

# CUDA

## Succinctly

by Chris Rose

# CUDA Succinctly

---

By

Chris Rose

Foreword by Daniel Jebaraj



Copyright © 2014 by Syncfusion, Inc.  
2501 Aerial Center Parkway  
Suite 200  
Morrisville, NC 27560  
USA  
All rights reserved.

## **I** mportant licensing information. Please read.

This book is available for free download from [www.syncfusion.com](http://www.syncfusion.com) on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from [www.syncfusion.com](http://www.syncfusion.com).

This book is licensed for reading only if obtained from [www.syncfusion.com](http://www.syncfusion.com).

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET  
ESSENTIALS are the registered trademarks of Syncfusion, Inc.

**Technical Reviewer:** Peter Shaw

**Copy Editors:** Suzanne Kattau and Courtney Wright

**Acquisitions Coordinator:** Jessica Rightmer, senior marketing strategist, Syncfusion, Inc.

**Proofreader:** Graham High, content producer, Syncfusion, Inc.

# Table of Contents

<b>The Story behind the <i>Succinctly</i> Series of Books .....</b>	<b>8</b>
<b>About the Author .....</b>	<b>10</b>
<b>Chapter 1 Introduction.....</b>	<b>11</b>
<b>Chapter 2 Creating a CUDA Project .....</b>	<b>12</b>
Downloading the Tools .....	12
Visual Studio 2012 Express .....	12
CUDA Toolkit.....	12
Device Query .....	13
Creating a CUDA Project.....	14
Text Highlighting.....	21
Timeout Detection and Recovery .....	22
<b>Chapter 3 Architecture .....</b>	<b>28</b>
Kernels, Launch Configurations, and dim3 .....	29
Kernels .....	29
Restrictions on Kernels .....	29
Launching a Kernel.....	30
Threads, Blocks, the Grid, and Warps .....	31
Threads .....	31
Thread Blocks.....	32
Grid.....	32
Warps .....	33
Device Memories.....	33
Registers .....	35

Shared Memory .....	36
Caches .....	36
Local Memory .....	37
Constant Memory .....	37
Texture Memory.....	39
Global Memory .....	40
Memories Summary .....	40
<b>Chapter 4 First Kernels.....</b>	<b>42</b>
Adding Two Integers .....	42
Function Qualifiers.....	43
CUDA API Memory Functions and cudaError_t .....	44
Copying Data to and from the GPU .....	45
Vector Addition Kernel .....	46
Recursive Device Functions .....	50
<b>Chapter 5 Porting from C++ .....</b>	<b>52</b>
C++ Host Version .....	52
CUDA Version .....	55
Tips and Tricks .....	60
<b>Chapter 6 Shared Memory.....</b>	<b>62</b>
Static and Dynamic Allocation .....	62
Statically Allocated Shared Memory.....	62
Dynamically Allocated Shared Memory .....	63
Using Dynamic Shared Memory as Multiple Data .....	64
CUDA Cache Config .....	64
Parallel Pitfalls and Race Conditions .....	65
Read-Modify-Write.....	67

Block-Wide Barrier .....	70
Atomic Instructions .....	71
Shared Memory Banks and Conflicts.....	72
Access Patterns.....	74
<b>Chapter 7 Blocking with Shared Memory .....</b>	<b>77</b>
Shared Memory Nearest Neighbor .....	77
<b>Chapter 8 NVIDIA Visual Profiler (NVVP).....</b>	<b>80</b>
Starting the Profiler.....	80
General Layout .....	81
Occupancy .....	83
Register Count.....	85
Shared Memory Usage .....	86
Unguided Analysis.....	86
Kernel Performance Limiter .....	87
Kernel Latency.....	88
Kernel Compute.....	89
Kernel Memory .....	90
Memory Access Pattern.....	91
Divergent Execution.....	93
Other Tools .....	93
Details Tab .....	93
Metrics and Events .....	94
<b>Chapter 9 Nsight .....</b>	<b>96</b>
Generating Profiling Reports .....	96
Debugging.....	98
CUDA Info Window.....	99

Warp Watch Window .....	100
<b>Chapter 10 CUDA Libraries .....</b>	<b>103</b>
Libraries: Brief Description .....	103
CUDA Random Number Generation Library (CURAND) .....	104
Host and Device APIs .....	104
Thrust.....	110
Basic Vector Manipulations.....	110
Sorting with Thrust.....	112
Transformations, Reductions, and Stream Compaction .....	114
<b>Conclusion .....</b>	<b>118</b>
<b>More Information.....</b>	<b>119</b>

# The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President  
Syncfusion, Inc.

**S**taying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

## Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

## The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

## The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.



## Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

## Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

## Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at [succinctly-series@syncfusion.com](mailto:succinctly-series@syncfusion.com).

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



## About the Author

Christopher Rose is an Australian software engineer. His background is mainly in data mining and charting software for medical research. He has also developed desktop and mobile apps and a series of programming videos for an educational channel on YouTube. He is a musician and can often be found accompanying silent films at the Majestic Theatre in Pomona, Queensland.

# Chapter 1 Introduction

CUDA stands for Compute Unified Device Architecture. It is a suite of technologies for programming NVIDIA graphics cards and computer hardware. CUDA C is an extension to C or C++; there are also extensions to other languages like FORTRAN, Python, and C#. CUDA is the official GPGPU architecture developed by NVIDIA. It is a mature architecture and has been actively developed since 2007. It is regularly updated and there is an abundance of documentation and libraries available.

GPGPU stands for general-purpose computing on graphics processing units. General purpose programming refers to any programming task that performs computations rather than standard graphics processing (CUDA is also excellent at graphics processing). Because graphics cards were originally intended to process graphics, there are very good reasons to want to harness their processing power for solving other problems. The most obvious reason is that they are extremely powerful processing units and they can take a lot of the workload off the CPU. The GPU often performs processing simultaneously with the CPU and is very efficient at certain types of computation—much more efficient than the CPU.

Parallel programming has become increasingly important in recent years and will continue to increase in importance in the coming years. The core clock speed of a CPU cannot increase indefinitely and we have almost reached the limit in this technology as it stands today. To increase the core clock speed of a CPU beyond the 3.5 GHz to 4.0 GHz range becomes increasingly expensive to power and keep cool. The alternative to increasing the clock speed of the processor is simply to include more than one processor in the same system. This alternative is exactly the idea behind graphics cards. They contain many hundreds (even thousands) of low-powered compute cores. Most graphics cards (at least the ones we will be programming) are called massively parallel devices. They work best when there are hundreds or thousands of active threads, as opposed to a CPU which is designed to execute perhaps four or five simultaneous threads. CUDA is all about harnessing the power of thousands of concurrent threads, splitting large problems up, and turning them inside out. It is about efficiently using graphics hardware instead of just leaving the GPU idle while the CPU struggles through problems with its handful of threads.

Studying CUDA gives us particular insight into how NVIDIA's hardware works. This is of great benefit to programmers who use these devices for graphics processing. The view of the hardware from the perspective of CUDA tends to be at a much lower level than that of a programmer who uses the GPUs to only produce graphics. CUDA gives us insight into the structure and workings of these devices outside the verbose and often convoluted syntax of modern graphics APIs.

This book is aimed at readers who wish to explore GPGPU with NVIDIA hardware using CUDA. This book is intended for folks with at least some background knowledge of C++ since all of the code examples will be using this language. I will be using the Visual Studio Express 2012 integrated development environment (IDE) but the examples should be easy to follow with the 2010 or 2013 versions of Visual Studio. [Chapter 9](#) focuses on Nsight which is only applicable to Visual Studio Professional editions, but the Express edition of Visual Studio will suffice for all the other chapters.

# Chapter 2 Creating a CUDA Project

## Downloading the Tools

### Visual Studio 2012 Express

Before starting any CUDA project, you need to ensure that you have a suitable IDE installed (this step can be skipped if you already have Visual Studio Express or Professional installed). Download and install Visual Studio 2012 Express. The code examples and screenshots in this book are all based on this IDE unless otherwise specified. The steps for creating a CUDA project for Visual Studio 2010 and 2013 are almost identical, and these IDEs should also be usable without any changes in my instructions. Visual Studio 2012 Express is available for download from the Microsoft website [here](#), and Visual Studio 2013 Express is available for download [here](#).

### CUDA Toolkit

Before downloading and installing the CUDA toolkit, you should make sure your hardware is CUDA-enabled. CUDA is specific to systems running with NVIDIA graphics hardware. AMD graphics cards, Intel, or any other graphics hardware will not execute CUDA programs. In addition, only relatively modern NVIDIA graphics cards are CUDA-enabled. The GeForce 8 series cards from 2006 were the first generation of CUDA-enabled hardware. You can check if your hardware is on the following list of CUDA-enabled devices on the NVIDIA developer website [here](#).

Once you are certain your hardware is CUDA-enabled, download and install the latest CUDA toolkit. You may like to register as an NVIDIA developer to receive news about the latest CUDA releases and CUDA-related events, and to get early access to upcoming CUDA releases. The toolkit is available from the NVIDIA developer website [here](#).

Be sure to download the version of the toolkit that is appropriate for the environment in which you will be developing. The CUDA toolkit requires a download of around 1 GB in size so the download may take some time. The version I will be using is the Windows Desktop version for 64-bits. We will not be programming any 64-bit specific code, so the 32-bit version would also be fine. The toolkit comes with many code samples, developer drivers for the graphics card, as well as the libraries required to code CUDA. Be sure to install Visual Studio before installing the CUDA toolkit as it adds the CUDA features to the IDE.

Run the downloaded file in a manner suitable for the platform on which you have downloaded it and be sure to follow any instructions given to install the toolkit correctly.

Once you install the CUDA toolkit, I encourage you explore the installed folder. We will refer to this install folder and its libraries and headers many times while programming CUDA. If you do not change the install path during installation, the default folder that the toolkit installs to is as follows:

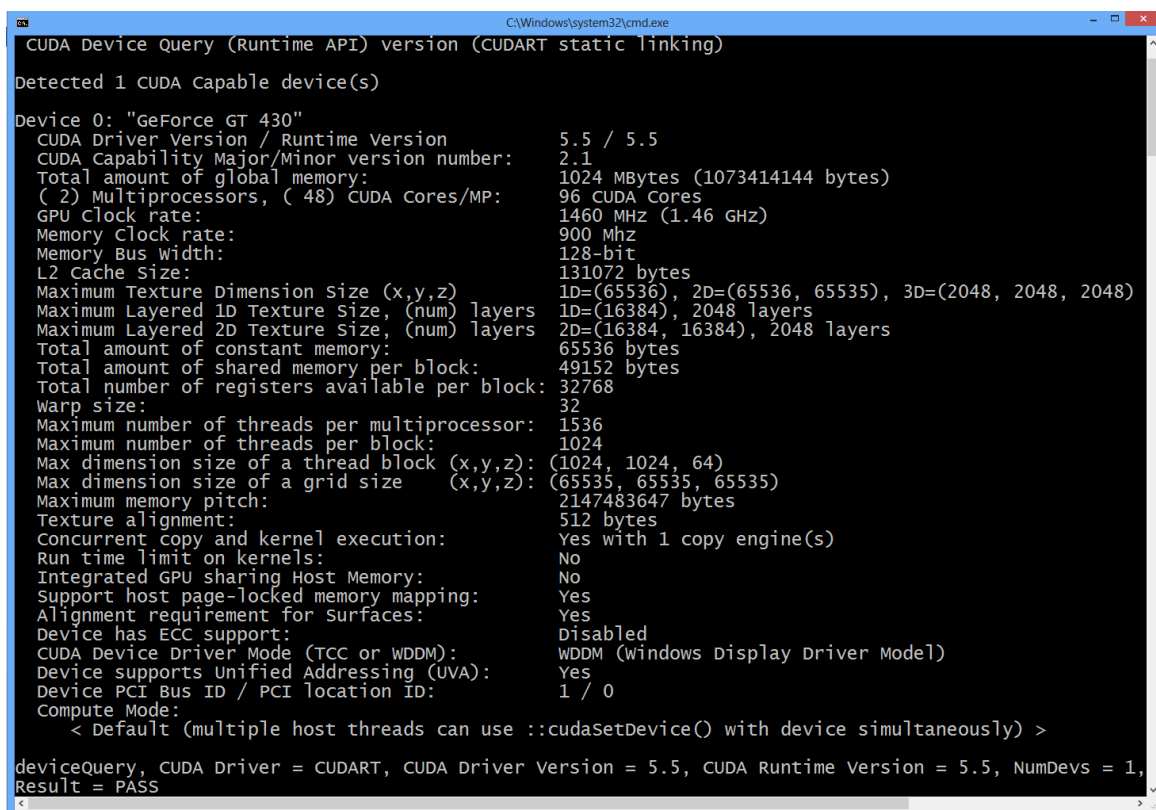
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v6.5

Where the **C:** is the main system drive and **v6.5** is the version of the toolkit you installed.

## Device Query

Before we begin a CUDA project, it is probably a good idea to get to know the hardware against which you will be programming. CUDA is low level; it is designed with very specific hardware in mind. Throughout the course of this book, many references are made to specific capabilities and metrics of the CUDA devices we will be programming. The CUDA toolkit installs a samples package containing a multitude of examples—one of which is called Device Query. This is a very handy application which prints interesting information about your installed hardware to a console window. Depending on the version of the CUDA toolkit you download, the DeviceQuery.exe program may be located in several different places. It is included as part of the CUDA samples, and you can find it by clicking the **Samples** icon in the NVIDIA program group and locating **DeviceQuery**.

The application can also be downloaded and installed separately from the NVIDIA website [here](#).



```
C:\Windows\system32\cmd.exe
CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 430"
  CUDA Driver Version / Runtime Version      5.5 / 5.5
  CUDA Capability Major/Minor version number: 2.1
  Total amount of global memory:             1024 MBytes (1073414144 bytes)
  ( 2) Multiprocessors, ( 48) CUDA Cores/MP: 96 CUDA Cores
  GPU Clock rate:                           1460 MHz (1.46 GHz)
  Memory Clock rate:                         900 Mhz
  Memory Bus Width:                          128-bit
  L2 cache size:                             131072 bytes
  Maximum Texture Dimension Size (x,y,z)     1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
  Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
  Total amount of constant memory:            65536 bytes
  Total amount of shared memory per block:    49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                 32
  Maximum number of threads per multiprocessor: 1536
  Maximum number of threads per block:        1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (65535, 65535, 65535)
  Maximum memory pitch:                      2147483647 bytes
  Texture alignment:                          512 bytes
  Concurrent copy and kernel execution:       Yes with 1 copy engine(s)
  Run time limit on kernels:                  No
  Integrated GPU sharing Host Memory:          No
  Support host page-locked memory mapping:    Yes
  Alignment requirement for Surfaces:         Yes
  Device has ECC support:                     Disabled
  CUDA Device Driver Mode (TCC or WDDM):       WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):    Yes
  Device PCI Bus ID / PCI location ID:        1 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 5.5, CUDA Runtime Version = 5.5, NumDevs = 1,
Result = PASS
```

Figure 2.1: Device Query Output



*Tip: You may find that when you run `deviceQuery.exe`, it opens and closes too quickly to read. Open the folder containing the `deviceQuery.exe` file in Windows Explorer. Hold down **Shift** and right-click a blank space in the folder. You should see an option to *Open command window here* in the context menu. Clicking this will open a command window in which you can type “`deviceQuery.exe`” to run the program, and the window will not close automatically.*

Figure 2.1 is the output from Device Query for the device I used throughout this book. References to various metrics such as the maximum number of threads per streaming multiprocessor (SM) can be found for your hardware by examining the output from Device Query. Where I refer to a particular value from Device Query in the text, you should look up the values for your own hardware. If you open Device Query as described previously, you might want to keep it open and minimized while working through this book for reference.

## Creating a CUDA Project

CUDA is initialized the first time a CUDA runtime function is called. The CUDA runtime is a collection of basic functions for memory management and other things. To call a CUDA runtime function, the project needs to include **CUDA.h** and link to the CUDA Runtime Library, **CUDART.lib**.

To create our first project, open a new solution in Visual Studio 2012 Express, choose **C++** as the language, and use an empty project. Once the project is created, right-click on the project's name in the Solution Explorer and click **Build Customizations** as per Figure 2.2. If you are using Visual Studio 2013, the Build Customizations option is under the submenu labeled **Build Dependencies**.

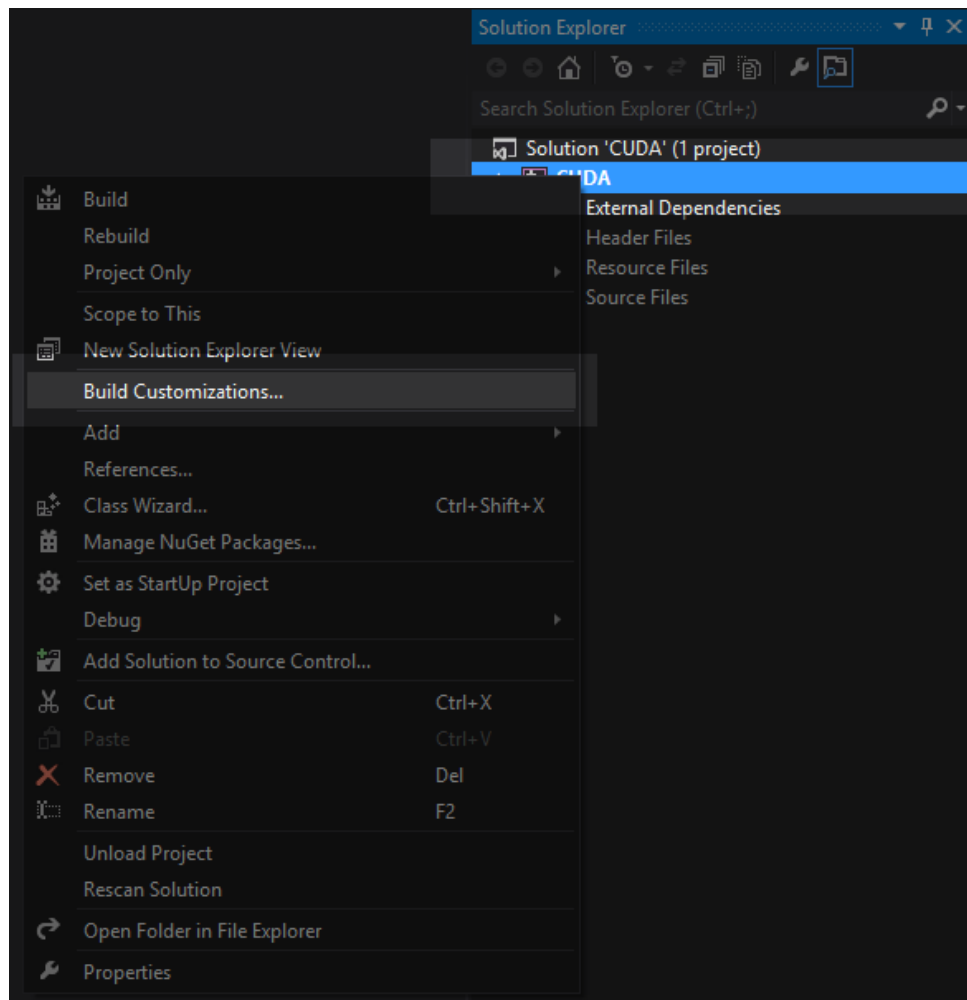


Figure 2.2: Build Customizations

This will open the Build Customizations window. Select the check box beside **CUDA x.x (.targets, .props)** as shown in Figure 2.3. Selecting this build customization will cause Visual Studio to use the CUDA toolset for files with a .cu extension, which are CUDA source files. CUDA kernels (functions designed for the GPU) are written in .cu files, but .cu files can contain regular C++ as well. Once you have selected the CUDA x.x(.targets, .props) build customization, click **OK** to save your changes.

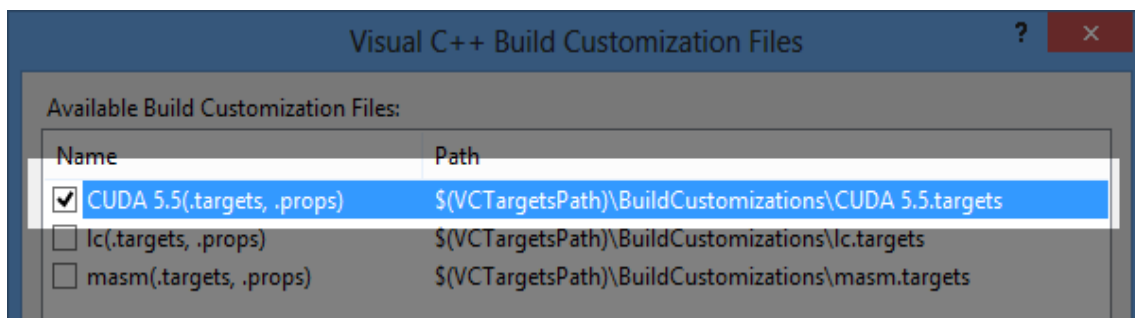
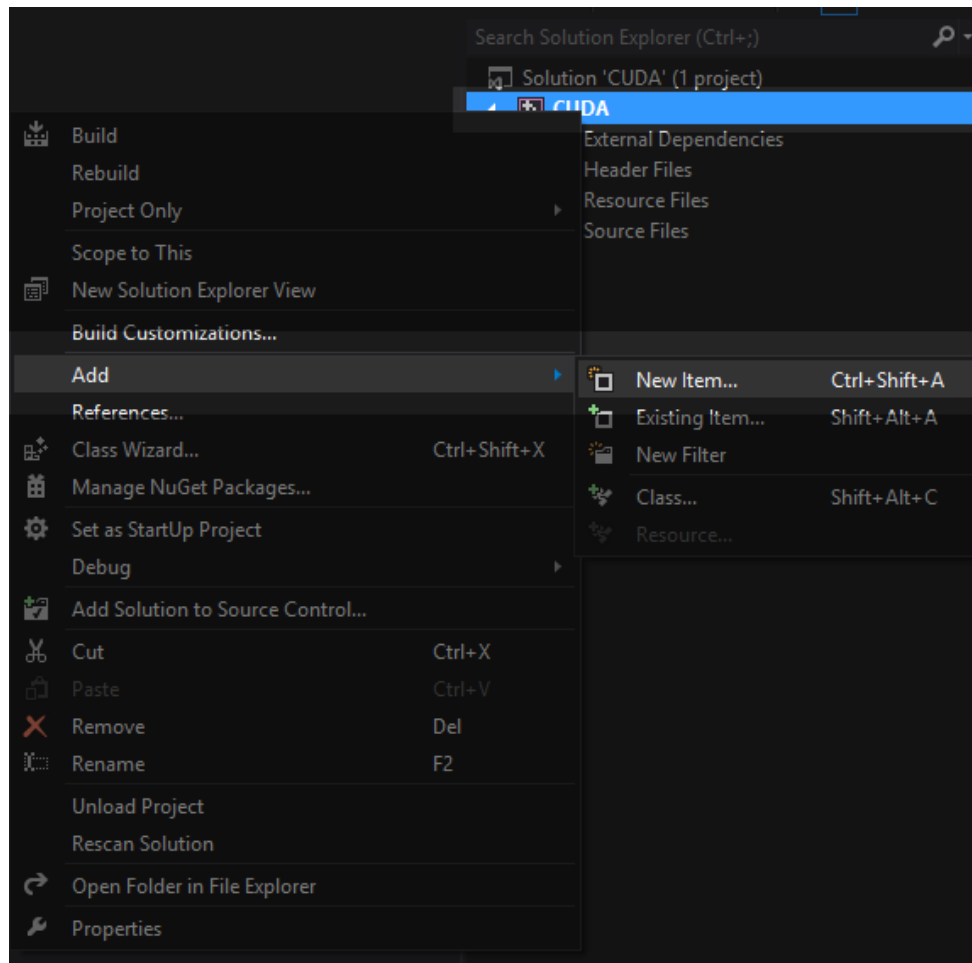


Figure 2.3: CUDA 5.5 (.targets, .props)

Next, we can add a CUDA source file to our project. Right-click on the project in the Solution Explorer, click **Add**, and then click **New Item** in the context menu as per Figure 2.4.



