code Resources

Although the NVIDIA GPU Computing SDK is a treasure trove of how-to samples, it's not designed to be used for much more than pedagogy. If you're hunting for production-caliber, CUDA-powered libraries or source code, you'll need to look a bit further. Fortunately, there is a large community of CUDA developers who have produced top-notch solutions. A couple of these tools and libraries are presented here, but you are encouraged to search the Web for whatever solutions you need. And hey, maybe you'll contribute some of your own to the CUDA C community some day!

12.4.1 CUDA DATA PARALLEL PRIMITIVES LIBRARY

NVIDIA, with the help of researchers at the University of California Davis, has released a library known as the CUDA Data Parallel Primitives Library (CUDPP). CUDPP, as the name indicates, is a library of data-parallel algorithm primitives. Some of these primitives include parallel prefix-sum (*scan*), parallel sort, and parallel reduction. Primitives such as these form the foundation of a wide variety of data-parallel algorithms, including sorting, stream compaction, building data structures, and many others. If you're looking to write an even moderately complex algorithms, chances are good that either CUDPP already has what you need or it can get you significantly closer to where you want to be. Download it at http://code.google.com/p/cudpp.

12.4.2 CULATOOLS

As we mentioned in Section 12.1.3: CUBLAS, NVIDIA provides an implementation of the BLAS packaged along with the CUDA Toolkit download. For readers who need a broader solution for linear algebra, take a look at EM Photonics' CUDA implementation of the industry-standard Linear Algebra Package (LAPACK). Its LAPACK implementation is known as *CULAtools* and offers more complex linear algebra routines that are built on NVIDIA's CUBLAS technology. The freely available Basic package offers LU decomposition, QR factorization, linear system solver, and singular value decomposition, as well as least squares and constrained least squares solvers. You can obtain the Basic download at www.culatools.com/versions/basic.You will also notice that EM Photonics offers Premium and Commercial licenses, which contain a far greater fraction of the LAPACK routines, as well as licensing terms that will allow you to distribute your own commercial applications based on CULAtools.

12.4.3 LANGUAGE WRAPPERS

This book has primarily been concerned with C and C++, but clearly hundreds of projects exist that don't employ these languages. Fortunately, third parties have written wrappers to allow access to CUDA technology from languages not officially supported by NVIDIA. NVIDIA itself provides FORTRAN bindings for its CUBLAS library, but you can also find Java bindings for several of the CUDA libraries at www.jcuda.org. Likewise, Python wrappers to allow the use of CUDA C kernels from Python applications are available from the PyCUDA project at

http://mathema.tician.de/software/pycuda. Finally, there are bindings for the Microsoft .NET environment available from the CUDA.NET project at www.hoopoe-cloud.com/Solutions/CUDA.NET.

Although these projects are not officially supported by NVIDIA, they have been around for several versions of CUDA, are all freely available, and each has many successful customers. The moral of this story is, if your language of choice (or your boss's choice) is not C or C++, you should not rule out GPU computing until you've first looked to see whether the necessary bindings are available.

12.5 Chapter Review

And there you have it. Even after 11 chapters of CUDA C, there are still loads of resources to download, read, watch, and compile. This is a remarkably interesting time to be learning GPU computing, as the era of heterogeneous computing platforms matures. We hope that you have enjoyed learning about one of the most pervasive parallel programming environments in existence. Moreover, we hope that you leave this experience excited about the possibilities to develop new and exciting means for interacting with computers and for processing the ever-increasing amount of information available to your software. It's your ideas and the amazing technologies you develop that will push GPU computing to the next level.