*Figure 2.4: Adding a Code File*

Click **C++ File (.cpp)** and name your file. I have called mine **Main.cu** as shown in Figure 2.5. The important thing is to give the file a .cu extension instead of the default .cpp. Because of the build customization setting, this code file will be sent to the NVIDIA CUDA C Compiler (NVCC). The NVCC compiler will compile all of the CUDA code and return the remaining C++ code to the Microsoft C++ compiler.



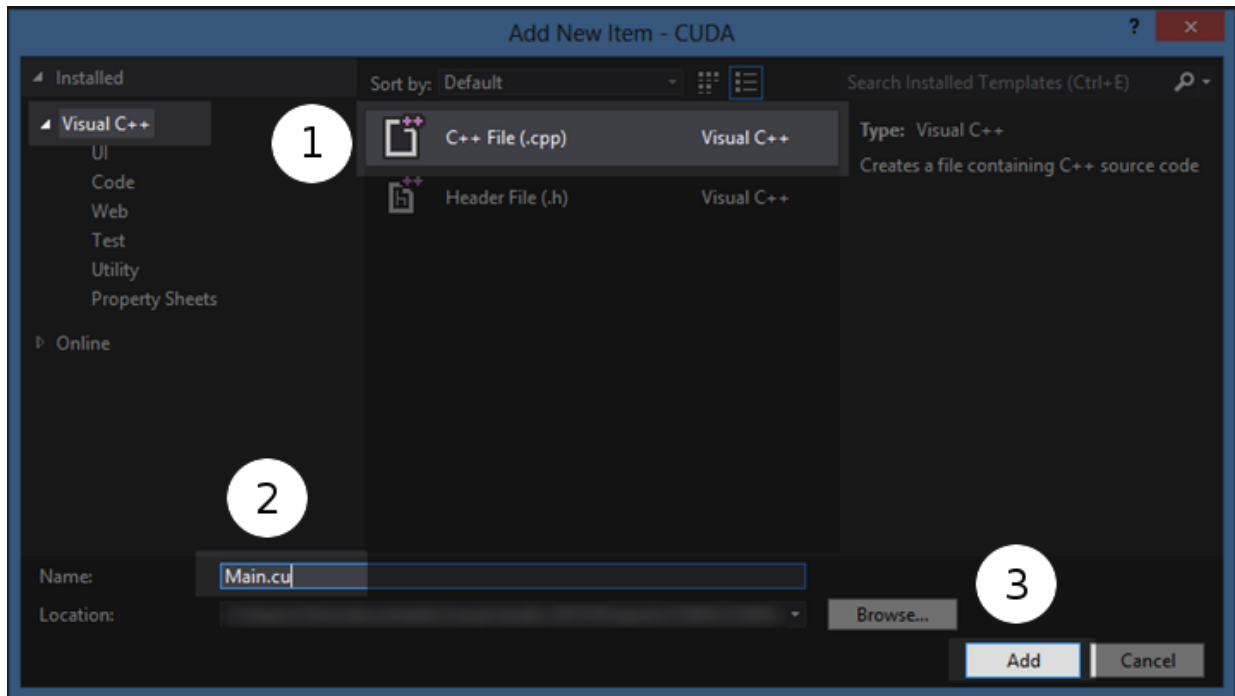


Figure 2.5: Adding a .cu File

Next, we need to ensure that Visual Studio is aware of the directory where the CUDA header files are located; this will likely be the same on your machine as it is on mine (depending on whether or not you changed the directory during installation). If you are using Visual Studio Professional, the paths may already be set up but it is always a good idea to check. Open the properties of the project by clicking **Project** in the menu bar and selecting **Project Name Properties** (where Project Name is the name of your project). Click **VC++ Directories** in the left panel and click the expand arrow beside the **Include Directories** entry in the right-hand panel. Click **Edit** in the menu that appears; this will allow you to specify paths which Visual Studio should search for headers, included with triangle braces (< and >). See Figure 2.6 for reference.

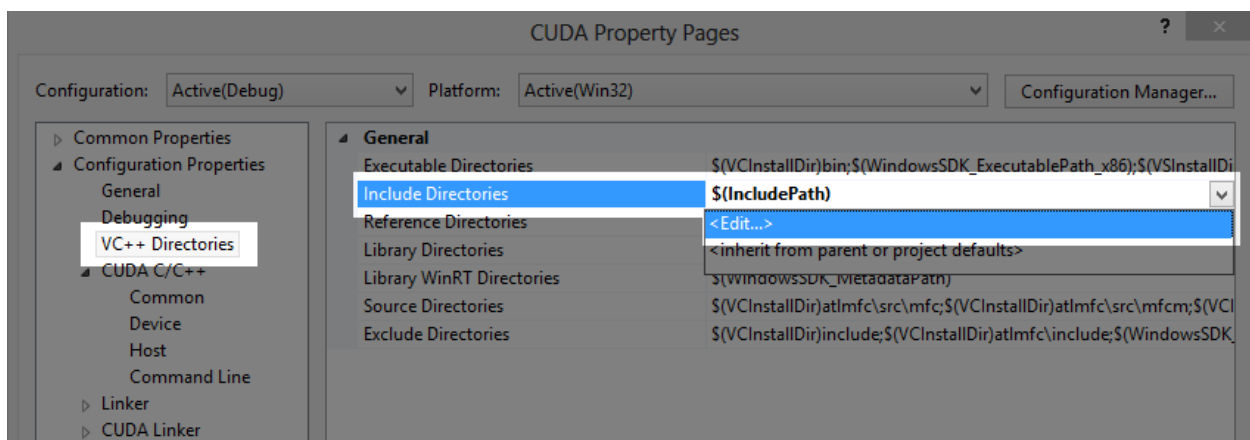


Figure 2.6: Properties for Including Directories

In the **Include Directories** dialog box, click the **New Folder** icon and locate the CUDA toolkit **include** folder on your computer. It will most likely be in a location similar to C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.5\include as shown in Figure 2.7. When you click **Select Folder**, you will be presented with the box in Figure 2.8. Click **OK** once you have selected your folder.

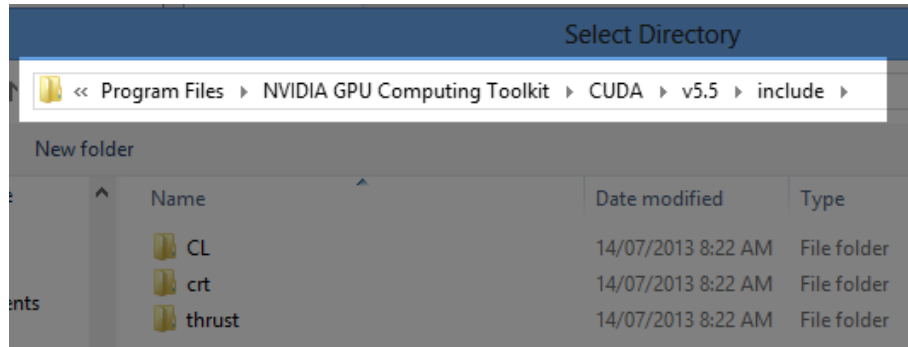


Figure 2.7: CUDA Toolkit include Directory

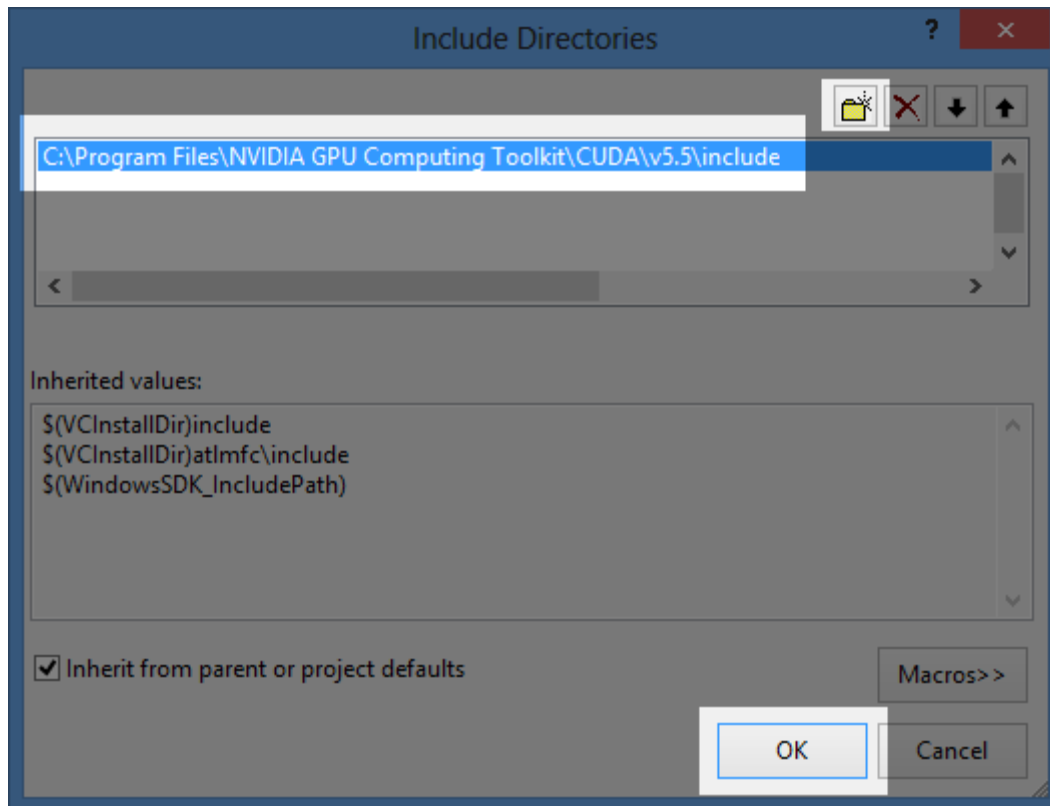


Figure 2.8: Click OK

Once this path is added, the CUDA **include** directory will be searched with the other standard include paths when Visual Studio locates headers included with the right-angle brackets (< and >).

Next, we need to specify where the CUDA library files are (that's the static link "lib" libraries). You may not need to specify where the CUDA library folder is located, depending on whether or not the CUDA installer automatically registered the folder where the CUDA libraries are installed. Unless you have specified a different directory when installing the CUDA toolkit, the library files will most likely be located in a place similar to the headers. Select **Library Directories**, click the drop-down arrow on the right, and click **Edit** as per Figure 2.9.

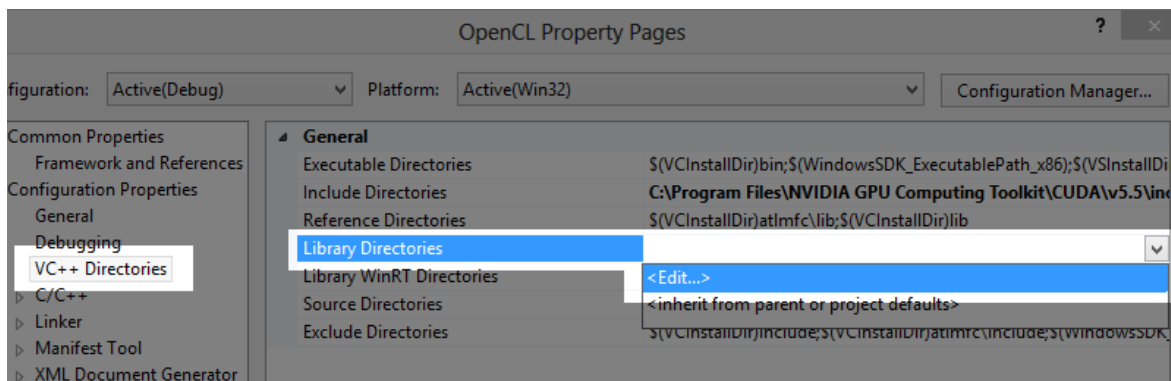


Figure 2.9: Properties for Library Directories

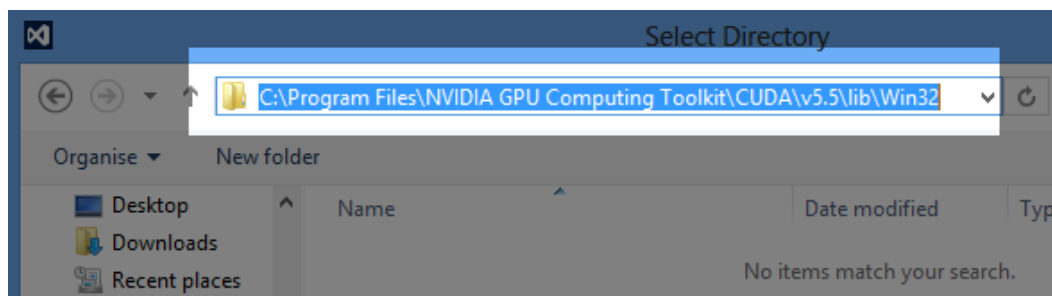


Figure 2.10: CUDA Library Directory

There are two library directories; one is for 32-bit projects and the other is for 64-bit projects. Select the one that is appropriate for your present solution platform. Remember, the CUDA version number and exact path may be different on your machine from the following examples.

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.5\lib\Win32

C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.5\lib\x64

Once the library directory is added, we need to link to the CUDA runtime library: a file called **CUDART.lib**. Expand the **Linker**, and then select **Input** from the left panel of the Project Properties page. Type **\$(CUDAToolkitLibDir)\CUDART.lib**; into the **Additional Dependencies** box. You can add the CUDART dependency to the beginning or to the end of the list but be careful not to delete all of the standard Windows library dependencies. Remember to click **Apply** to save these changes, and then click **OK** to close the box as per Figure 2.10. Later, when we link to other libraries, the steps are the same. For instance, to link to the **curand.lib** library, you would add **\$(CUDAToolkitLibDir)\CURAND.lib**; to the additional dependencies.



**Tip:** You can actually just supply the name of the library if the path is already known to Visual Studio. For example, you can type `CUDART.lib`; instead of the longer `$(CUDAToolkitLibDir)\CUDART.lib`.

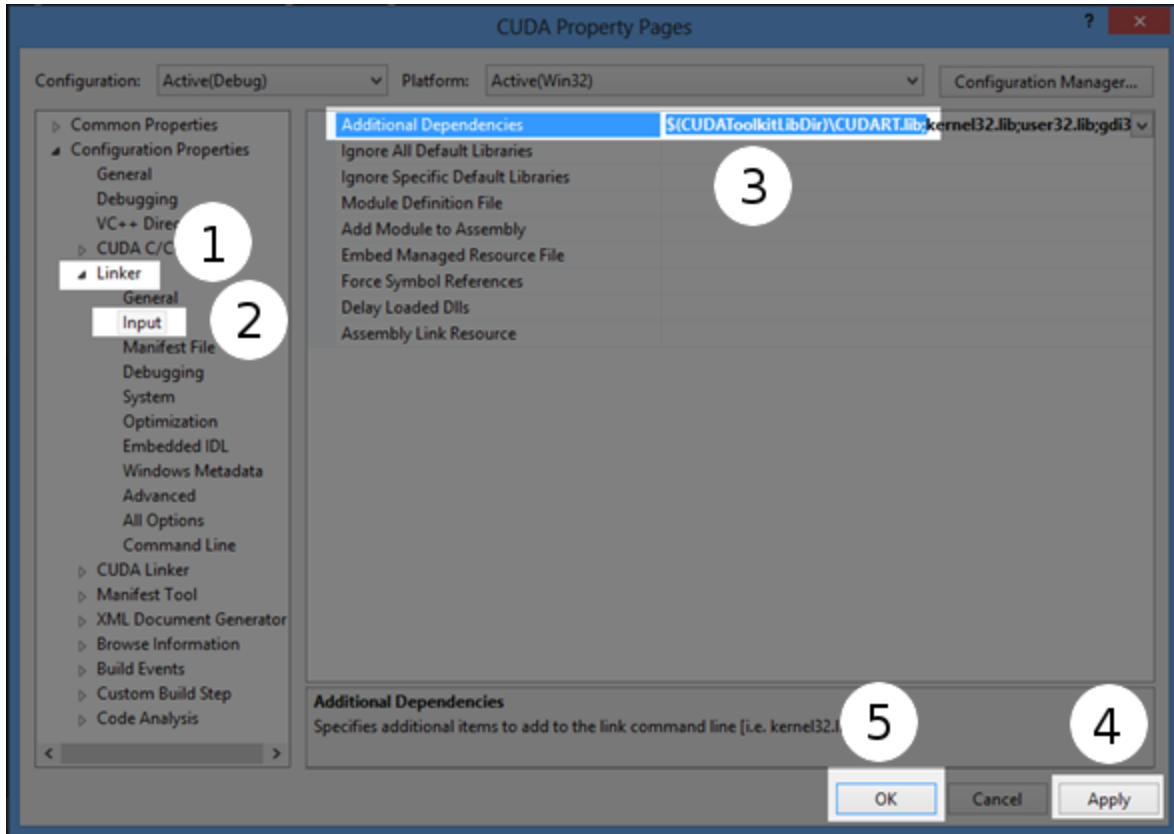


Figure 2.10: CUDA Runtime Library

Now that we have added the include path and the runtime library, we are ready to write some CUDA code. Initially, we can test that the Include directory and the link to CUDART.lib are working by writing the source code in Listing 2.1 into the .cu file we previously added to our project.

```
#include <iostream>
#include <cuda.h>           // Main CUDA header
using namespace std;
int main() {
    cout<<"Hello CUDA!"<<endl;
    return 0;
}
```

Listing 2.1: Basic CUDA Program

Debug your project by pressing F5 or clicking **Start Debugging** in the **Debug** menu. Listing 2.1 does not do anything with CUDA, but if the project builds correctly, it is a good indication that everything is installed properly. The program will print the text "Hello CUDA!" to the screen.



**Note:** Each version of CUDA is designed to work with one or more specific versions of Visual Studio. Your platform toolset (Visual Studio version) may not be supported by CUDA. You can change the platform toolset in the project's properties but only if you have other versions of Visual Studio installed. The platform toolset can be specified in the Configuration Properties | General page of the project properties. In the Platform option, you can see the versions of Visual Studio you have installed. To use other platform toolsets, you need to install different versions of Visual Studio. The code in this text was built with Visual Studio 2012 and the version of the platform toolset I used was v110.

## Text Highlighting

It is very handy to have Visual Studio recognize that the .cu files are actually C++ source code files. This way, we get all the benefits of code highlighting and IntelliSense while we program. By default, Visual Studio Express is not aware that the code in these files is basically just C++ with a few CUDA extensions. Open the Visual Studio properties pages by clicking **Tools** and then selecting **Options** from the menu.

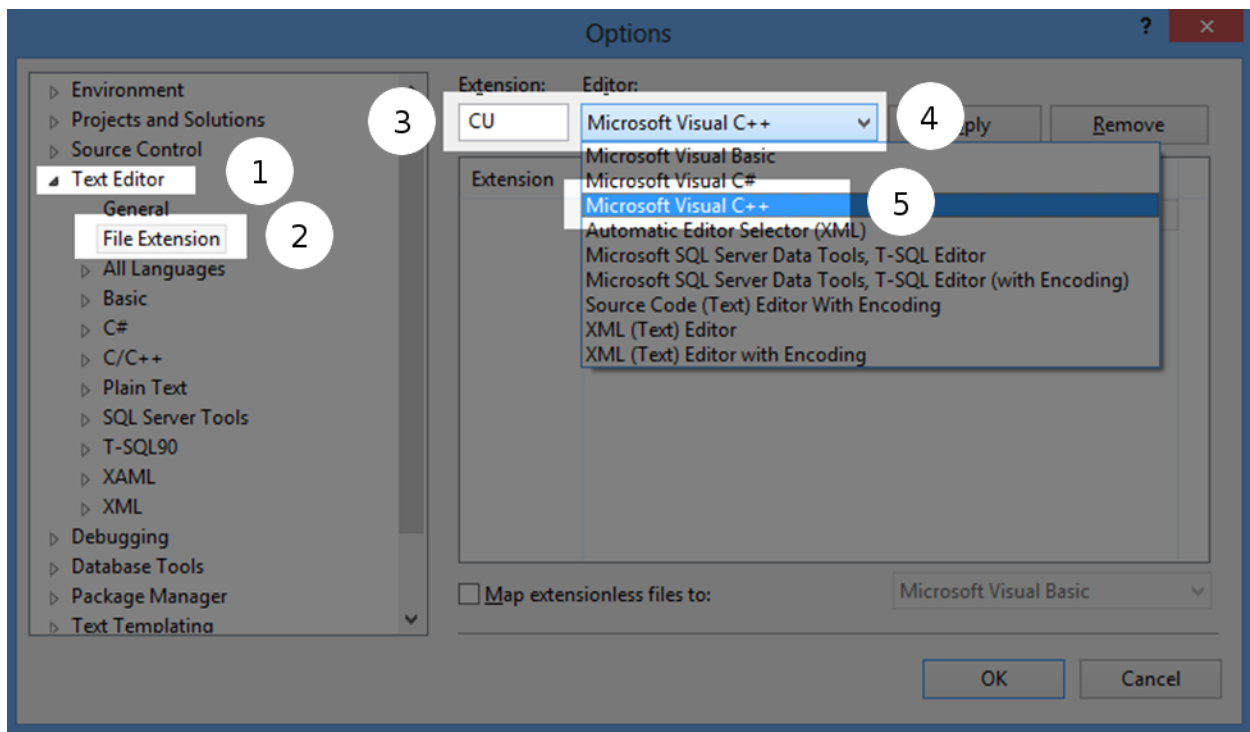


Figure 2.11: Syntax highlighting for .cu files

To enable C++ syntax highlighting and IntelliSense code suggestions for .cu files, click **Text Editor** and then **File Extension** in the panel on the left (see Figure 2.11). In the right panel, type “CU” into the **Extension** box. Select **Microsoft Visual C++** from the **Editor** drop-down list, and then click **Apply**. These changes will be saved for future projects. Whenever a .cu file is added to a project, Visual Studio will always use C++ colors and IntelliSense.

You may notice that most of the CUDA keywords are underlined in red as if they were errors. Visual Studio treats the .cu files as regular C++ but does not understand the CUDA keywords. The CUDA keywords are understood by NVCC without any declaration. There are headers you can include that define the keywords and help Visual Studio recognize them in the same manner as regular code. The only necessary header for general CUDA C programming is the CUDA.h header. The following headers shown in Listing 2.2 may also be included (all are located in the same folder as cuda.h) as well as the `#defines` shown; this will enable kernel highlighting and prevent Visual Studio from underlining CUDA keywords as errors.

```
#define __cplusplus
#define __CUDACC__

#include <iostream>
#include <cuda.h>
#include <device_launch_parameters.h>
#include <cuda_runtime.h>
#include <device_functions.h>
```

*Listing 2.2: Suggested Headers and Defines*



**Note:** Programs can be built with or without these additional headers but Visual Studio reports all undefined symbols as errors. When a program has legitimate errors and fails to compile, the CUDA symbol errors can number in the hundreds. This makes it very difficult to know which errors reported by Visual Studio are real errors and which are undefined CUDA symbols. For this reason, it is usually better to include the headers in Listing 2.2 (or whatever headers define the keywords in your code) even though they are optional.

## Timeout Detection and Recovery

Before we can compute intensive algorithms with the graphics card, we need to make sure the system will not assume the device has stalled. Usually during graphics processing, the device will compute a frame of graphics very quickly and respond to the operating system within 1/60 of a second (if it is running application at 60 frames per second). When using the device for general-purpose programming, we often expect computations to last for longer periods of time. A CUDA kernel may execute in a fraction of a second but, depending on the complexity of the computation, it might be expected to compute over a period of weeks.

The trouble is, when the device does not respond within two seconds, the operating system will assume the device has stalled while computing graphics. The operating system will reset the graphics driver, returning the user back to Windows or worse. At best, this will result in the user's screen turning off, the program crashing, and then the screen turning back on. But it can also result in the system resetting without safely shutting down any running applications.

We can change the amount of time the operating system allows the device by setting the **TdrLevel** and **TdrDelay** keys in the Windows Registry. The “Tdr” stands for timeout detection recovery. These registry keys are used to set the amount of time Windows will wait for the device to respond before it assumes it has stalled. They can also be used to turn off timeout detection altogether.

The program RegEdit.exe is used to alter the Windows Registry keys. To run RegEdit.exe, press and hold the Windows key and press R to open the **Run** dialog (the Windows key is in the lower left corner of your keyboard; it has an image of the Windows logo on it). Type “regedit” in the **Open** box as per Figure 2.12, and click **OK**. You may be prompted with the User Account Control box; if you are, click **Yes** to allow Registry Editor to make changes to the system. If you do not have administration rights to the system, you will need to contact your administrator before proceeding.

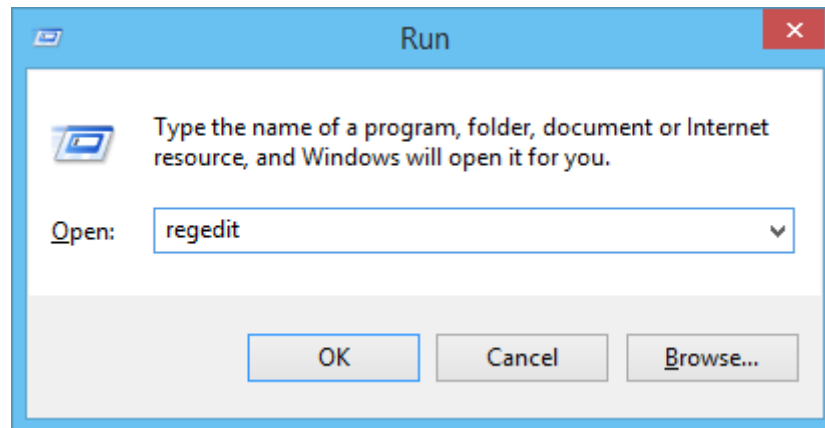
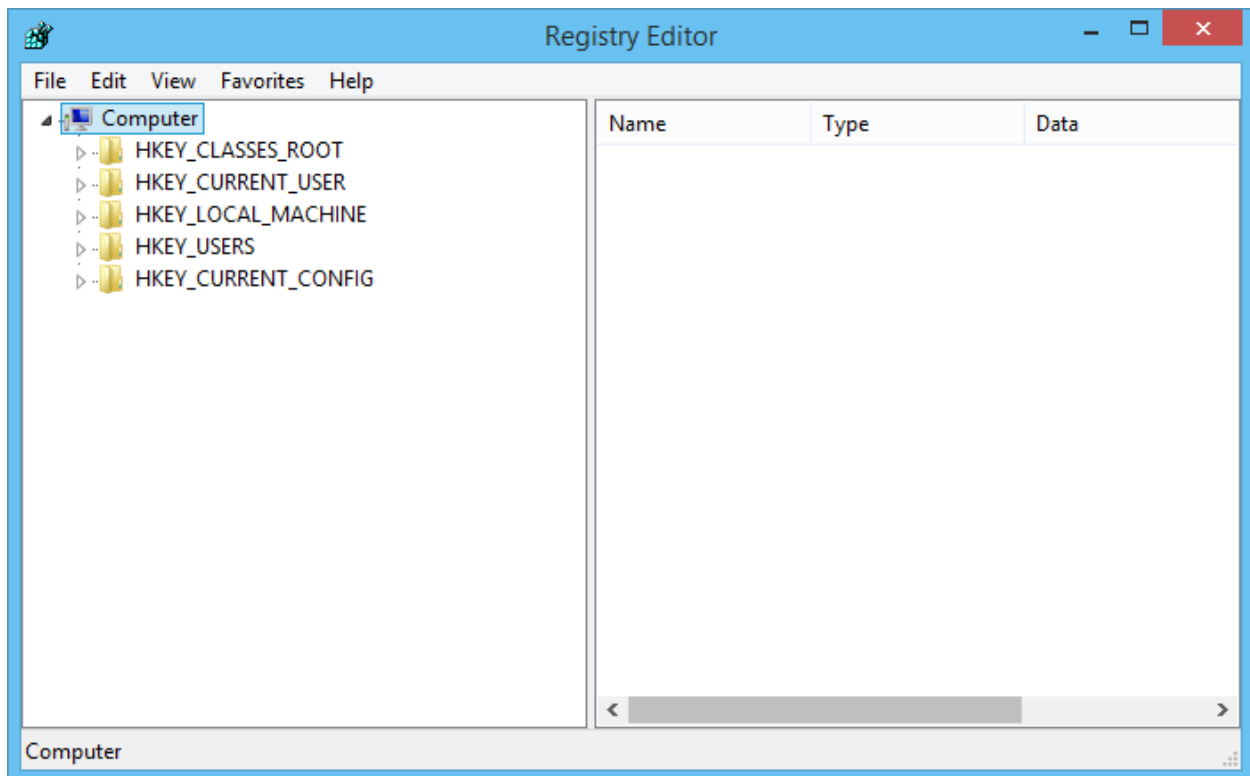


Figure 2.12: Windows Run Dialog

**Warning:** Be careful when using the RegEdit.exe program. The Windows Registry contains many keys that are important to the operation of the system. Save a backup of your registry before making changes by using the File > Export option or create a system restore point.



*Figure 2.13: Registry Editor*

When you open the Registry Editor application, you'll see a display similar to a regular file-browsing window as shown in Figure 2.13. We need to ensure two keys, **TdrLevel1** and **TdrDelay**, exist in the graphics driver's registry settings and have values suitable for our use.

The path to reach the graphics driver's keys can be found at

`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\GraphicsDrivers`

Click the **HKEY\_LOCAL\_MACHINE** folder in the left-hand panel, and then click the **System** sub-folder and continue in this manner until you find the **GraphicsDrivers** subfolder. Your screen should look similar to Figure 2.14.



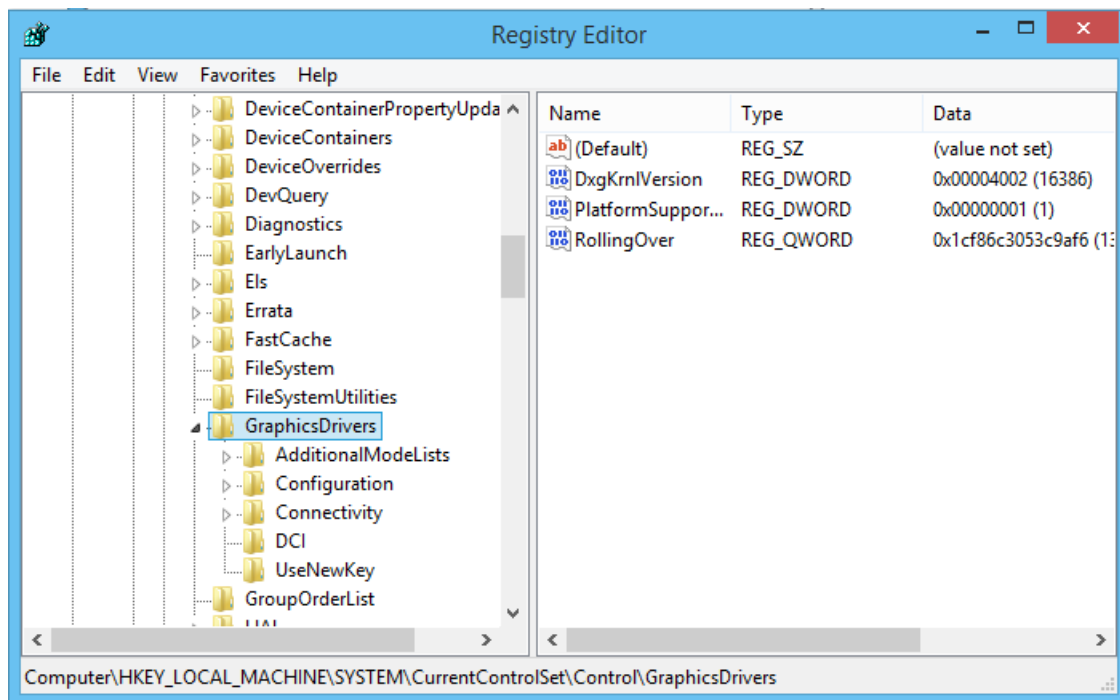


Figure 2.14: GraphicsDrivers Subfolder

You may have different keys in the GraphicsDrivers subfolder and you may already have the **TdrLevel1** and **TdrDelay** keys. If these keys already exist, you can skip the next step (where we will create them) and jump to Figure 2.17 where we modify the keys.

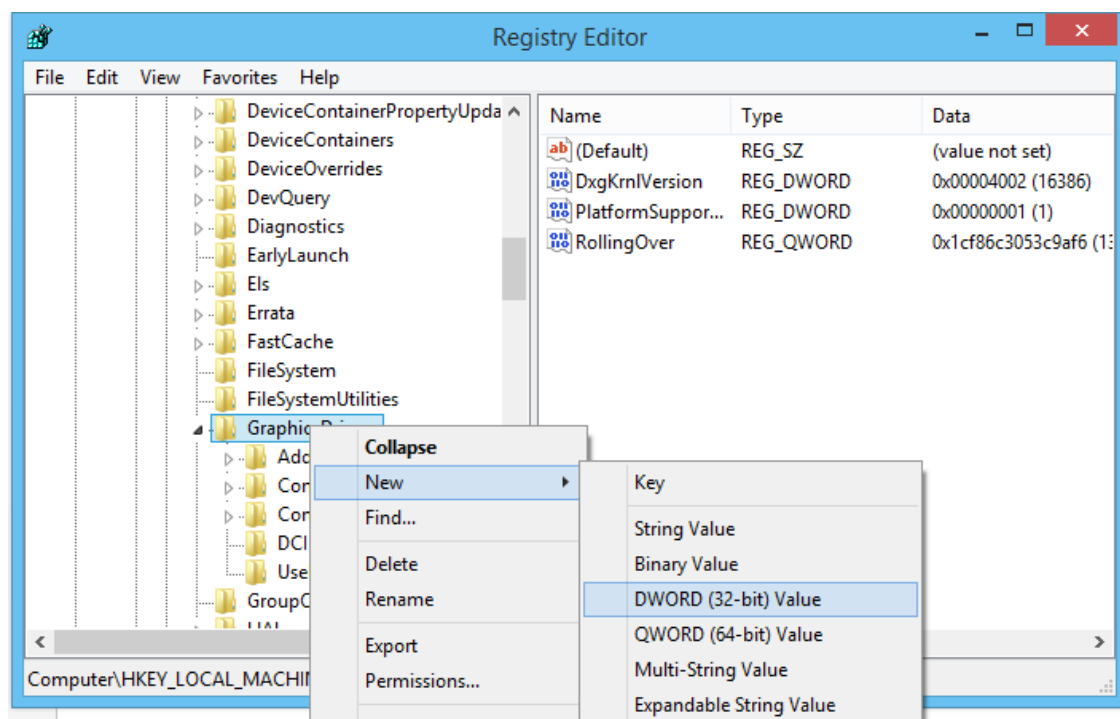


Figure 2.15: Adding a New Key

To add a new key to the registry, right-click the **GraphicsDrivers** folder in the left panel, click **New** and then select **DWORD (32-bit) Value** from the context menu. This will add a new key to the registry, which is a 32-bit integer. The key will initially be called New Value #1 or something similar. Rename the key **TdrLevel1** by right-clicking it and selecting **Rename** from the context menu as per Figure 2.16.

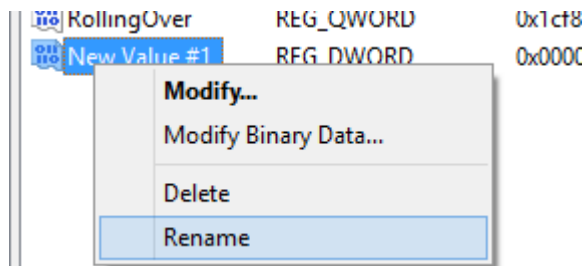


Figure 2.16: Renaming a Variable

Add another 32-bit integer key called **TdrDelay** using the same steps. Once the keys are added, you can edit their values by right-clicking their name and selecting **Modify** from the context menu as per Figure 2.17.

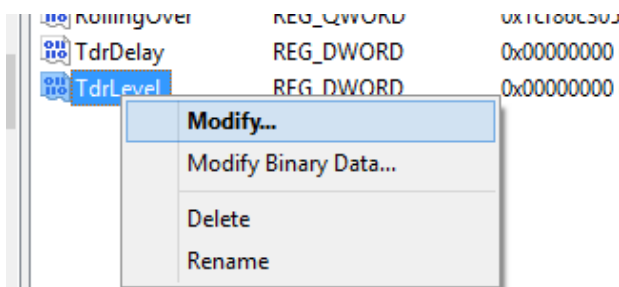


Figure 2.17: Modifying a Value

If the keys already exist, then all you need to do is modify the values. When you select **Modify** as shown in Figure 2.17, you will be presented with the Edit DWORD dialog (see Figure 2.18). You can then type a new value for the key in the **Value data** text box, selecting either hexadecimal or decimal as your base using the **Base** options.

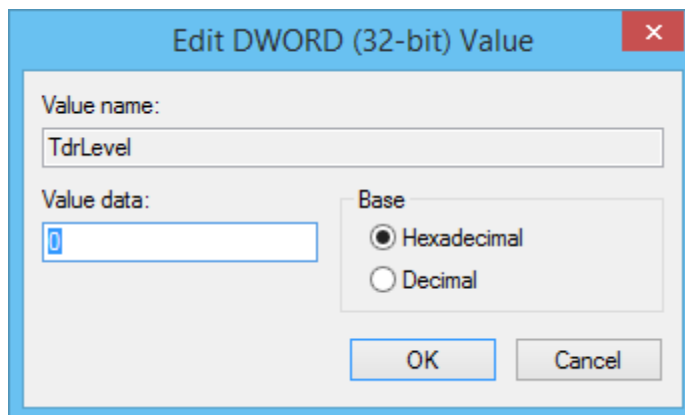




Figure 2.18: Editing a Value

For a complete description of all the settings for the TDR keys, visit the MSDN Dev Center at <http://msdn.microsoft.com/en-us/library/windows/hardware/ff569918%28v=vs.85%29.aspx>.

The **TdrLevel** key can be set to any of four values; we are only concerned with two of these:

- 0—Detection disabled. This is the most convenient for CUDA programming; it means that the system will never detect a timeout and reset the driver.
- 3—Recover on timeout. This is the default setting. The driver will reset if it takes longer to respond than the number of seconds specified in the **TdrDelay** key.

By default, the settings are 3 and 2 for **TdrLevel** and **TdrDelay**, respectively. This means that Windows will give the device two seconds and, if it does not respond, it will reset the driver. To allow more time (for instance, 60 seconds), set the **TdrLevel** variable to 3 and set the number of seconds (which is the value of the **TdrDelay** variable) to 60 (as per Figure 2.19).

 TdrDelay	REG_DWORD	0x0000003c (60)
 TdrLevel	REG_DWORD	0x00000003 (3)

*Figure 2.19: Example TDR Variable Settings*

Instead of allowing some amount of time before Windows should reset the driver, we can turn off timeout detection altogether by setting the **TdrLevel** key to 0. This means that no matter how long the device takes to respond, the driver will never be reset. This is the most convenient setting when you are CUDA programming—and it is the setting that I have used throughout this book.

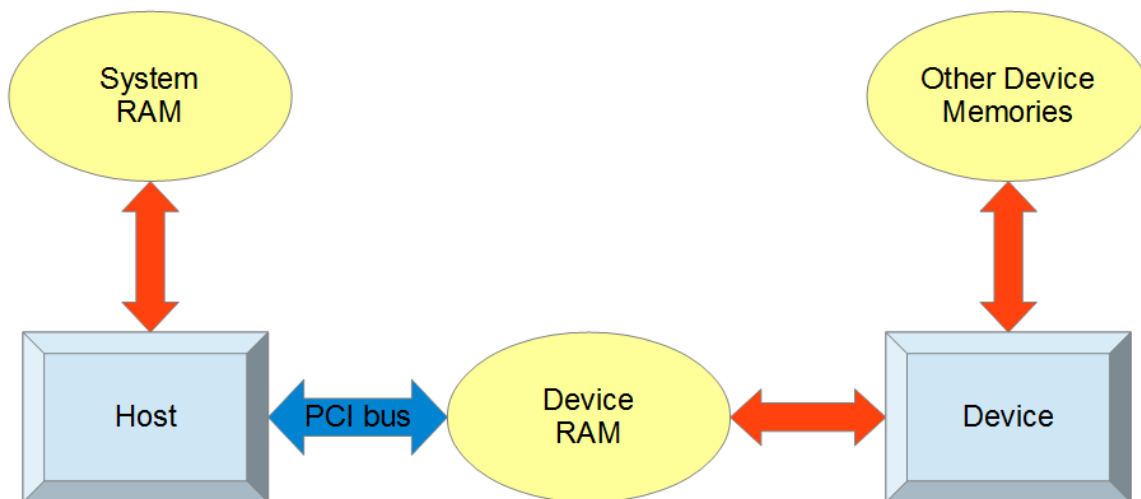
## Chapter 3 Architecture

We are about to run quickly through many terms which must be understood in order for CUDA to be appreciated. It is natural to be confused by this onslaught of terms. If this chapter is confusing, read it and the next chapter a few times, code the examples in Chapter 4, and examine the effect of changing the code. All of these concepts are connected and one cannot be described without knowing the others.

The GPU is called the device; the CPU and the rest of the system are called the host. The host is in control of the system memory and other peripheral hardware, including the device itself. The device has its own architecture and memory, often (but not always) separate from the system memory. The device is not able to control things such as hard drives, system RAM, or any other peripherals; this is all the domain of the host.

In computer systems where the device is on-board (i.e. included as part of the original build hardware and not added as an expansion card of some kind), it may share some of the main system memory. In these circumstances, the operating system and graphics driver may partition some of the system memory for specific use by the device as graphics RAM. This memory effectively becomes dedicated graphics RAM; it is programmed in exactly the same way as physical graphics RAM. For the remainder of this book, however, I will assume that this is not the case and that the device has its own dedicated memory.

The device communicates with the host through the PCI bus as shown in Figure 3.1.



*Figure 3.1: Device and Host*

All communication between the host and the device passes through the PCI bus. The host is able to read and write its own system RAM as well as the device RAM through special CUDA API function calls. The device can also read and write device RAM but it has an additional hierarchy of other memory spaces, which we will examine in detail.

The device RAM pictured in Figure 3.1 is often a special, fast type of DDR RAM (double data rate) called graphics DDR (GDDR). This main memory area of the device is often quite large (one or more gigabytes is not uncommon today). The amount of this device RAM is often cited as “dedicated graphics RAM” in the specifications of a GPU. As mentioned, in systems without dedicated graphics RAM, the device RAM will actually be a portion of system RAM and only separated conceptually. Usually, the device’s RAM chips sit on the device itself and the only communication channel to the host is the PCI bus socket on the motherboard the device is plugged into.

## Kernels, Launch Configurations, and dim3

### Kernels

CUDA kernels are functions that are designed to be executed by the device. They usually differ from regular host functions in that they are meant to be executed by many threads simultaneously. The host tells the device to run kernels by using something called a kernel launch. A kernel launch is similar to a regular C function call, only it has a launch configuration beside the function’s name. Listing 3.1 shows a kernel launch.

```
SomeKernel<<<12, 64>>>(a, b, c);
```

*Listing 3.1: Kernel Launch*

The launch configuration (Listing 3.1) is described with triple right-angle brackets (<<< and >>>); it configures the device’s execution units so the kernel’s code can be executed by many threads. Here, the kernel would be launched with 12 blocks—each containing 64 threads; **a**, **b** and **c** are the parameters. We will look at launch configurations in more detail momentarily.

### Restrictions on Kernels

Kernels have a syntax almost identical to regular C++ or C but there are some restrictions. Kernels cannot be recursive, they cannot have a variable number of arguments, and they must return void. They are marked with the `__global__` specifier and they are called by the host.

Not all device functions have to return void, only `__global__` kernels have this restriction. Helper functions, callable from the device and marked with `__device__`, need not return void. As of Fermi (Fermi is the name of the architecture of NVIDIA devices from the GeForce 400 series), `__device__` functions are allowed a limited depth of recursion as illustrated in Listing 4.6.

## Launching a Kernel

The launch configuration is specified after the kernel's name in a kernel call. The launch configuration can contain two or three parameters. The first two parameters specify the number of blocks in the grid and the number of threads in each block (we will describe threads and blocks shortly). The third parameter specifies the amount of dynamic shared memory.

```
Kernel_Name<<<gridSize, blockSize>>>(params...);
```

```
Kernel_Name<<<gridSize, blockSize, dynamic_Shared_Memory>>>(params...);
```

**Kernel\_Name** is the name of the kernel function being launched. The **gridSize** parameter is the number of blocks in the grid. And the **blockSize** parameter is the number of threads in each block. The **gridSize** and **blockSize** parameters can be either **dim3** (described next) or regular integers.

### dim3

The **dim3** data type is a structure composed of three unsigned integers: **x**, **y**, and **z**. It is used to specify the sizes of grids and blocks of multiple dimensions. It is also the data type of the **idx** indices (i.e., **threadIdx**, **blockIdx**, **blockDim**, and **gridDim**) which are used by threads to calculate unique IDs in kernels. In Listing 3.1, the launch configuration was specified using integers; in Listing 3.2, the launch configuration is specified using **dim3**.

```
dim3 gridSize(10, 10, 10);
dim3 blockSize(32, 32);

Call_Kernel<<<gridSize, blockSize>>>(params...);
```

*Listing 3.2: Multidimensional Kernel Launch*

Listing 3.2 illustrates the code for a multidimensional kernel launch using **dim3** structures for the block and grid size. The launch configuration uses two **dim3** structures to specify the arrangement of threads in the grid. The **gridSize** parameter specifies that the grid is to be a three-dimensional array of blocks, with dimensions **10**, **10**, and **10** (this means a total of 1,000 blocks will be launched). The **blockSize** parameter specifies that each block is to be composed of a two-dimensional grid of threads with dimensions **32** and **32**. This launch configuration will result in a total of 1,000 blocks with 32 × 32 threads in each block, for a total thread count of 1,024,000 threads.

The number of blocks and threads launched by the host can exceed the physical capabilities of the device. If your device is able to execute 768 threads per core and you have 16 cores, then the device can potentially execute 16 × 768 threads in parallel. But this number (16 × 768 = 12,288) is only the number of blocks and threads the device can physically execute in parallel; it is not a limit on the sizes specified by the launch configuration. The device has a scheduler which allocates blocks to physical cores for execution. When a core finishes executing a block, the scheduler will either give the core another block to execute or, if there are no more blocks to be executed in the launch, the kernel will complete.

The `dim3` data type supplies constructors for defining single-dimension, 2-D, and 3-D instances of the structure. The single-dimensional version (using a constructor with one parameter) initializes a `dim3` structure with only the X component set; the other dimensions default to `1`. The 2-D constructor (which takes two operands) can be used to set the X and Y dimensions of a `dim3` structure and the Z dimension is set to `1`. The 3-D constructor allows the programmer to set all three dimensions.

Multidimensional grids and blocks are a theoretical construction provided to simplify programming when data is inherently multidimensional. For instance, when working with a 2-D image, it might be convenient to launch 2-D blocks of threads and have each thread work with a single pixel.

## Threads, Blocks, the Grid, and Warps

The device is a physical piece of hardware, a dedicated graphics card which plugs into the motherboard of the computer. The device consists of some number of execution units called streaming multiprocessors (SM). An SM has a collection of thread processors, a register file, double precision unit, and shared memory. The SMs are capable of executing many threads simultaneously, with the thread processor units each executing a different thread.

### Threads

In parallel programming (and CUDA), a thread is a conceptual execution unit. It is a theoretical object which executes a single line of serial instructions. Parallel programming is entirely based on running concurrent threads. Instead of using a single thread to solve a problem, we can split problems into separate subproblems and use multiple threads at once to solve the larger problem. A single thread can be illustrated as an arrow with a squiggly tail as per Figure 3.2. Each thread in CUDA has an index called `threadIdx` which, along with the `blockIdx` and `blockDim`, can be read in the kernels and which is of type `dim3`.



Figure 3.2: A Thread



**Note:** Concurrency has a specific meaning in parallel programming. Concurrent threads may or may not actually execute in parallel (at exactly the same time). The order that threads execute is theoretically outside the programmer's control. Two threads are said to be concurrent if they could potentially access the same variable at the same time, or if it is impossible to tell based on the code which of the threads will access first.

## Thread Blocks

In CUDA, we group threads together into thread blocks. A thread block is a collection of threads which can easily communicate with each other. Each block has a block number (**blockIdx**) which is unique within the grid and which is a **dim3** type. Threads can use the **blockIdx** in kernels as a reference to which block the thread belongs. Threads can also reference the **blockDim** structure (also a **dim3**) which is a record of how many threads there are per block.

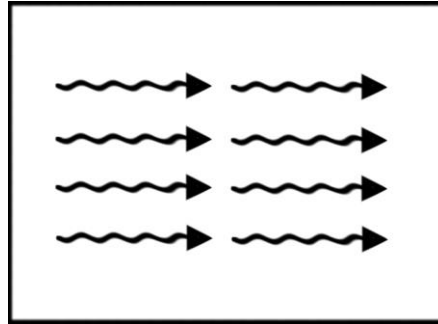


Figure 3.3: A Thread Block

## Grid

When we launch a kernel, we specify a collection of thread blocks called a grid. A thread block is a collection of threads and the grid is a collection of thread blocks. Figure 3.4 depicts a 2-D grid of thread blocks.

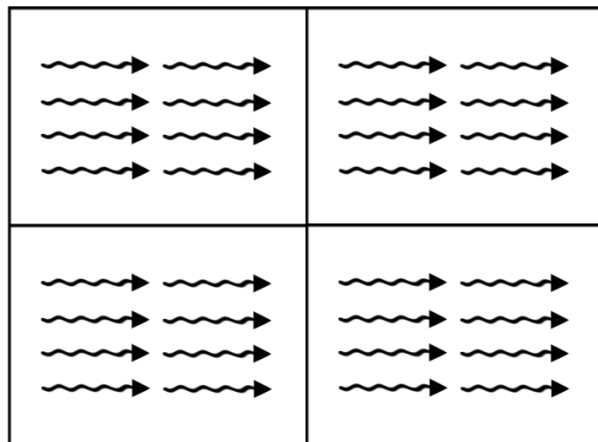


Figure 3.4: A grid of thread blocks

There is a maximum number of threads per block allowed (1,024 on Fermi, and in the previous example) but the maximum number of blocks per grid is very large. The total number of threads that runs concurrently is the size of the grid multiplied by the size of each block in threads. Threads can use the **gridDim** structure, which is a **dim3**, as a record of how many blocks the grid contained when the kernel was launched.



The extra layer of abstraction from threads to thread blocks means that, as new hardware becomes available, it can execute more blocks simultaneously and the code need not be changed. Threads within blocks have fast communication channels with each other but they cannot easily communicate with threads of other blocks.

## Warps

When the device executes a kernel, it splits threads into groups of 32, called warps. Each warp contains 32 threads with consecutive `threadIdx` values. All threads of a warp are from the same block.

The threads of a warp execute instructions in lockstep; this means they all execute the same instruction at the same time unless there is a branch. If the threads are in a warp branch (branching occurs when there is a condition such as an `if` statement and some threads take one path while others take another path), the device executes the two paths for the branch in serial. Managing the branching of threads within warps is an important consideration for performance programming.

Warps are also an important consideration for resource management. As mentioned, warps always consist of 32 threads, with consecutive `threadIdx` values from the same thread block. Every warp allocates enough resources for all 32 threads whether or not there are 32 threads in the block. This means, if the number of threads per block is less than 32, then the device will allocate resources for a complete warp of 32 threads and some of the resources will be wasted.

Even if a thread block contains more than 32 threads, or if it consists of any number of threads not evenly divisible by 32, then the block will waste resources in one of the warps. It is recommended that thread blocks have a number of threads which is evenly divisible by 32.

## Device Memories

There are many different types of memory in the GPU, each having a specific use along with various limitations and performance characteristics. Figure 3.5 illustrates the theoretical relationship between these memories. The two boxes marked “Block 0” and “Block 1” represent thread blocks; there are usually many more than two of these running concurrently. Each block has some amount of shared memory and a collection of threads. The threads each have access to their own local memory and registers.

In addition to the memory spaces which blocks and threads access individually (i.e., the shared memory, local memory, and registers), there are several memory spaces that all threads can access. All threads and the CPU host have access to the global memory, the constant memory, and the texture memory.

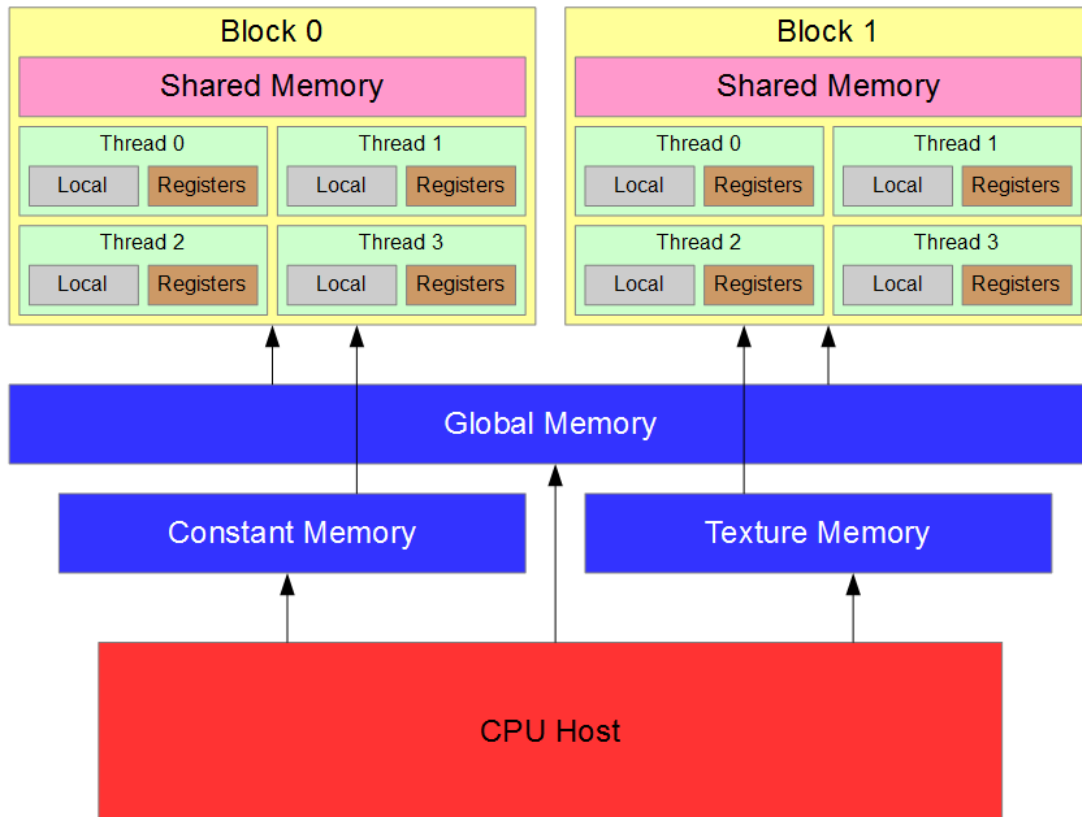


Figure 3.5: Memory Hierarchy

Figure 3.6 shows a typical graphics card. A graphics card is a combination of hardware including the GPU itself. It is common to refer to an entire graphics card as a GPU, but the GPU is actually only the main chip on a graphics card, like the chip marked *B* in Figure 3.6. This particular graphics card was designed by Gainward and features an NVIDIA GT 440. The heat sink (which is a large fan for cooling the device) has been removed to better view the layout.

Examining the image in Figure 3.6, you will notice four horizontal and four vertical black rectangles marked with an *A*. These rectangles are the device's main graphics memory chips. The device's main memory is used for global, local, constant, and texture memory (all of which will be explained shortly). Each of these memories also has associated caches, and the caches are inside the GT 440 chip.

The NVIDIA GT 440 chip itself is marked *B*. It is a small black square sitting on a larger green square. The SMs, including shared memory, caches, CUDA cores, and registers, are all located inside the chip.

The golden pins marked with a *C* plug into the system's motherboard. It is through these pins that the host communicates with the device. There are two related terms, "on-device" and "on-chip". On-device means anywhere on the entire card, while on-chip means inside the GT 440 chip itself.

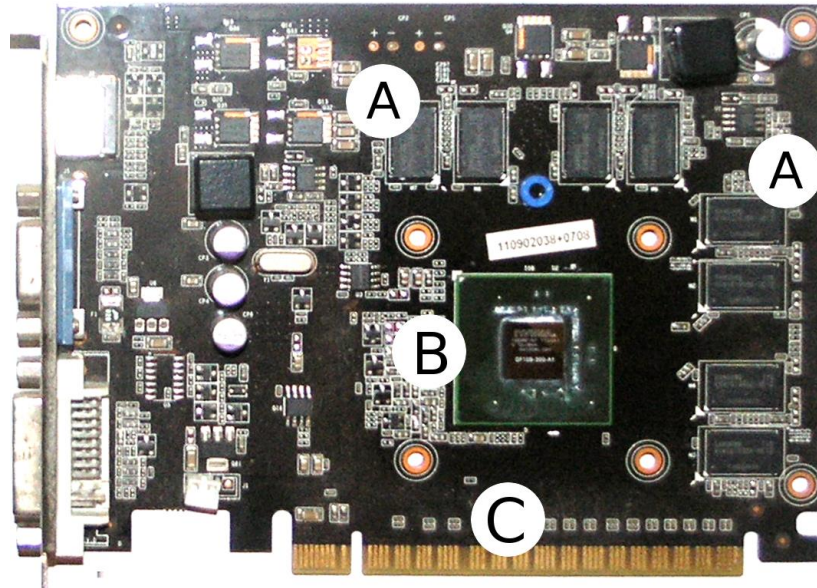


Figure 3.6: Graphics Card

## Registers

The fastest memory in the device is the registers. Registers are fundamental to the device and have no address because they are not in any type of RAM. The registers are hardware variables exposed to the programmer in software. They are essentially the workhorses of the device. They are on-chip (which, again, means they are physically located inside the SM). The SMs have a register file consisting of some limited number of registers which they can divvy out to the concurrent blocks.



**Note:** All computations are performed in the registers. This fact is often hidden from the programmer. There are no instructions which perform arithmetic, logic, or shifting on any of the other memories. When the device executes an operation on data in another memory space, it first copies the data into a register, then performs the operation on that data before copying the results back to the original location.

Threads do not share registers; each thread is given its own collection of registers with which to work. When the device runs out of registers, it causes what is called register spilling. The device will use global memory for extra storage space. When global memory is used in this way, it is called local memory. In this sense, local memory is not a distinct type of memory; it is the device's way of using global memory for extra storage when it runs out of registers. We will discuss local memory in more detail later; it is used under other circumstances besides coping with register spilling.



**Tip:** The value reported as “Total number of registers available per block” in Device Query is the size of the register file per SM. To find the total number of registers available on your device, multiply this by the number of SMs. For instance, with the GT 430 in this

*machine, the number of multiprocessors (SMs) is 2, and there are 32,768 registers per block, so the total number of registers available on this device is 65,536.*

To use a register in a kernel, you can simply declare a variable. If there are enough registers free (i.e. there is no register spilling), this will result in the variable being stored in a register. If, however, there are not enough registers free, the variable will be created in local memory and will physically reside in global memory. The distinction between registers, local memory, and global memory can be confusing. Generally speaking, you want to use as few registers per thread as possible and avoid register spilling. Local memory is not a distinct type of memory; it is a usage of global memory.

## Shared Memory

Shared memory is shared between the threads of a block. It is very fast, running at almost the same speed as the registers, and it is on-chip. Shared memory is the same physical memory as the L1 cache for global memory (see the [Caches](#) section that follows). Shared memory is a way for programmers to control the L1 cache in the device. We will examine it in detail in [Chapters 6](#) and [7](#) since careful use of shared memory can improve the performance of code substantially. Understanding shared memory is a big part of efficiently using the device.

## Caches

On modern devices (from the Fermi generation and newer), there are two caches, L1 and L2. These caches are controlled almost entirely by the device. The L1 cache is on-chip, so it is exceptionally fast (only the registers are faster). The L2 cache is slower than the L1 cache but is still much faster than global memory. All reads and writes within global memory, including data copied from the host, gets processed in the L2 cache. When a thread makes a global memory request, the device first checks to see if the request can be satisfied from the L1 cache and, if it cannot, then the L2 cache is checked second. Only when neither of the caches can satisfy the request is global memory read; this final read is the slowest type of read available.

Caches are an automatic memory optimization system which the device controls to help reduce traffic on the global memory bus. The use of the caches by the device is fairly simple: when data is read from global memory for the first time, it is stored in the L1 cache (usually). The device assumes that the same data is likely to be requested again, and storing a copy in the L1 will mean that if it is requested again, it can be read very quickly the second time. After a while, as more and more values are stored in the L1, it eventually becomes full. At this point, when more data is read from global memory, the L1 cache must evict the oldest (least likely to be read) values and replace them with the newer data. Data evicted from the L1 goes to the L2 cache.

The L2 cache is much farther away from the chip than the L1 cache (leading to slower access times) but it is still closer than using global memory. The L2 cache is much larger than the L1 cache (see Device Query for the actual sizes of these memories on your device) but, unfortunately, must also eventually evict stale data in the same manner as the L1 cache.

As mentioned in the Shared Memory section, the L1 cache and shared memory are actually the same physical memory. Shared memory is a way to control the L1 cache.

## Local Memory

Local memory is local to each thread. Data in local memory is not shared by any other threads in the grid. Local memory is physically the same memory as global memory. There are two circumstances under which global memory might be called local memory. Always remember that local memory is not a distinct type of memory but, rather, it is a particular way of using global memory.

The first circumstance under which global memory might be referred to as local memory is when register spilling occurs. When we create local variables within a thread's code (with normal variable declarations `int j`, `float q`, etc.), the device will assign these variables to a register but only if there are enough registers available. If there are no registers available, then the device will store the variable in global memory; this use of global memory is called local memory. The term “local memory” refers to the fact that other threads are not meant to access these variables. The variables are supposed to be local to each thread, just like the registers. The partitioning of variables into registers and memory is automatic. Local memory is far slower than the registers and you will generally want to minimize local memory usage as much as possible. You can examine the amount of local memory a kernel uses by profiling your application (see [Chapter 8](#)).

The second circumstance under which global memory will be used (and will be called local memory) is when structures and arrays are used within a kernel. The device stores variables in registers when it can, but structures and arrays require pointers in order to access their elements. When the device accesses a particular element of a structure or an array, it uses a base and offset pointer behind the scenes. This means that in order to use a structure or an array in a kernel, the variables require addresses to which they can be pointed. Registers do not have addresses so they cannot be used to store and access the elements of a structure or an array. So, structures and arrays are stored in the device's global memory. Just like register spilling, this usage of global memory is automatic and it is called local memory.

## Constant Memory

Constant memory is a special memory on the device specifically designed for storing grid-wide constants. Constant memory cannot be set by the device (hence its name) but the host can set values in constant memory prior to calling a kernel (in this sense, it is not constant at all). Constant memory is cached in its own cache, unrelated to the L1 and L2 caches present in global memory.

Constant memory is very fast when all of the threads of a block read the same value (shared memory speeds). To create a variable in constant memory, mark it as `__constant__`. Constant memory is actually stored in global memory but, as mentioned earlier, because of its dedicated cache, it can be accessed very fast.



**Tip:** Use C++ constants (i.e. the `const` keyword) if the value of a variable will never change, and use `__constant__` (device's constant memory) if the host needs the ability to alter the variable during the program's execution. C++ constants become immediate values in machine code and do not take any cycles to read from memory (other than the initial reading of the instruction). On the other hand, device constants require a read from

*the constant memory. Constant memory is fast when cached but not as fast as no read at all.*

If all of the threads of a warp do not access exactly the same value from constant memory, the accesses will be serialized. If all threads of a warp do access exactly the same value, the access will be extremely fast, the memory will be read once from the constant cache, and will be broadcast to the threads. If the requested memory is not in the constant cache, then the first read will be slow as the data must come from global memory; subsequent reads will result in a fast cache hit.

The host can change the value of constant memory using the `cudaMemcpyToSymbol` API function call. The syntax is as follows:

```
cudaError_t cudaMemcpyToSymbol(__constant__ devConst, const void* src_ptr,
size_t size);
```

The first parameter is a pointer to a `__constant__` qualified device variable. The second parameter is a pointer to the data to copy from the host, and the final parameter is the number of bytes to copy.

Listing 3.3 contains an example of setting a device constant with the `cudaMemcpyToSymbol` function and reading the value in a device kernel. We are skipping ahead a little and most of the code in Listing 3.3 is completely new. We will not talk about constant memory again; this code is provided as a reference. Do not feel that you should be familiar with these syntaxes as they will be covered in the next chapter.

```
#include <iostream>
#include <cuda.h>

using namespace std;

// The device's scale constant
__device__ __constant__ float DEVICE_CONST_SCALE;

// This kernel scales values in arr by the constant:
__global__ void ScaleArray(float* arr, int count)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if(idx < count) arr[idx] *= DEVICE_CONST_SCALE;
}

int main() {
    // The host's scale constant
    float scale;

    // Arrays of floats for host and device
    float arr[100], *d_arr;

    // Generate random floats on the host:
    cout<<"The unscaled array values are:"<<endl;
```

```

for(int i = 0; i < 100; i++) {
    arr[i] = (float)(rand() % 1000) / 1000.0f;
    cout<<i<<" ". "<<arr[i]<<endl;
}

// Read a scale value from the user:
cout<<"Enter a scale value:"; cin>>scale;

// Set this scale to the DEVICE_CONST_SCALE in constant memory
cudaMemcpyToSymbol(DEVICE_CONST_SCALE, &scale, sizeof(float));

// Malloc and copy the arr from host to device:
cudaMalloc(&d_arr, sizeof(float) * 100);
cudaMemcpy(d_arr, arr, sizeof(float) * 100, cudaMemcpyHostToDevice);

// Scale the values in the array on the device
ScaleArray<<<1, 100>>>(d_arr, 100);

// Copy the results back to the host and free
cudaMemcpy(arr, d_arr, sizeof(float) * 100, cudaMemcpyDeviceToHost);
cudaFree(d_arr);

// Cout the values to make sure they scaled:
cout<<"The scaled array values are:"<<endl;
for(int i = 0; i < 100; i++) cout<<i<<" ". "<<arr[i]<<endl;

return 0;
}

```

Listing 3.3: Writing and reading device constants



**Note:** In earlier versions of CUDA, the prototype for the `cudaMemcpyToSymbol` function was very different to what it is now. Many of the online sources and tutorials still reference the earlier function prototype which no longer works as expected. The original syntax included a string as its first parameter; this string was the symbol's name (e.g, `DEVICE_CONST_SCALE` as previously shown in Listing 3.3). The use of strings to specify the names of device symbols has been deprecated in more recent versions of the CUDA toolkit.

## Texture Memory

Texture memory is another special kind of memory. This memory belies the fact that the device is designed to work with graphics. Texture memory is cached in its own cache (like the constant memory), and is designed to store and index pixels in bitmap textures and images. Texture memory has some interesting (and very useful) indexing abilities. For example, it is able to automatically and very quickly interpolate several values from an array together or to normalize an array to a consistent median. Texture memory uses device memory (global memory) but it has its own cache.



The access speed to texture memory works out the same as that of constant memory. If data is cached, the read will be very fast; otherwise it must be read from global memory. For the remainder of this book, we will not be addressing the use of texture memory in any of the tutorials presented.

## Global Memory

Global memory is the largest store of memory available on the device but is also the slowest to access compared to the on-chip memories previously described. The `cudaMemcpy`, `cudaMalloc`, and `cudaMemset` functions all reference global memory (we will learn more about these functions in the next chapter). Global memory is cached with two levels of cache: L1 and L2. Global memory is extremely important because the host is able to communicate with the device via the use of global memory.

The general flow of a CUDA kernel is for the host to copy some data to global memory using the CUDA API functions. The kernel is then executed by many threads at once which, in turn, reads the data from global memory and places it into other memory spaces available on the device. A result is then calculated and stored back in global memory. The host then copies the result from global memory back to the system using the CUDA API functions.

## Memories Summary

This section summarizes some of the key features of each memory area for convenience. Figure 3.6 illustrates the memories in relation to the core of the SM (yellow box to the right). The slowest memory is generally that which is furthest from the core of the SM. The left-hand side of the diagram shows system memory; this memory must be copied to global memory over the PCI bus using the CUDA API functions. The next slowest memories are those that reside in the device's main memory storage; these include global memory, local memory, constant memory, and texture memory. The global memory L2 cache is close to the SM but not on-chip. It is illustrated in the diagram as being between the global memory and the SM. The memories on the right-hand side are the fastest of all; they are all on-chip. The registers are the fastest available of all the memories.



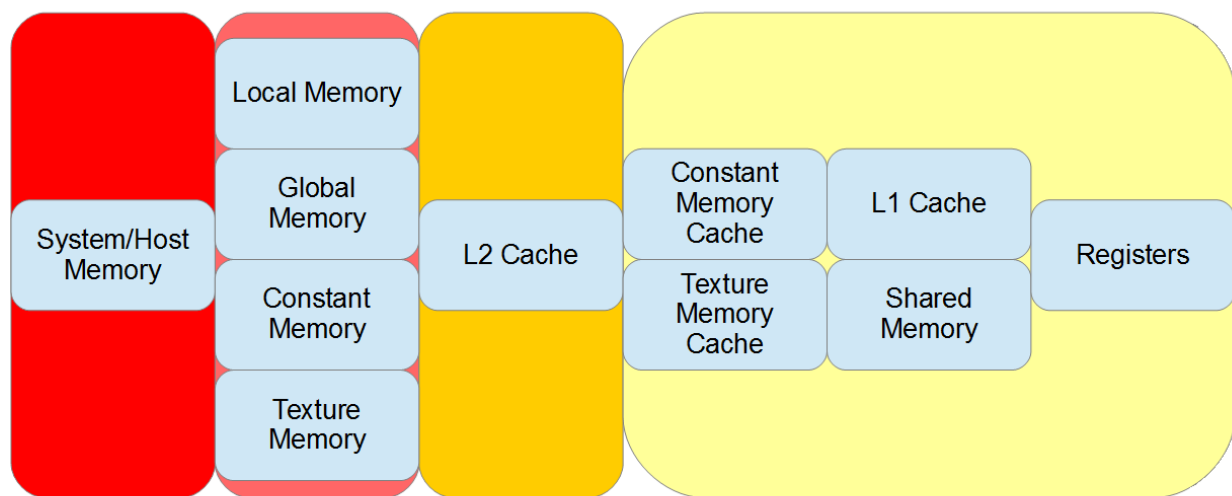


Figure 3.6: Memories ordered by access speed

The following table summarizes some of the key aspects of the CUDA memory spaces. Global memory speed is dependent on the current L1 and L2 cache usage and may not be slow at all.

Table 3.1: CUDA memories summary

Memory	Scope	Speed	Notes
System	Host	-	Device cannot read/write system RAM.
Registers	Thread	Very Fast	Limited supply.
L1 Cache	Block/SM	Fast	Device-controlled, same memory as shared memory.
L2 Cache	Grid	Slow	Device-controlled, all global memory r/w goes through L2.
Shared	Block	Fast	User-controlled L1 cache memory.
Texture	Grid	Slow/Fast	Uses global memory but has its own cache.
Constant	Grid	Slow/Fast	Uses global memory but has its own cache, read only.
Local	Thread	Slow	Used for local variables when registers cannot be used.

# Chapter 4 First Kernels

## Adding Two Integers

The project described in [Listing 2.1](#) of Chapter 2 will serve as the template for later projects. All of the code I present in this book should be placed in .cu files as described previously (unless otherwise specified). These files should be placed in a project which links to the CUDA libraries and appropriate CUDA headers as shown in Chapter 2.

Our first kernel will add two integers together on the device and return the resulting sum to the host.



**Tip:** Type the code into your IDE instead of using copy and paste, especially for these early examples. Copying and pasting code may paste some characters that are not what the compiler expects.

```
#include <iostream>
#include <cuda.h>

using namespace std;

__global__ void AddInts(int* a, int *b) {
    a[0]+=b[0];
}

int main() {
    int a, b;           // Host copies
    int *d_a, *d_b;     // Device copies

    // Read some integers from the user
    cout<<"Input a number? ";
    cin>>a;
    cout<<"And another? ";
    cin>>b;

    // Allocate RAM on the device
    if(cudaMalloc(&d_a, sizeof(float)) != CUDA_SUCCESS) {
        cout<<"There was a problem allocating memory on the GPU"<<endl;
        cudaFree(d_a);
        cudaDeviceReset();
        return 0;
    }

    if(cudaMalloc(&d_b, sizeof(float)) != CUDA_SUCCESS) {
        cout<<"There was a problem allocating memory on the GPU"<<endl;
        cudaFree(d_a);
        cudaFree(d_b);
        cudaDeviceReset();
    }
}
```

```

    return 0;
}

// Copy host values to device
cudaMemcpy(d_a, &a, sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, sizeof(float), cudaMemcpyHostToDevice);

// Run kernel
AddInts<<<1, 1>>>(d_a, d_b);

// Copy results back to host
cudaMemcpy(&a, d_a, sizeof(float), cudaMemcpyDeviceToHost);

cout<<"The GPU addition results in "<<a<<endl;

// Free device memory
cudaFree(d_a);
cudaFree(d_b);

// Reset device and write performance indicators
cudaDeviceReset();
return 0;
}

```

Listing 4.1: Adding Integers

Listing 4.1 requests two integers from the user using `std::cout` and `std::cin`. It copies them to the device and adds them together using a CUDA kernel. The result of the addition is copied from device memory back to system memory and displayed to the user.



**Tip:** The console window opens and closes too quickly to read the results. To prevent the window from closing, you can place a breakpoint on the `return 0;` statement at the end of the main function. When a program is debugged, Visual Studio will pause execution when it reaches the breakpoint on this line and it will give the programmer time to view the output console.

## Function Qualifiers

Functions can be designed to execute on the host, the device, or both. CUDA provides several function qualifiers which are placed at the start of the function declaration and describe the hardware on which the function is meant to be executed.

- **\_\_global\_\_** Functions marked as **\_\_global\_\_** are callable from the host but they run on the device. They are CUDA kernels that the host calls. They cannot be recursive, cannot have variable parameter lists, and they must return void.

- **\_\_host\_\_** Functions marked as **\_\_host\_\_** are called and executed by the host. These are perfectly normal C++ functions. **\_\_host\_\_** is the default and, if a function has no qualifiers at all (such as `main` in Listing 4.1), it is assumed to be a normal C++ host function.
- **\_\_device\_\_** Functions marked as device are called and executed on the device. They are often helper functions for kernels. **\_\_device\_\_** functions cannot be called from the host. They can only be called from kernels or other device functions.
- **\_\_host\_\_ \_\_device\_\_** Functions marked as both **\_\_host\_\_** and **\_\_device\_\_** will be callable from the host or the device. This qualifier actually results in two functions being compiled: one for the host and the other for the device. The host version is a perfectly normal C++ function and is not able to alter the device's memory. Likewise, the device version of the function is a **\_\_device\_\_** function and is not able to alter any host variables. This qualifier is used to define helper functions that both the host and the device might call.

## CUDA API Memory Functions and `cudaError_t`

CUDA API functions return a `cudaError_t` which indicates if the call was successful. You can check the return value of a CUDA API function against `CUDA_SUCCESS` to determine if a function executed properly. There are many `cudaError_t` values the CUDA API can return. For a full list, see the `cuda.h` header.

```
cudaError_t cudaMalloc((void**) devPtr, size_t size);
```

**cudaMalloc** allocates global memory on the device. The first parameter is the address of a pointer to the allocated memory. If the call is successful, this pointer will be initialized to point to the newly allocated memory on the device. The second parameter is the size, in bytes, to allocate to the requested memory area.

```
cudaError_t cudaFree(void* devPtr);
```

This function frees memory previously allocated on the device. Memory should always be freed on the device when it is no longer needed. The first parameter is the pointer previously allocated in a **cudaMalloc** call; each call to **free** should always be matched by a preceding call to **Malloc**.

```
cudaError_t cudaDeviceReset();
```

This function cleans up allocated memory and resets the state of the device. This function is similar to the garbage collector in .NET applications: it tidies things up. But unlike the garbage collector, the programmer must call this function explicitly. Once reset, the device can be returned to its initialized state by calling any CUDA API function. Calling this function also causes all the performance counters to be written by the device. If you are planning to profile a kernel, or your CUDA code in general, then you should always make sure you call this function before your program shuts down. This ensures the data is written correctly to the performance counters for the profiler to use.

```
cudaError_t cudaMemcpy(void* dest, void* src, size_t size, cudaMemcpyKind direction);
```

This function copies memory from the **src** pointer (source) to the **dest** pointer (destination). The **size** parameter is the size, in bytes, of the data to be copied. The final parameter specifies the direction of the copy. The direction can be any one of the following:

- **cudaMemcpyHostToHost**
- **cudaMemcpyHostToDevice**
- **cudaMemcpyDeviceToHost**
- **cudaMemcpyDeviceToDevice**

The pointers supplied as **dest** and **src** must agree with the direction selected. For example, using **cudaMemcpyHostToDevice**, the **src** pointer is a host pointer and the **dest** pointer is a device pointer.

## Copying Data to and from the GPU

The next example illustrates copying a block of data from the device to the host and using the CUDA API function **cudaMemset** to zero the array. The **cudaMemcpy** function copies data over the PCI bus to and from the device's global memory. In the following example, the GPU is used to clear an array of floating point values to **0.0f**.

```
#include <iostream>
#include <cuda.h>

using namespace std;

int main() {
    float values[100]; // CPU copy of an array
    float* d_Values;    // Pointer to device copy of values

    // Print the initial values to screen
    for(int i = 0; i < 100; i++)
        cout<<values[i]<<" "; // These will initially be random garbage

    // Allocate RAM on the GPU the same size as the values
    if(cudaMalloc(&d_Values, sizeof(float) * 100) != CUDA_SUCCESS) {
        cout<<"There was a problem allocating ram on the GPU"<<endl;
        return 0;
    }

    // Set the GPU ram to 0, floats with all bits as 0 in IEEE are = 0.0f
    if(cudaMemset(d_Values, 0, sizeof(float) * 100) != CUDA_SUCCESS) {
        cout<<"There was a problem setting the values to 0"<<endl;
    }
    else {
        // Copy this array of 0s to the CPU's array of values
        if(cudaMemcpy(values, d_Values, sizeof(float) * 100,
```

```

        cudaMemcpyDeviceToHost) != CUDA_SUCCESS) {
            cout<<"There was a problem copying the data from the GPU"<<endl;
        }
    }

    // Free the GPU's array
    cudaFree(d_Values);

    // Print out the CPU's array to make sure they have all been set to 0.0f
    for(int i = 0; i < 100; i++)
        cout<<values[i]<<" ";

    cudaDeviceReset();

    return 0;
}

```

*Listing 4.2: cudaMemset*

The new API function in Listing 4.2 is **cudaMemset**:

```
cudaError_t cudaMemset(void* devPtr, int value, size_t size);
```

This function can be used to set the elements of an array to some initial value. The first operand is a device pointer to the data to be set. The second operand is the value to which you wish to set each byte in the allocated memory, and the final operand is the size, in bytes, of the memory area to set. The function sets the allocated memory to a byte-sized value, meaning its usefulness for setting things such as floating point or integer values is quite limited—unless you are initializing everything to 0.

There are several steps involved in clearing an array to 0 using the CUDA API. First, there must be host and device copies of the same data. The host copy is called **values[]** in Listing 4.3 and the device copy is called **d\_Values**. Data is allocated on the device to be the same size as it is on the host. It is then set to 0 with a call to **cudaMemset** and then the device memory is copied over the PCI bus back to the host, again using the **cudaMemcpy** function.

Although Listing 4.3 illustrates a very basic use of the device (it would be far more practical to simply zero the array with the host), the steps involved (i.e. copying data back and forth over the PCI bus with **cudaMemcpy**) are very common in CUDA programming. Also very common is the use of multiple pointers to point to host and device copies of the same data. Host pointers do not point to sensible areas of the device and vice versa.

## Vector Addition Kernel

The next example illustrates adding two floating point vectors together.

```

#include <iostream>
#include <ctime>

```

```

#include <cuda.h>

using namespace std;

__global__ void AddVectors(float* a, float* b, int count)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if(idx < count) {
        a[idx] += b[idx];
    }
}

int main() {
    // Number of items in the arrays
    const int count = 100;

    // CPU arrays
    float a[count], b[count];

    // Device pointers
    float* d_a, *d_b;

    // Set the random seed for rand()
    srand(time(NULL));

    // Set the initial values of the CPU arrays
    for(int i = 0; i < count; i++) {
        a[i] = (float)(rand() % 100);
        b[i] = (float)(rand() % 100);
    }

    // Allocate data on the device
    if(cudaMalloc(&d_a, sizeof(float) * count) != CUDA_SUCCESS) {
        cout<<"Memory could not be allocated on the device!"<<endl;
        cudaDeviceReset();
        return 0;
    }
    if(cudaMalloc(&d_b, sizeof(float) * count) != CUDA_SUCCESS) {
        cout<<"Memory could not be allocated on the device!"<<endl;
        cudaFree(d_a);
        cudaDeviceReset();
        return 0;
    }

    // Copy from host to device
    cudaMemcpy(d_a, &a, sizeof(float) * count, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, sizeof(float) * count, cudaMemcpyHostToDevice);

    dim3 gridSize((count / 512) + 1);
    dim3 blockSize(512);
    AddVectors<<<gridSize, blockSize>>>(d_a, d_b, count);

    cudaMemcpy(&a, d_a, sizeof(float) * count, cudaMemcpyDeviceToHost);

```

```

// Print out the results
for(int i = 0; i < count; i++)
    cout<<"Result["<<i<<"]="<<a[i]<<endl;

// Free resources
cudaFree(d_a);
cudaFree(d_b);
cudaDeviceReset();

return 0;
}

```

*Listing 4.3: Adding vectors*

Listing 4.3 defines two floating point arrays on the host, **a** and **b**. It sets the values of the arrays to random integers using the **srand** and **rand** functions. It then copies the two arrays to the device pointers **d\_a** and **d\_b** and calls a kernel called **AddVectors**. The kernel adds the values from the **b** vector to the corresponding values in the **a** vector and stores the result in the **a** vector. This is then copied back to the host and printed to the screen. Note the use of the **dim3** parameters in the launch configuration and the launch of many threads at once.

The kernel illustrates a very common pattern and a major difference between coding with CUDA and regular serial code. In serial code, the previous algorithm would be executed by a loop similar to that shown in Listing 4.4.

```

for(int i = 0; i < count; i++) {
    a[i] += b[i];
}

```

*Listing 4.4: Serial vector addition*

The serial version of the algorithm uses a loop to iterate through the two arrays, adding a single element each iteration of the loop. The parallel version (as shown in the kernel from Listing 4.3) uses multiple threads in place of a loop.



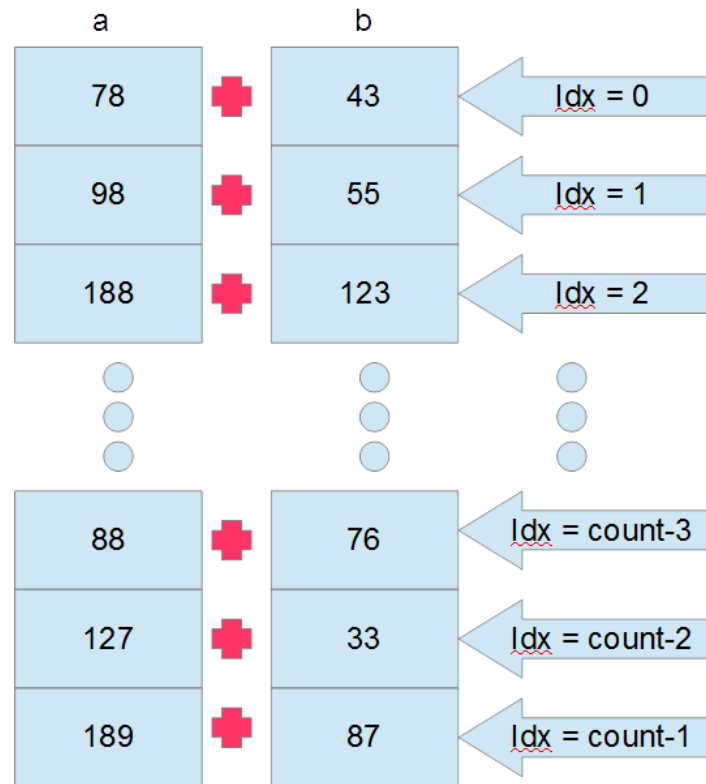


Figure 4.1: Parallel Vector Addition

Figure 4.1 illustrates how the vector addition takes place on the GPU. The vector arrays **a** and **b** (which are actually device pointers **d\_a** and **d\_b** passed to the kernel as parameters) are illustrated as a series of vertically stacked boxes, each containing a random value originally copied from the host. Each thread takes care of a single addition and all run at the same time (execution is only simultaneous in theory, some threads may actually be scheduled to execute after others. See the note that follows). The 20<sup>th</sup> thread adds together values **a[20]** and **b[20]**, and the 13<sup>th</sup> thread adds **a[13]** and **b[13]**. There is no loop; each thread calculates a unique ID, called **idx** in Listing 4.3. The **idx** variable is used to index data elements from the array which are particular to the thread.



**Note:** The devices we are programming are massively parallel. The sheer number of parallel threads enables a GPU to outperform a CPU for many computations. But depending on the number of threads in the grid, the device will often schedule thread blocks to execute one after the other instead of simultaneously. The device does not have unlimited resources; it executes as many blocks as resources permit, and it schedules the remaining blocks for later execution as resources become available. Maximizing the number of simultaneous threads involves a metric called occupancy. We will examine how to view a kernel's occupancy with a profiler in [Chapter 8](#).

Calculating a unique ID for each thread (like the `idx` variable in Listing 4.3) is very common. We often need to partition a problem into smaller subproblems so that each thread can tackle some tiny bit of the whole problem. It is common that threads each work on some unique unit of the problem, like the additions in Figure 4.1 where each thread performs a single addition. For example, observe the following line.

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
```

This line calculates a unique thread ID. The IDs are in sequence from 0 up to however many threads were launched, minus one. This calculation can be used whenever the grid and blocks have a single dimension. If you are working with multidimensional grids, blocks, or both, you will still need to account for the extra dimensions in the calculation.

## Recursive Device Functions

Listing 4.5 shows a recursive device function to calculate an integer factorial.

Recursive device functions are not supported on compute capability 1.x devices. If you have such a device, the code in the following example will not work.

If, however, you have a compute capability 2.x or later device, the code will run fine; however, you will need to be sure that the code is being compiled for this compute capability.

The compute capability and SM settings are specified in the project's properties. Under the **CUDA C/C++** section called **Device**, you will see an option for **Code Generation**. Set this to match your device (e.g., `compute_20, sm_20`). I present further information on this in the [Tips and Tricks](#) section of Chapter 5, under the details on setting code generation options.

```
// Illustration of Factorial with recursive device function
#include <iostream>
#include <cuda.h>
using namespace std;
// Recursive device function
__device__ int Factorial(int x) {
    if(x == 0) return 1;
    else return x * Factorial(x-1);
}

// Kernel
__global__ void MyKernel(int* answer) {
    answer[0] = Factorial(answer[0]);
}

int main() {
    int i, answer, *d_i;

    // Read an int from user
    cout<<"Input a number: ";
    cin>>i;
```

```

// Copy int to device
cudaMalloc(&d_i, sizeof(int));
cudaMemcpy(d_i, &i, sizeof(int), cudaMemcpyHostToDevice);

// Launch kernel
MyKernel<<<1, 1>>>(d_i);

// Copy answer back to host
cudaMemcpy(&answer, d_i, sizeof(int), cudaMemcpyDeviceToHost);

// Print out
cout<<"The answer is "<<answer<<endl;

cudaFree(d_i);
return 0;
}

```

*Listing 4.5: Recursive device functions*

Listing 4.5 illustrates a simple recursive device function but it is very limited in the number the user can type. The **Factorial** function quickly overflows 32-bit integers. This leads to incorrect results for any user input greater than 12.

# Chapter 5 Porting from C++

We examined some very simple uses of the device in the previous chapter. They are not practical examples, and the host would probably perform the calculations of all the previously cited kernels much faster than the device. In this chapter, we will introduce a problem that is better suited to a device using CUDA (and parallel processing in general) than any general purpose CPU would be.

## C++ Host Version

One of the most common reasons one might be interested in CUDA is to optimize an algorithm that has already been written for the host in C++. In this chapter, we will examine porting a simple C++ algorithm to CUDA to see if the device can offer any performance improvements over the original host version. We will write a direct port from standard C++ code straight into CUDA-based code.

The algorithm works on an array of 3-D points and, for each point, finds the nearest neighbor from the other points in the list. The host code (Listing 5.1) finds each nearest neighbor by traversing the array as many times as there are points.



**Note:** *There are well-known data structures that can dramatically improve the number of comparisons involved with solving the nearest neighbor problem. An implementation using k-d trees will greatly reduce the number of comparisons.*

```
// main.cu
#include <iostream>
#include <ctime>
#include <cuda_runtime.h> // For float3 structure

#include "FindClosestCPU.h"

using namespace std;

int main() {
    // Number of points
    const int count = 10000;

    // Arrays of points
    int *indexOfClosest = new int[count];

    float3 *points = new float3[count];

    // Create a list of random points
    for(int i = 0; i < count; i++) {
        points[i].x = (float)((rand()%10000) - 5000);
        points[i].y = (float)((rand()%10000) - 5000);
    }
```

```

        points[i].z = (float)((rand()%10000) - 5000);
    }

    // This variable is used to keep track of the fastest time so far
    long fastest = 1000000;

    // Run the algorithm 10 times
    for(int q = 0; q < 10; q++) {
        long startTime = clock();

        // Run the algorithm
        FindClosestCPU(points, indexOfClosest, count);

        long finishTime = clock();

        cout<<q<<" "<<(finishTime - startTime)<<endl;

        // If that run was faster, update the fastest time so far
        if((finishTime - startTime) < fastest)
            fastest = (finishTime - startTime);
    }

    // Print out the fastest time
    cout<<"Fastest time: "<<fastest<<endl;

    // Print the final results to the screen
    cout<<"Final results:"<<endl;
    for(int i = 0; i < 10; i++)
        cout<<i<<"."<<indexOfClosest[i]<<endl;

    // Deallocate RAM
    delete[] indexOfClosest;
    delete[] points;

    return 0;
}

```

*Listing 5.1: Front end, main.cu*

Listing 5.1 is the front end to the algorithm. It is a small C++ program that creates a list of 10,000 random 3-D points. It then runs the **FindClosestCPU** function which finds the nearest neighbors to each point. The **FindClosestCPU** function is defined in Listing 5.2, **FindClosestCPU.h**. To add the new header to your project, right-click the project's name in the Solution Explorer panel, and click **Add > New Item** in the context menu.



**Note:** In the host code (Listing 5.2) and the device code, we are not actually computing the distance between points. The real distance between points would require an additional square root function. In this example, we do not need the actual distance; we only need to find the nearest point; therefore, the square root function has been omitted.

```

// FindClosestCPU.h
#pragma once

#include <cuda_runtime.h> // For float3 structure

// FindClosestCPU host function
void FindClosestCPU(float3* points, int* indices, int count) {
    if(count <= 1) return; // If there are no points return

    // Loop through every point
    for(int curPoint = 0; curPoint < count; curPoint++) {
        // Assume the nearest distance is very far
        float distToClosest = 3.40282e38f;

        // Run through all the points again
        for(int i = 0; i < count; i++) {
            // Do not check distance to the same point
            if(i == curPoint) continue;

            // Find distance from points[curPoint] to points[i]
            float dist = ((points[curPoint].x - points[i].x) *
                (points[curPoint].x - points[i].x) +
                (points[curPoint].y - points[i].y) *
                (points[curPoint].y - points[i].y) +
                (points[curPoint].z - points[i].z) *
                (points[curPoint].z - points[i].z));

            // Is dist nearer than the closest so far?
            if(dist < distToClosest) {
                // Update the distance of the nearest point found so far
                distToClosest = dist;

                // Update index of this thread's nearest neighbor so far
                indices[curPoint] = i;
            }
        }
    }
}

```

*Listing 5.2: FindClosestCPU.h*

Listing 5.2 shows a simple method for finding the nearest neighbors. The CPU begins with the first point in the list and assumes the closest point to be a distance of 3.40282e38f away. This is around the maximum possible value for a **float** in C++. It steps through the array, checking the distance to each of the other points. Each time a point is found closer than the current smallest distance, the smallest distance is updated and the index of the point is recorded. By the time the CPU has traversed the entire array, it has found the index of the point nearest to the first (**points[0]**). It then repeats the process for the second point, then for the third, and so on until it has found the nearest neighbor to every point in the list.

Upon building and running the program, you should see a series of outputs similar to Figure 5.1. The first lines of the output show the amount of time (in milliseconds) that it took the CPU to run the **FindClosestCPU** algorithm.

```
0 188
1 188
2 185
3 184
4 183
5 181
6 182
7 183
8 187
9 185
Fastest time: 181
Final results:
0.1982
1.8115
2.4738
3.1162
4.4517
5.3983
6.4263
7.3513
8.9489
9.7961
```

Figure 5.1: Output

The fastest time the host ran the algorithm is reported in Figure 5.1 as 181 milliseconds. When we port to CUDA, it is this time which we hope to improve. The first 10 results are printed to the screen. The results in Figure 5.1 show that the nearest point to `point[0]` was found to be `point[1982]`, the nearest point to `point[1]` was `point[8115]`.



**Tip:** Use the host's results to check the device results. When porting host code to CUDA, it is a very good idea to check the results that the device is finding against the results that the original host code finds. Host code ported to CUDA is useless if the device gives incorrect results. Sometimes (especially without using a debugger) it is very difficult to know that the device's results are correct just by looking at the kernel's code.

## CUDA Version

C++ code can be ported in many different ways. One of the initial questions is, "How can the problem be broken into separate threads?" There is very often more than one way to do this. Our current problem is multiple nearest neighbor searches. A natural way to break this particular problem into threads is to assign a single point to each thread we may have available which, ultimately, will run on its own individual kernel. Each thread in this case will be responsible for finding the nearest neighbor of its own point.

The host code employed a nested for loop; by assigning an individual thread each to a different point, we will remove one of these loops (the outer loop) and replace it with our concurrent CUDA threads.

The **FindClosestGPU.h** in Listing 5.3 contains the prototype for the device version of the algorithm; **FindClosestGPU** is a CUDA kernel.

```
// FindClosestGPU.h
#pragma once

#include <cuda_runtime.h>

__global__ void FindClosestGPU(float3* points, int* indices, int count);
```

*Listing 5.3: FindClosestGPU.h*

The **FindClosestGPU.cu** in Listing 5.4 contains the CUDA kernel **FindClosestGPU** which performs the nearest neighbor searches. To add this file to your project, right-click the project's name in the Solution Explorer and click **Add > New Item** in the context menu. This is a very basic and direct port of the original **FindClosestCPU** function to CUDA. The outer **for** loop from the original code is replaced by multiple threads, but other than this, the two functions are identical.

```
// FindClosestGPU.cu
#include <cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>

#include "FindClosestGPU.h"          // Not required!

// FindClosestGPU kernel
__global__ void FindClosestGPU(float3* points, int* indices, int count) {
    if(count <= 1) return;          // If there are no points return

    // Calculate unique thread idx
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    // If the calculated thread idx is within the bounds of the array
    if(idx < count) {
        // Assume the nearest distance is very far
        float distanceToClosest = 3.40282e38f;

        // Run through all the points
        for(int i = 0; i < count; i++) {
            // Do not check distance to this thread's point
            if(i == idx) continue;

            // Find distance from this thread's point to another point
            float dist = (points[idx].x - points[i].x) *
                (points[idx].x - points[i].x) +
                (points[idx].y - points[i].y) *
                (points[idx].y - points[i].y) +
                (points[idx].z - points[i].z) *
                (points[idx].z - points[i].z);

            // Is distance nearer than the closest so far?
```



```

        if(dist < distanceToClosest) {
            // Update the distance of the nearest point found so far
            distanceToClosest = dist;

            // Update index of this thread's nearest neighbor so far
            indices[idx] = i;
        }
    }
}

```

Listing 5.4: FindClosestGPU.cu

Next, we need to alter the C++ front end (**main.cu**) to call the new CUDA version of the algorithm instead of the original C++ one. This involves copying the array of points to the device's global memory (**d\_points**) and allocating space for the device to store the indices (**d\_indexOfClosest**). The altered version of the main.cu file is presented in Listing 5.5.



**Note:** The following program requires more data and execution time than the previous programs. CUDA is low-level programming and it can cause your device or driver to reset. It can also cause the Blue Screen of Death, which results in the whole system resetting. When programming CUDA, it is advisable that you close any unnecessary applications and save your work in other applications before debugging a CUDA app.

```

// main.cu
#include <iostream>
#include <ctime>
#include <cuda.h>

// #include "FindClosestCPU.h"
#include "FindClosestGPU.h"

using namespace std;

int main()
{
    // Number of points
    const int count = 10000;

    // Arrays of points
    int *indexOfClosest = new int[count];

    float3 *points = new float3[count];
    float3* d_points;    // GPU version
    int* d_indexOfClosest;

    // Create a list of random points
    for(int i = 0; i < count; i++) {
        points[i].x = (float)((rand()%10000) - 5000);
        points[i].y = (float)((rand()%10000) - 5000);
        points[i].z = (float)((rand()%10000) - 5000);
    }
}

```

```

    }

    // Copy the points to the device and malloc space for results
    cudaMalloc(&d_points, sizeof(float3) * count);
    cudaMemcpy(d_points, points, sizeof(float3) * count, cudaMemcpyHostToDevice);
    cudaMalloc(&d_indexOfClosest, sizeof(int) * count);

    // This variable is used to keep track of the fastest time so far
    long fastest = 1000000;

    // Run the algorithm 10 times
    for(int q = 0; q < 10; q++) {
        long startTime = clock();

        // Run the algorithm
        FindClosestGPU<<<(count / 64)+1, 64, 64 * sizeof(float4)>>>
            (d_points, d_indexOfClosest, count);

        // Copy the results back to the host
        cudaMemcpy(indexOfClosest, d_indexOfClosest,
            sizeof(int) * count, cudaMemcpyDeviceToHost);

        long finishTime = clock();

        cout<<q<<" "<<(finishTime - startTime)<<endl;

        // If that run was faster, update the fastest time so far
        if((finishTime - startTime) < fastest)
            fastest = (finishTime - startTime);
    }

    // Print out the fastest time
    cout<<"Fastest time: "<<fastest<<endl;

    // Print the final results to the screen
    cout<<"Final results:"<<endl;
    for(int i = 0; i < 10; i++)
        cout<<i<<"."<<indexOfClosest[i]<<endl;

    // Deallocate ram
    delete[] indexOfClosest;
    delete[] points;
    cudaFree(d_points);
    cudaFree(d_indexOfClosest);

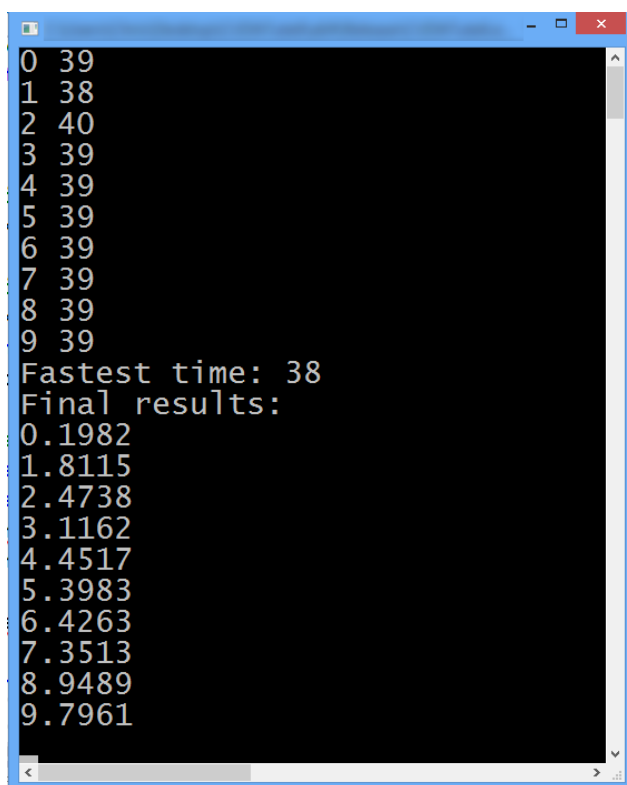
    // Dispose of the CUDA context and write device performance counters
    cudaDeviceReset();

    return 0;
}

```

*Listing 5.5: Updated main.cu*

Upon executing the modified application, we can see a vast improvement in performance. The actual increase in performance is completely dependent upon the power of the CPU as compared to the GPU in the machine running the kernel. Figure 5.2 shows the output from a machine with an Intel i7 2600 CPU and an NVIDIA GT 430 GPU.



```
0 39
1 38
2 40
3 39
4 39
5 39
6 39
7 39
8 39
9 39
Fastest time: 38
Final results:
0.1982
1.8115
2.4738
3.1162
4.4517
5.3983
6.4263
7.3513
8.9489
9.7961
```

*Figure 5.2: Output from FindClosestGPU Kernel*

The output in Figure 5.2 shows that the device kernel was able to perform the nearest neighbor searches with a minimum time of 38 milliseconds. Remembering that the host took around 181 milliseconds (Figure 5.1), the performance gain from this naïve port to CUDA is almost a 5x speed increase (the device only took about 21 percent of the time the host required). The design principle behind the port to CUDA was very simple: one of the nested **for** loops from the serial code was replaced by concurrent threads. This type of optimization would be extremely difficult to perform on the host version of the algorithm without using multithreading and the x86 SIMD instruction set, or without hand coding some assembly language. This type of performance gain is not just a simple speed increase but a major reduction in the amount of work needed. All of these techniques would require more effort than the simple CUDA port presented if you were not making use of the GPU on the device. In [Chapter 7](#), we will revisit this problem and increase the gap in performance between the device and host even further.

# Tips and Tricks

## Compile for 64-bit Hardware

Sometimes small changes in the project or program can provide better performance. To achieve the times in the previous example (Figure 5.2), the project was set to 64 bits. 64-bit machines (CPUs and GPUs) can often perform slightly faster when they run native 64-bit code rather than 32-bit.

## Allow an Epsilon between Host and Device Results

When checking the output from the device against a host output which is known to be correct, bear in mind the rounding errors. This is applicable when the algorithms deal with floating point data types. There is often going to be a slight difference between the device and the host output; a small margin of error might be allowed (often called epsilon).

## Use Release Mode and Compiler Optimization

When testing for performance, it is often best to use optimizations since the compiler (both NVCC and Microsoft C++ compiler) can give an excellent increase in speed automatically. The project (Listing 5.5) was compiled in release mode, with all optimizations turned on for both the host and device in order to produce the output shown in Figures 5.1 and 5.2.

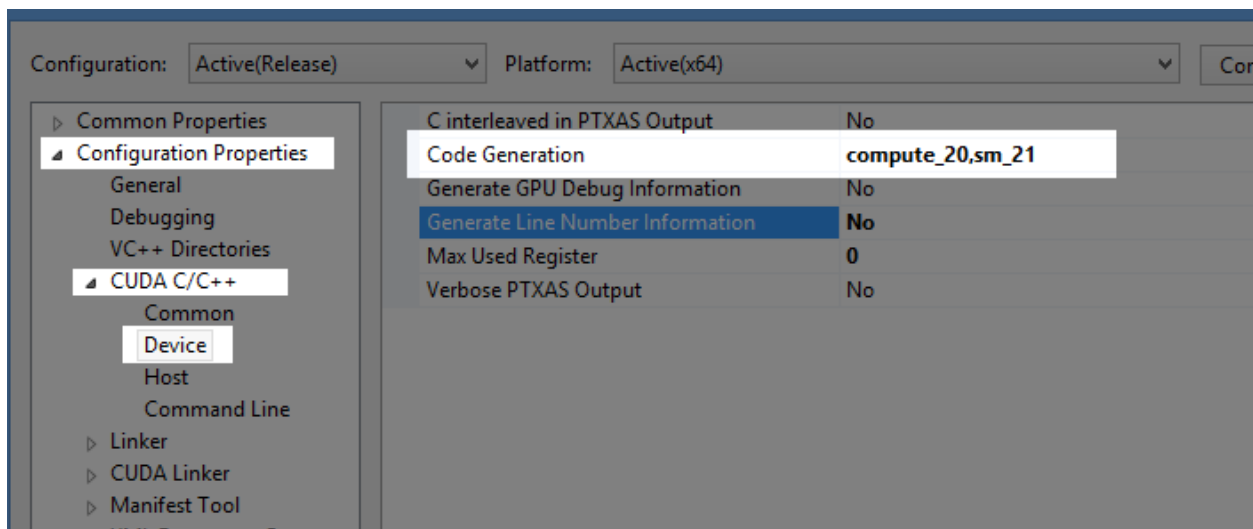


Figure 5.3: Setting Code Generation

The **Device > Code Generation** option was set to match the hardware in this machine (for the output in Figure 5.2). To set the Code Generation property, open the **Project Properties** and click **CUDA C/C++** in the left panel, and then click **Device**. Find the **Code Generation** setting in the right panel and select appropriate values for the compute capability (the **compute\_xx** setting) and **sm** version (**sm** here is not streaming multiprocessor! The next paragraph explains what it is).

When NVCC compiles a CUDA C/C++ file, it converts it into a quasi-assembly language called PTX. PTX is not actually the hardware's assembly language but it closely resembles it. The PTX code is then compiled to a real device's machine code. The **compute** setting is the PTX version to compile to, and the **sm** setting is the real machine (that is, the compute capability of the real hardware). If you are targeting more than one machine, it is best to use **compute** and **sm** values which match the lowest hardware you intend for your application. You can include more than one specification in the code generation by separating each by semicolons (for example, **compute\_10, sm\_10; compute\_20, sm\_20**).

For a complete list of NVIDIA's cards and their compute capabilities (these are the valid values for **sm**) visit the NVIDIA website at <https://developer.nvidia.com/cuda-gpus>.

The value supplied for the **compute** setting can be selected from the following bulleted list based on the required capabilities and the **sm** setting. Keep in mind that the first digits of the **compute** and **sm** values should match (i.e. **compute\_20, sm\_10** is an incompatible code generation request).

- **compute\_10**: Original, least features
- **compute\_11**: Atomic global memory operations
- **compute\_12**: Atomic shared memory operations
- **compute\_13**: Doubles (64-bit floats)
- **compute\_20**: Fermi hardware
- **compute\_30**: Kepler hardware

The information in this section was taken from the NVCC manual available online at <http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html>.

# Chapter 6 Shared Memory

Shared memory is a small amount of on-chip memory, ranging from 16 kilobytes to 48 kilobytes, depending on the configuration of the kernel. Remember that the term “on-chip” means inside the main processor of the graphics card (the chip is marked *B* in [Figure 3.6](#)).

Shared memory is very fast, approaching register speeds, but only when used optimally. Shared memory is allocated and used per block. All threads within the same thread block share the same allocation of shared memory, and each block allocation of shared memory is not visible to the other blocks running in the grid.

Because of the speed difference between global memory and shared memory, using shared memory is almost always preferred if the operation you’re going to perform permits efficient use of it. In this chapter, we will examine the efficient use of shared memory, but first we need to learn the syntax and API calls for basic shared memory usage.

## Static and Dynamic Allocation

Shared memory can be allocated per block in one of two ways: static or dynamic. The static method is less flexible and requires that the amount of shared memory be known at compile time. The dynamic method is slightly more complicated (syntactically), but allows for the amount of shared memory to be specified at runtime.

### Statically Allocated Shared Memory

To declare that a variable in a kernel is to be stored in shared memory, use the `__shared__` qualifier keyword beside the variable’s name in its definition (see Listing 6.1).

```
__global__ void MyKernel() {
    __shared__ int i; // Shared int
    __shared__ float f_array[10]; // 10 shared floats
    // ... Some other code
}
```

*Listing 6.1: Statically Allocated Shared Memory*

Listing 6.1 shows two variables declared as `__shared__`, an integer called `i`, and a floating point array called `f_array`. This shared memory (both the `i` and the `f_array` variables) are described as being statically allocated because the amount of shared memory must be constant at compile-time. Here, we are allocating an `int` and ten `floats` per block, for a total of 44 bytes of shared memory per block (both `int` and `float` are 4 bytes long).

All the threads in a given block will share these variables. Every block launched in a grid will allocate its own copy of these variables. If we launch a grid of 25 blocks, there will be 25 copies of `i` and 25 copies of `f_array` in shared memory. If the device does not have enough shared memory to allocate all 25 copies of the requested shared memory, then it cannot execute all the blocks simultaneously. The device executes as many blocks as it can simultaneously, but when there are not enough resources, it schedules some blocks to execute after others. Shared memory is one of the most important resources of the device and careful allocation of shared memory is an important factor for determining occupancy, which is described in [Chapter 8](#).

## Dynamically Allocated Shared Memory

The other way to allocate shared memory is to dynamically allocate it. This allows the amount of shared memory per kernel to change from launch to launch. In other words, the amount need not be a compile-time constant. To allocate a dynamic amount of shared memory for a kernel, we need to supply a third argument in the kernel launch configuration from the host. The code in Listing 6.2 shows an example launch configuration with this third parameter.

```
// Kernel
__global__ void SomeKernel() {
// The size of the following is set by the host
extern __shared__ char sharedbuffer[];
}

int main() {

// Other code

// Host launch configuration
SomeKernel<<<10, 23, 32>>>();

// Other code

}
```

*Listing 6.2: Dynamic Shared Memory Allocation*

The kernel launch configuration parameters (in the `main` method) specify to launch the kernel with **10** blocks of **13** threads each. The final parameter is the amount of dynamic shared memory to allocate per block. The parameter is measured in bytes, so the value **32** here would mean that 32 bytes of shared memory should be allocated per block. The 32 bytes can be used for many different purposes—they could be 8 **floats**, 16 **shorts**, 32 **chars**, or any combination of data types that would consume 32 bytes.

The kernel declares the shared array as `extern __shared__`, which means that the amount of shared memory is dynamic and will be determined by the host in the launch configuration. In the previous example, the shared array is of `char` type, but it could be any type at all. Also, the 32 bytes of shared memory per block were specified in the launch configuration with a literal constant, but this amount can be a variable.

## Using Dynamic Shared Memory as Multiple Data

The launch configuration only allows a single extra value to specify the amount of dynamic shared memory per block. If an algorithm requires multiple arrays of shared memory per block, the programmer must allocate a single store of shared memory and use pointers to access it. In other words, if you need multiple dynamic shared memory arrays, they must be coordinated manually. Listing 6.3 shows an example of using a single block of shared memory for two arrays, one of chars and the other of floats.

```
__global__ void SomeKernel(int sizeofCharArray) {  
    // Declare a single dynamic store of shared memory per block  
    extern __shared__ char bothBuffers[];  
  
    // Make a char* pointer to the first element  
    char* firstArray = &bothBuffers[0];  
  
    // Make a float* pointer to some other element  
    float* secondArray = (float*)&bothBuffers[sizeofCharArray];  
  
    firstArray[0]++; // Increment first char  
    secondArray[0]++; // Increment first float  
}
```

*Listing 6.3: Using Two Dynamic Arrays*

In this listing, the same dynamic shared memory allocation (called **bothBuffers**) is used for both a character array (**firstArray**) and a floating point array (called **secondArray**). Obviously, care must be taken not to read and write outside of the bounds of these arrays.

In the listing, the size of the character array is specified and passed as a parameter to the kernel, **sizeofCharArray**. Passing offsets and array sizes as parameters to the kernel allows arrays of variable types and sizes. The **bothBuffers** array used dynamic shared memory, but it could easily be a static allocation, and the **sizeofCharArray** parameter could still be used to manually control the sizes of the arrays.

## CUDA Cache Config

As mentioned previously, the L1 cache of global memory and shared memory are actually the same physical memory. There is 64k (on all current cards) of this memory, and it can be split up by the programmer to use more L1, more shared memory, or the same amount of both. The L1 is a perfectly normal, automatic cache for global memory. Data is stored in the L1 cache and evicted as the device sees necessary. Shared memory is completely in the programmer's control.

When launching a kernel you can split this 64k of memory into 48k of L1 and 16k of shared memory. You might also use 16k of L1 and 48k of shared memory. On newer cards (700 series onwards) there is another option, which is to split the memory in half—32k of L1 and 32k of shared. To set the configuration of shared memory and L1 for a particular kernel, use the **cudaFuncSetCacheConfig** function:



`cudaFuncSetCacheConfig(kernelName, enum cudaFuncCache);`

Where `kernelName` is the kernel and `cudaFuncCache` is one of the values from the `cudaFuncCache` column of Table 6.1.

Table 6.1: `cudaFuncCache` values

<code>cudaFuncCache</code>	Integer value	Configuration
<code>cudaFuncCachePreferNone</code>	0	Default
<code>cudaFuncCachePreferShared</code>	1	48k of shared, 16k of L1
<code>cudaFuncCachePreferL1</code>	2	16k of shared, 48k of L1
<code>cudaFuncCachePreferEqual</code>	3	32k of shared, 32k of L1



**Note:** The `cudaFuncCachePreferEqual` setting is only available on newer cards—700 series and newer. If this setting is selected for an older card, the default value will be used instead.

Whatever value the programmer uses for the `cudaFuncCache` setting, it is only a recommendation to NVCC. If NVCC decides that, despite this setting, the kernel needs more shared memory, then it will override the settings and choose the 48k of shared memory setting. This overriding of the programmer's preference occurs at compile time, not while the program runs.

The following code example demonstrates how to call the `cudaFuncSetCacheConfig` function.

```
// Set the cache config for SomeKernel to 48k of L1
cudaFuncSetCacheConfig(SomeKernel, cudaFuncCachePreferL1);

// Call the kernel
SomeKernel<<<100, 100>>>();
```

Listing 6.4: Setting the Cache Configuration

## Parallel Pitfalls and Race Conditions

This section highlights some dangers of sharing resources between concurrent threads. When resources are shared between multiple threads (this includes any shared resources from global memory, shared memory, texture memory etc.) a potential hazard arises that is not present in serial code. Unless care is taken to properly coordinate access to shared resources, multiple threads may race for a resource at precisely the same time. This results in code that is often unpredictable and buggy. The outcome of multiple threads simultaneously altering the value of a resource is unknown (for practical purposes it is not safe to assume any particular value).

To understand how a race condition causes trouble and why it results in unpredictable outcomes, consider how a computer actually operates on variables in memory. Listing 6.5 illustrates a simple kernel with code that purposely causes two race conditions, one in shared memory, and the other in global memory.

```
#include <iostream>
#include <cuda.h>

using namespace std;

// Kernel with race conditions
__global__ void Racey(int* result) {

    __shared__ int i;

    // Race condition 1, shared memory
    i = threadIdx.x;

    // Race condition 2, global memory
    result[0] = i;
}

int main() {
    int answer = 0;
    int* d_answer;
    cudaMalloc(&d_answer, sizeof(int));

    Racey<<<1024, 1024>>>(d_answer);

    cudaMemcpy(&answer, d_answer, sizeof(int), cudaMemcpyDeviceToHost);

    cout<<"The result was "<<answer<<endl;

    return 0;
}
```

*Listing 6.5: Race Conditions*

The kernel declares a single shared integer called **i**. Each thread attempts to set this shared variable to its own **threadIdx.x**. Each thread in a block has a different **threadIdx.x**. In theory, all 1024 threads of each block would simultaneously set this shared variable to different values. This is only “in theory” because the scheduler is in charge of the order that blocks actually execute, and depending on the availability of resources, the scheduler may or may not execute all 1024 threads at the same time. Regardless of whether the scheduler actually executes the threads simultaneously, or if some threads execute after others, setting the variable **i** to 1024 different values is meaningless. This type of activity is not conducive to productive software development. In practice, the actual resulting **i** will be a single value, but we do not know what.

The next race condition in Listing 6.5 occurs when global memory is set by every thread in the grid concurrently. The parameter `result` is stored in global memory. The line `result[0] = i;` is nonsense. For a start, the value of `i` was set with a race condition in shared memory. In essence, the programmer is no longer in control of these variables. The final value presented by these variables is completely up to the device. The device will schedule threads in some order and actually come up with an answer as shown in Figure 6.2, but it would be very foolish to assume that the device will always return 831.

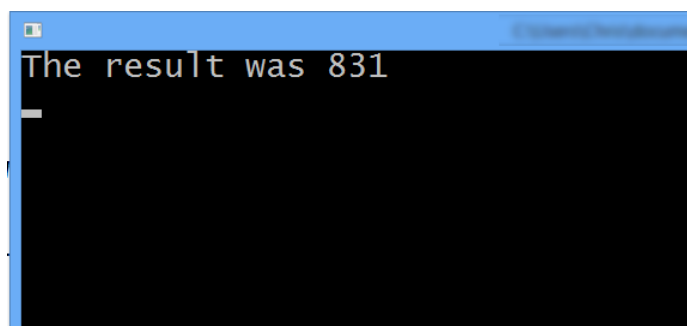


Figure 6.2: Output from Race Conditions

## Read-Modify-Write

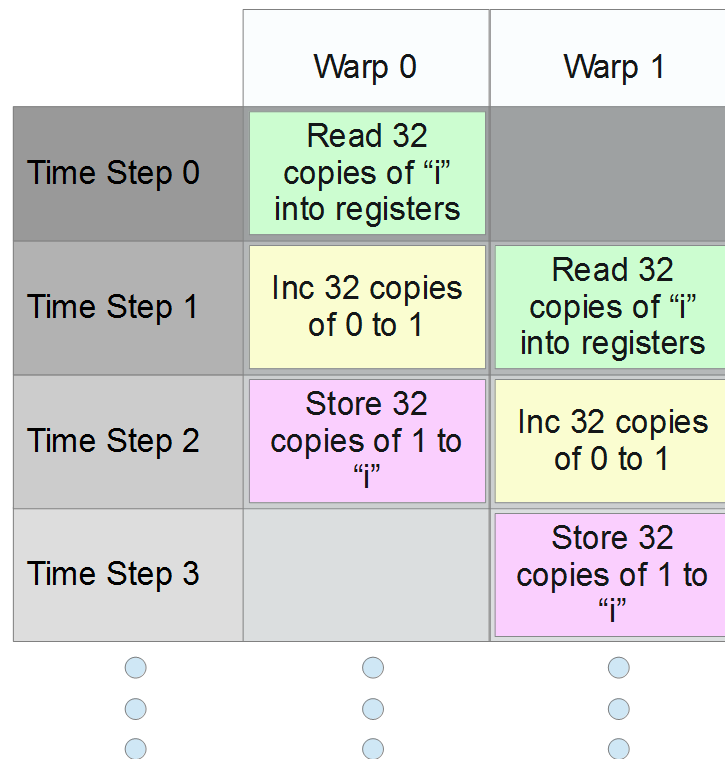
It may seem that if a shared variable was incremented by multiple threads instead of being set to multiple values as in Listing 6.5, the operation would be safe. After all, it is completely irrelevant what order the threads actually increment. So long as they all do increment, we should end up with the same value. The trouble is that this is still a race condition. To see why, we need to look in a little more detail at the increment instruction. The statement `i++` actually does three things (this is applicable to both GPUs and CPUs):

1. Read the value of `i` into a register.
2. Increment the value in the register.
3. Store the result back to `i`.

This process is called a *read-modify-write* operation. Almost all operations which modify memory (both in the host and the device) do so with these three steps. The GPU and indeed the CPU never operate directly on RAM. They are only able to operate on data if it is in a register. This is what the registers are for; they are the variables that a piece of hardware uses for its core calculations. This means the hardware must read the value from RAM first, and write the resulting value when the operation is complete. The trouble is that when more than one thread runs at the same time, they each do the read-modify-write simultaneously, jumbled up in any order, or both. 100 concurrent threads all trying to increment the same shared variable `i` might look like the following:

1. All 100 threads read the value of `i` into registers.
2. All 100 threads increment their register to 1.
3. All 100 threads store 1 as the result.

It is tempting to think that we will always get the value **1**, but apart from being a very pointless use of 100 threads, this is not even true. The 100 threads may or may not actually operate in step with each other. In CUDA, threads are executed in warps of 32 threads at a time (a warp is a collection of 32 threads with sequential `threadIdx` values, all from the same block, and all executing simultaneously). Figure 6.3 illustrates 64 threads running concurrently in two warps.



*Figure 6.3: CUDA warps attempting to increment concurrently*

In Figure 6.3, we see four time steps from 0 to 3 (these might be thought of as individual clock cycles). In the example, threads from the first warp (warp 0) first execute a read. As they increment `i`, threads from the second warp read the original value of `i` (which is still 0). The first 32 threads then write their resulting 1 to memory as the second warp increments. Finally, the second warp writes a 1 to memory as well.

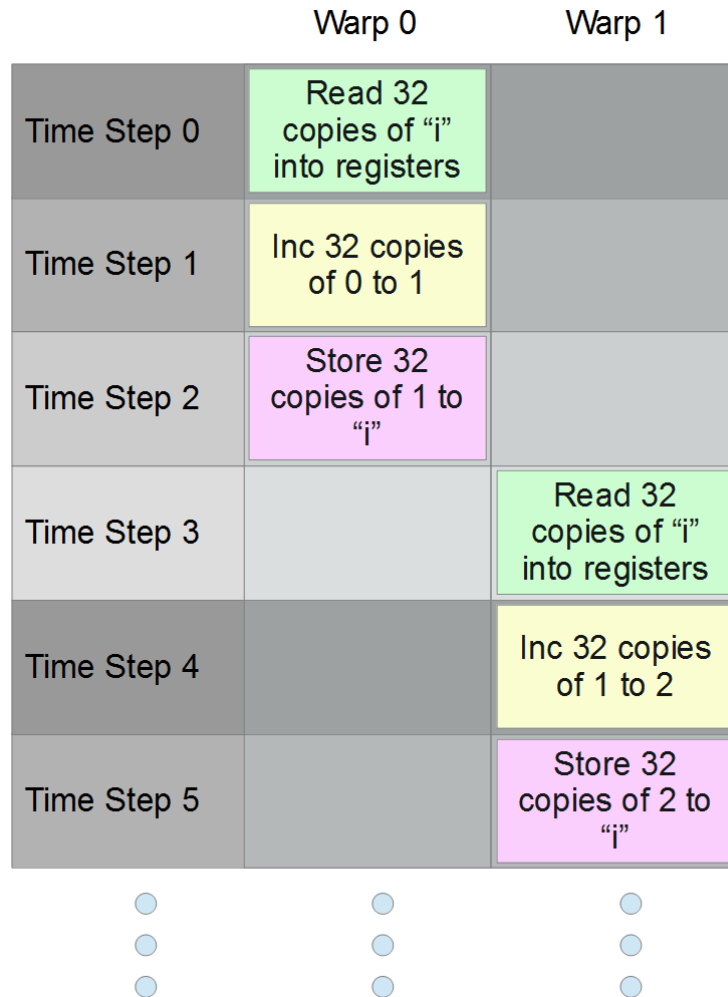


Figure 6.4: Another possible execution order

Figure 6.4 shows another possible way the scheduler might order two warps trying to increment a shared variable. In this example, the first warp completes the read-modify-write and the second warp increments the variable to 2. The scheduler might choose either of these execution orders when scheduling warps and it might choose other possibilities.

There is more than one possible output from the algorithm. Besides, if 32 threads are programmed to increment a variable, the programmer is probably hoping for an end result of 32, which they will almost certainly never get.



**Note:** There are special primitives for dealing with multiple threads when they share resources. Mutexes and semaphores are the most common. For those interested in these parallel primitives, I strongly recommend reading *The Little Book of Semaphores* by Robert Downey.

## Block-Wide Barrier

A “barrier” in parallel programming is a point in the code where all threads must meet before any are able to proceed. It is a simple but extremely useful synchronization technique, which can be used to eliminate race conditions from code. In CUDA there is a block-wide barrier function, `__syncthreads`. The function takes no arguments and has no return value. It ensures that all threads of the block are synchronized at the point of the function call prior to proceeding. Using the `__syncthreads` function, we can ensure that threads do not race for a `__shared__` variable. See Listing 6.6.

```
__global__ void SomeKernel() {
    // Declare shared variable
    __shared__ int i;

    // Set it to 0
    i = 0;

    // Wait until all threads of the block are together
    __syncthreads();

    // Allow one thread access to shared i
    if(threadIdx.x == 0)
        i++;

    // Wait until all threads of the block are together
    __syncthreads();

    // Allow another single thread access to i
    if(threadIdx.x == 1)
        i++;
}
```

Listing 6.6: `__syncthreads()`, the block-wide barrier function

In the previous listing, `__syncthreads` is used to ensure that only one thread at a time increments the shared variable `i`. The initial setting of the variable to 0 by all threads is guaranteed to result in at least one of them successfully setting the value last. The next line contains a call to `__syncthreads`. The threads will all wait until every thread of the block has executed the `i=0` instruction. Once all the threads have paused at the first `__syncthreads` call, only the thread with `threadIdx.x == 0` will fall through the first `if` statement and find itself at another `__syncthreads()`. As they wait, the first thread (`threadIdx.x == 0`) will increment `i` and then join the other threads of the block waiting at the barrier. The threads will then proceed and the thread with `threadIdx.x == 1` will increment `i`. The code shows that `__syncthreads` and single thread access to shared resources is a safe operation, and we are guaranteed that by the end of this code, the shared variable `i` will be incremented to 2.



**Note:** The `__syncthreads()` method is only a block-wide barrier. Threads that belong to other blocks will not be blocked. It is not useful for synchronizing access to global resources. Never use `__syncthreads` in situations where threads of a block branch. If some threads find a `__syncthreads()` in the code of an `if` statement while other threads of the same block fall through the `if` statement, it will produce a deadlock. The waiting

*threads will pause indefinitely, and will never see their brethren again, most likely causing the program to freeze.*



**Note:** There is no safe way to produce a grid-wide barrier inside a kernel. The device is not designed to allow grid-wide barriers from within its own code. However, the host can cause grid-wide barriers. The function `cudaMemcpy` causes an implicit grid-wide barrier. The host will wait until the device has completed any executing kernels before the memory is copied. Also, the host function `cudaDeviceSynchronize()` is designed to explicitly allow the host to wait for the device to finish executing a kernel.

## Atomic Instructions

Aside from barriers, threads can be made to safely access resources by using atomic instructions. An atomic instruction is one that performs the read-modify-write in a single, uninterruptible step. If 32 threads perform an atomic increment concurrently, the variable is guaranteed to be incremented 32 times. See Listing 6.7 for an example of using the `atomicAdd` instruction to increment a global variable.



**Note:** Atomic instructions are only available on devices with compute capability 1.1 or higher. To compile the code in Listing 6.7, you will need to specify `compute_11,sm_11` or higher in the Code Generation option of your project.

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>

using namespace std;

__global__ void AtomicAdd(int* result) {
    // Atomically add 1 to result[0]
    atomicAdd(&result[0], 1);
}

int main() {
    int answer = 0;
    int* d_answer;

    // Allocate data and set to 0
    cudaMalloc(&d_answer, sizeof(int));
    cudaMemcpy(d_answer, 0, sizeof(int));

    // Run 2048 threads
    AtomicAdd<<<64, 32>>>(d_answer);

    // Copy result and print to screen
```

```

cudaMemcpy(&answer, d_answer, sizeof(int), cudaMemcpyDeviceToHost);
cout<<"The result was "<<answer<<endl;

return 0;
}

```

Listing 6.7: Atomically incrementing a global variable

The result from Listing 6.7 (the host's **answer** variable) is always 2048. Every thread increments the global variable **result[0]** atomically, guaranteeing that its increment is complete before any other thread is allowed access to **result[0]**. The kernel was launched with 2048 threads, so the result will be 2048. When a thread begins an atomic read-modify-write of the **result[0]** variable, all other threads will have to wait until it has finished. The threads will each increment **results[0]** one at a time.

Atomic instructions are slow but safe. Many common instructions have atomic versions. For a complete list of the atomic instructions available, see the Atomic Functions section of the CUDA C Programming guide: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions>.



**Note:** Devices with compute capability 2.0 and higher are able to perform atomic instructions on shared memory. Older devices (compute capability 1.xx) had atomic instructions for global memory only.

## Shared Memory Banks and Conflicts

Now that we have examined some of the common problems in sharing resources between concurrent threads, we can turn our attention back to shared memory, and in particular how to use it efficiently. The device performs all operations (adding, multiplication, Boolean operations, etc.) on data in registers. Shared memory must be read into registers before it is operated on (global, texture, constant, and all other memories must also be read into registers first). Once a calculation is performed in the registers, the results can be stored back into shared memory. Shared memory is organized into words of four bytes each. Any four-byte word could hold a single 32-bit **int**, a **float**, half a **double**, two **short** ints, and any other possible combinations of 32 bits. Each word belongs to one of 32 banks, which read and write values to and from the registers.

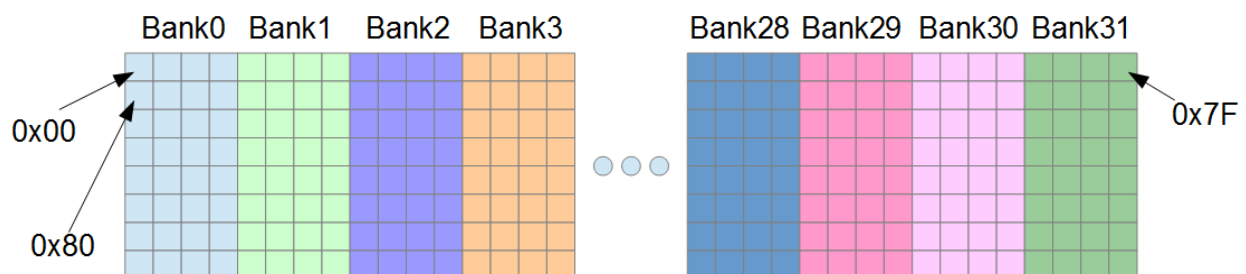


Figure 6.5: Addresses in Shared Memory



Figure 6.5 illustrates some of the addresses in shared memory. Bank0 is responsible for reading and writing the first four bytes of shared memory, and Bank1 reads and writes the next. Bank31 reads and writes bytes with addresses 0x7C to 0x7F, and after this point, the banks repeat. Bytes at addresses 0x80 to 0x83 belong to Bank0 again (they are the second word of Bank0).

Shared memory can be addressed as bytes, but it is also specifically designed to allow very fast addressing of 32-bit words. Imagine that we have an array in shared memory comprised of floats (floats are 32 bits wide or one word each):

```
__shared__ float arr[256];
```

Each successive `float` in the array will belong to a different bank until `arr[32]`, which (like `arr[0]`) belongs to Bank0. Figure 6.5 illustrates the indices of these words and which banks they belong to.

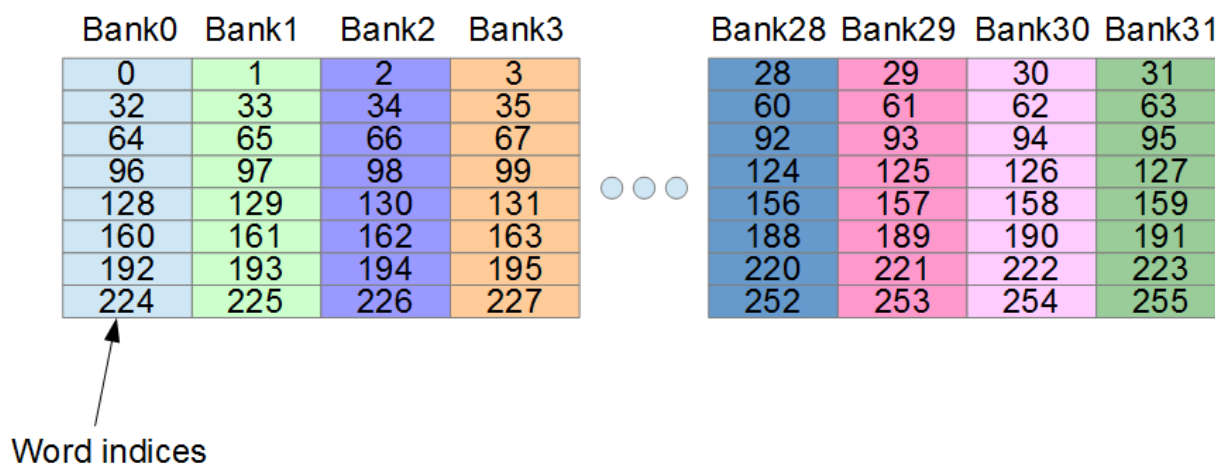


Figure 6.6: Word indices in shared memory

This is important because each bank can only serve a single word to a warp at once. All 32 banks can simultaneously serve all 32 threads of a warp extremely quickly, but only if a single word is requested from each bank. When a warp requests some pattern of addresses from shared memory, the addresses correspond to any permutation whatsoever of the banks. Some permutations are much faster than others. When more than one word is requested from any single bank by the threads of a warp, it is said to cause a bank conflict. The bank will access the words in serial, meaning first one, and then the other.



**Note:** Bank conflicts are only a consideration at the warp level; any inter-block access patterns do not cause bank conflicts. There is no bank conflict if block0's warp accesses word0 from Bank0 at the same time that block1 accesses word1 from Bank0.

When all threads of a warp access exactly the same word from shared memory, an operation called broadcast is performed. Shared memory is read once and the value is divvied out to the threads of a warp. Broadcast is very fast—the same as reading from every bank with no conflicts.

On devices of compute capability 2.0 and up, there is an operation similar to a broadcast (but more flexible) called a multicast. Any time more than one thread of a warp accesses exactly the same word from any particular bank, the bank will read shared memory once and give the value to any threads that require it. The multicast is similar to the broadcast, only all 32 threads of the warp need not access the same word. If there are no bank conflicts, a multicast operates at the same speed as a broadcast operation.

## Access Patterns

In the previous discussion on bank conflicts, broadcasts and multicasts have important implications for performance coding. There are many access patterns the threads of a warp could potentially request from shared memory. Some are much faster than others. All of the following examples are based on an array of words called `arr[]`.

The following table shows a handful of patterns, the speed one might expect from employing them, and a description of conflicts that may be caused.

*Table 6.2: Access Patterns*

Access Pattern	Notes
<code>arr[0]</code>	Fast, this is a broadcast.
<code>arr[blockIdx.x]</code>	Fast, this is a broadcast.
<code>arr[threadIdx.x]</code>	Fast, all threads request from different banks.
<code>arr[threadIdx.x/2]</code>	Fast, this is a multicast. Every 2 <sup>nd</sup> thread reads from the same bank.
<code>arr[threadIdx+71]</code>	Fast, all threads request from different banks.
<code>arr[threadIdx.x*2]</code>	Slow, 2-way bank conflict.
<code>arr[threadIdx.x*3]</code>	Fast, all threads request from different banks.
<code>arr[threadIdx.x*8]</code>	Very slow, 8-way bank conflict.
<code>arr[threadIdx.x*128]</code>	Extremely slow, 32-way bank conflict.
<code>arr[threadIdx.x*129]</code>	Fast, all threads request from different banks.

Accessing multiples of `threadIdx.x` is analogous (produces the same address patterns) to accessing structures in an array. For instance, the following structure is exactly four words long.

```
// 4-word-long structure
struct Point4D {
    float x, y, z, w;
};
```

*Listing 6.8: Structure of four words in length*

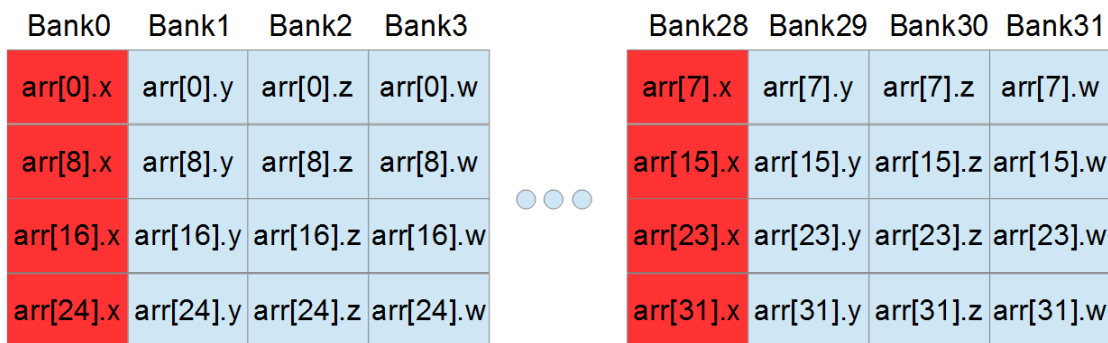
Given an array of instances of this structure in a kernel, and given that the threads are accessing subsequent elements based on their `threadIdx.x`, we will get a four-way bank conflict from the operation in Listing 6.9.

```
__shared__ Point4D arr[32];

arr[threadIdx.x].x++; // Four-way bank conflict
```

*Listing 6.9: Four-Way Bank Conflict*

The increment of the `x` element requires three instructions (read-modify-write) and causes not one, but two four-way bank conflicts. The elements of the array are each four words long. The structures each have their `x` values four banks apart. Every fourth bank is serving the warp four values and the intermediate banks (Banks 1, 2, 3, 5, 6, 7 etc.) are not doing anything. See Figure 6.7.



*Figure 6.7: Four-way bank conflict from structures*

There is a very simple solution to this four-way bank conflict—pad the structure with an extra word, as shown in Listing 6.10.

```
// 4-word-long structure with extra padding

struct Point4D {

float x, y, z, w;

float padding;

};
```

*Listing 6.10: Structure padded to offset banks*

By adding an extra word, the `sizeof(Point4D)` has gone from four words to five, but the access pattern from the threads of a warp no longer causes any bank conflicts (at least not when each thread accesses a subsequent element of the array). With exactly the same code as before (Listing 6.9), we now see that, thanks to the extra padding, there are no bank conflicts at all—every thread is requesting a single word from a different bank.

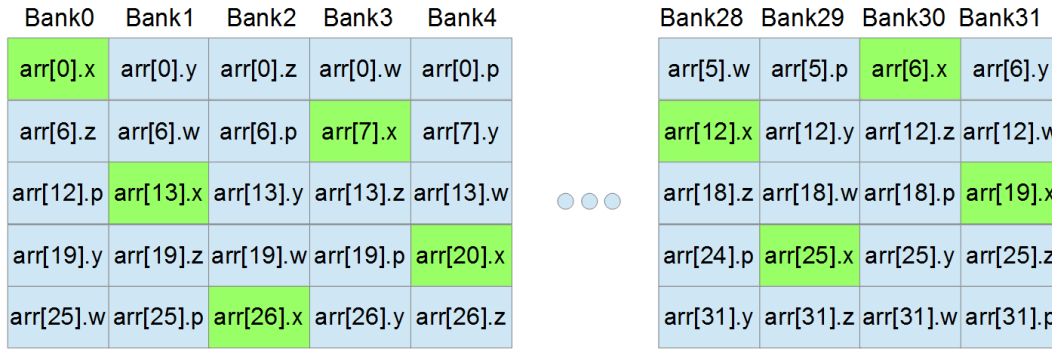


Figure 6.8: Access pattern with padding word

The access pattern in Figure 6.8 looks a lot more complicated than the one from Figure 6.7, but it is much faster because every bank is going to be used exactly once.

Adding padding to offset the requested words and better utilize the 32 banks is often an excellent idea, but not always. It requires enough shared memory to store the extra padding word. The padding word is often completely useless other than to offset the request pattern. There are times when adding padding will be beneficial, and other times when shared memory is too valuable a resource to simply throw away 4 bytes of padding at every thread that wants it.

It is difficult (if not impossible) to entirely invent the best access patterns and padded structures using theory alone. The only way to fine-tune a shared memory access pattern is trial and error. The theory, and even the profiler, can only offer suggestions.

# Chapter 7 Blocking with Shared Memory

In [Chapter 5](#), we examined a simple algorithm for solving the nearest neighbor problem for many points in a list of 3-D points. We followed this in [Chapter 6](#), where we examined shared memory in some detail. We will now apply this new knowledge to greatly increase the efficiency of the program from Chapter 5.

One of the most basic uses of shared memory is known as blocking. Blocking allows more efficient use of the global memory bus, and it also allows the bulk of the calculations to be performed on shared memory and registers instead of relying on the L1 and L2 caches of global memory for speed. It works like this: instead of repeatedly having all threads read and write to and from global memory, we copy blocks of global memory to shared memory. The threads work on the shared memory copy of the data, and when they are done they load another block from global memory. In this way the threads do almost all of their operations with shared memory, minimizing the reads and writes to global memory.



**Tip:** Another side benefit of blocking is that it offers an opportunity for the programmer to change the format of the stored data when it is copied to shared memory. For instance, we could copy 3-D points from global memory and store them as 4-D points in shared memory. This particular operation may or may not have an impact on performance, but it certainly opens up many possibilities.

## Shared Memory Nearest Neighbor

The following version of the nearest neighbor (Listing 7.1) is an optimized version of the one we saw previously in [Chapter 5](#). It uses the blocking technique I just described. The threads of a thread block each copy a single point from global memory to shared memory. They all check if a neighbor near their own point is in the list of points in shared memory. Once they are done, they copy more points from global memory, greatly increasing the speed of the algorithm. The code in Listing 7.1 can replace the code in the `FindClosestGPU.cu` file.

```
// FindClosestGPU.cu
#include <cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>

#include "FindClosestGPU.h"          // Not required!

// Constant block size
__device__ const int blockSize = 128;

// Find nearest neighbor using shared memory blocking
__global__ void FindClosestGPU(float3* points, int* indices, int count) {
```

```

// Shared memory for block
__shared__ float3 OtherBlocksPoints[128];

if(count <= 1) return;

// Calculate a unique idx
int idx = threadIdx.x + blockIdx.x * blockSize;

// Assume the closest if points[-1]
int indexOfClosest = -1;

// This thread's point
float3 thisPoint;

// Read in this block's points
if(idx < count) {
    thisPoint = points[idx];
}

// Assume distance to nearest is float.max
float distanceToClosest = 3.40282e38f;

// Read in blocks of other points
for(int currentBlockOfPoints = 0; currentBlockOfPoints < gridDim.x;
currentBlockOfPoints++) {

    // Read in a block of points to the OtherBlocksPoints array
    if(threadIdx.x + currentBlockOfPoints * blockSize < count)
        OtherBlocksPoints[threadIdx.x] = points[threadIdx.x +
(currentBlockOfPoints * blockSize)];

    // Wait until blocks read from global into shared memory
    __syncthreads();

    if(idx < count) {
        // Use pointer for faster addressing
        float* ptr = &OtherBlocksPoints[0].x;

        // For each point in shared memory block:
        for(int i = 0; i < blockSize; i++) {

            // Calculate distance
            float dist = (thisPoint.x - ptr[0]) * (thisPoint.x - ptr[0]);
            dist += (thisPoint.y - ptr[1])*(thisPoint.y - ptr[1]);
            dist += (thisPoint.z - ptr[2])*(thisPoint.z - ptr[2]);
            ptr+=3;

            // If this point is within the list and nearer than the
            // current closest, update nearest with this one:
            if(dist<distanceToClosest &&(i+currentBlockOfPoints*blockSize)
< count && (i+currentBlockOfPoints*blockSize) != idx) {
                distanceToClosest = dist;
                indexOfClosest = (i + currentBlockOfPoints * blockSize);
            }
        }
    }
}

```

```

        }
    }

    __syncthreads();
}
if(idx < count)
    indices[idx] = indexOfClosest;
}

```

*Listing 7.1: Nearest neighbor with shared memory*

As you can see, the kernel in Listing 7.1 is much faster than the previous version from Chapter 5. This version requires that the grid be launched with blocks of 128 threads. At the top of the listing I have used a constant `int` called `blockSize`; this will become a literal in the compiled code. Literals are usually faster than variables, even automatic variables like `blockDim.x`.

Each thread calculates a unique `idx` value and reads the corresponding point from the list of points. Each thread will find the nearest neighbor to its own point. This is identical to the operation the previous version performs in Chapter 5.

Next we have a `for` loop, which counts up using the variable `currentBlockOfPoints`. Each time this loop iterates, the threads read a collection of points from global memory into the shared memory belonging to their block. Note how I use `__syncthreads` after reading the points from global memory into the shared memory block, ensuring that no threads begin checking the distances to the points before the entire block is copied to shared memory.

Once the points are in shared memory, threads iterate through the data and see if any are the nearest neighbors so far. Once the entire global memory array has passed through shared memory in these blocks, the calculation is complete. Note also there is another `__syncthreads` call after the `for` loop. This ensures that all the threads have finished checking the points in the shared memory block prior to reading in a new block.

I have manually used a pointer to address the values of the shared memory points (this is in the variable called `ptr`). This is often (but possibly not always) faster than using a structure addressing syntax. It might also be a good idea when performing complicated calculations to use small steps (like the calculation of the `dist` variable in Listing 7.1). This is more efficient for a compiler to optimize, rather than placing an entire calculation on a single, long line of code. Of course, like any code, it's always worth trying to see if something is effective.

This version of the algorithm (Listing 7.1) runs at three times the speed of the original CUDA version presented in Chapter 5, and at 14 times the speed of the original CPU implementation (these times will differ depending on the performance ratio between your particular GPU and CPU). It is very possible that this code could be optimized further, but for the current illustration, it is fast enough.

# Chapter 8 NVIDIA Visual Profiler (NVVP)

We will now shift our attention to something completely different. We will step away from straight CUDA C coding and look at some of the other tools, which collectively form the CUDA toolkit. One of the most powerful tools that comes with the CUDA toolkit is the NVIDIA Visual Profiler, or NVVP. It offers a huge collection of profiling tools specifically aimed at drilling down into kernels and gathering hundreds of interesting performance counters.

Using a profiler can greatly help code optimization and save programmers a lot of time and trial and error. NVVP can pinpoint many possible areas where the device could potentially be used more efficiently. It runs the application, collecting thousands of metrics from the device while the kernel executes. It presents the information in an effective and easy-to-understand manner.

Older devices with compute capability 1.x do not collect the same metrics as newer devices. If you are using an older device, you may find that the profiler has insufficient data for some of the following tests. The profiler will not generate the charts depicted in the figures when there is no data.

## Starting the Profiler

The visual profiler will be installed to any one of several different locations, depending on the version of the CUDA Toolkit you have installed. The simplest way to run the profiler is to search your installed apps for NVVP. You can search for NVVP using the Start menu for Windows 7 and previous versions, or the Start screen in Windows 8.

When you run the NVVP application, you will see a brief splash screen. Then, you should see the main profiler window. A wizard may open and assist you in creating the project and loading a program to profile.

To open a program for profiling, open the **File** menu and choose **New Session**. You will see the Create New Session dialog. I will use the shared memory version of the nearest neighbor algorithm (from Chapter 7) for testing out the profiler in the following screenshots, but you can load any .exe file here. Type a path to an .exe file you have compiled or find the compiled file you wish to profile by clicking the **Browse** button (see Figure 8.1). It is usually best to profile applications which are compiled with optimizations (i.e. release mode rather than debug). The performance difference between release mode and debug is vast, and the profiler's information on a debug mode program is probably not relevant at all to the release mode compilation.



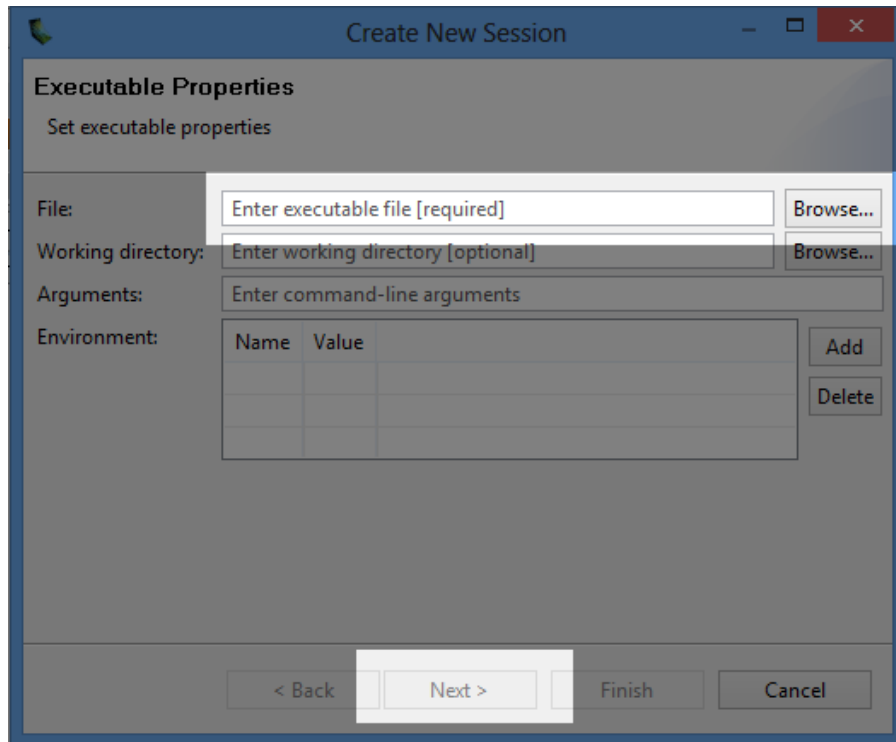


Figure 8.1: Create New Profiler Session

You can select the working directory and command line arguments to pass the program when it executes, as well as environment variables, if needed. The only setting that is required in the Create New Session dialog is the File setting. Once you have chosen the file, click **Next**.

The next page of the dialog allows you to specify some basic profiling settings, like whether the application should be profiled straight away (or if the profiler should simply locate the file and await your command to begin the profiling) and what the device's driver timeout should be. Click **Finish** to create the profiler project. If you left the **Start execution with profiling** check box selected, the program will run and profile your application straight away.



**Tip:** Use the *Generate Line Number Info* option in Visual Studio to get the most out of the visual profiler. The profiler is able to provide detailed information when the compiled program includes line number references. This option can be set to Yes (-lineinfo) in the Project Settings dialog in Visual Studio, in the CUDA C/C++ section.

## General Layout

The profiler consists of a menu bar and a toolbar at the top of the window, and three subwindows, each containing one or more tabs (see Figure 8.2). The top-left window provides an overview of the kernels and other CUDA API function calls. It consists of a collapsible tree view of devices (including the host), streams, and contexts that are active in the application.

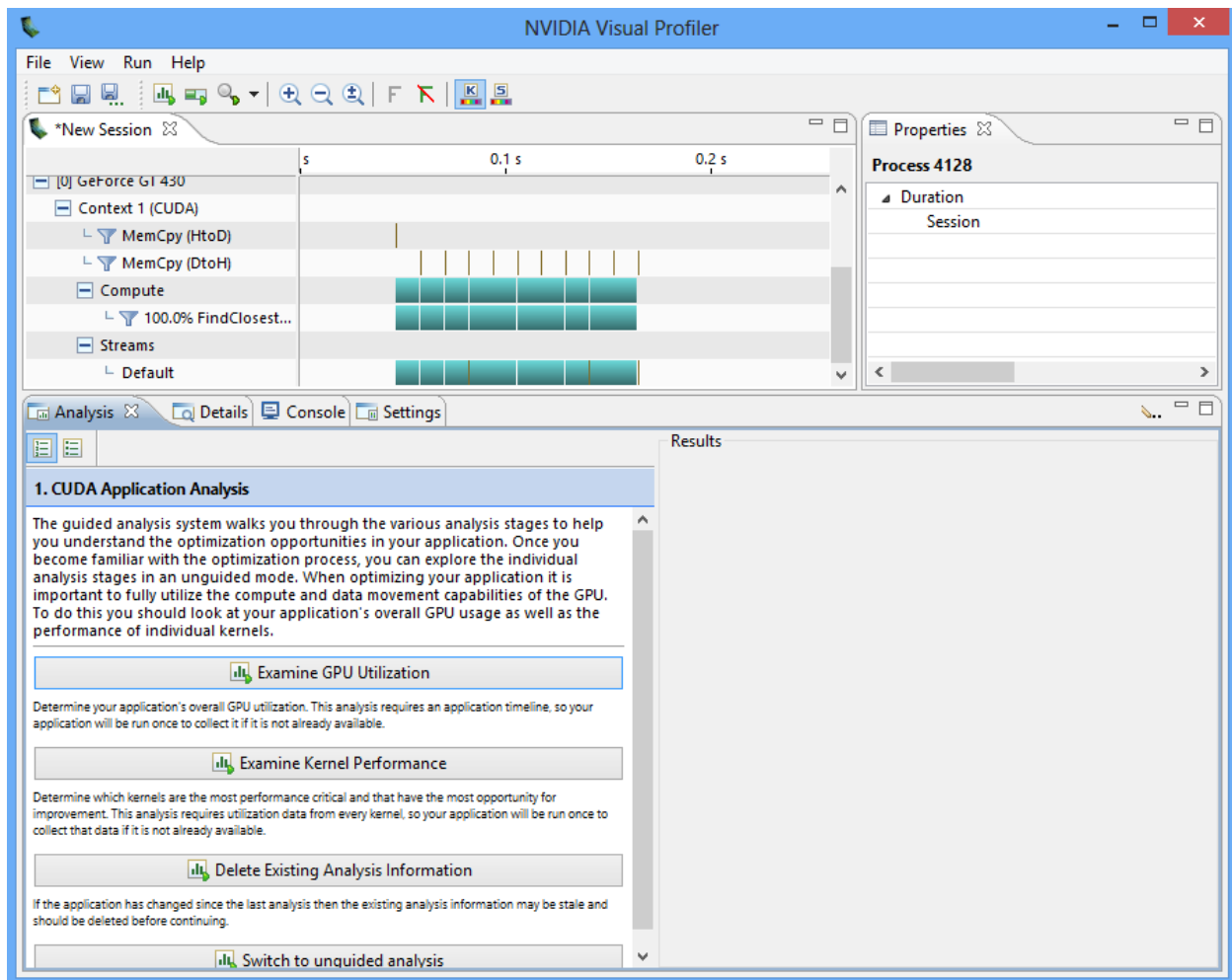


Figure 8.2: Main Window of NVVP

This overview panel also contains a chart to the left of the tree view. The rows of the chart can be opened or closed using the tree view. The chart depicts the amount of time taken for kernels to execute and the CUDA API function calls. The longer a function or kernel takes, the longer the bar representing the call will be.

If you choose not to profile the application straight away or need to re-run the profile after making changes to the .exe, you can click the **Play** button in the toolbar (the one with a loading bar behind it) to re-run the profile at any time.

Once the application has been profiled, select the bar that represents the kernel call you are interested in. I have selected the first call to **FindClosestGPU** in Figure 8.3.

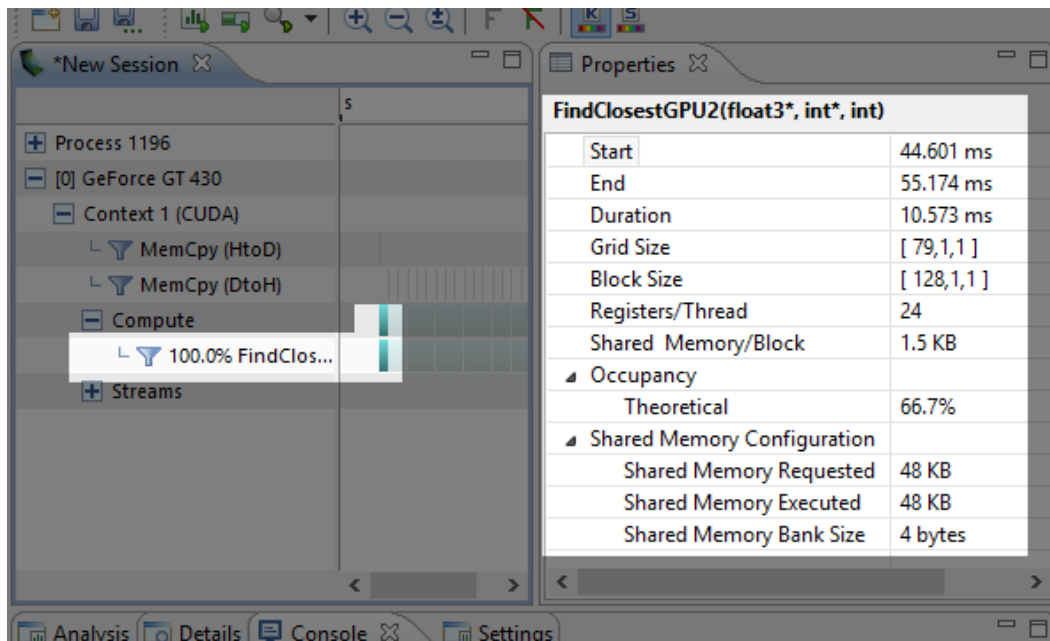


Figure 8.3: Properties of a Kernel Call

The Properties panel will be filled with information about the kernel call (right side of Figure 8.3). The start, end, and total execution time is displayed in milliseconds. The launch configuration shows that the grid consisted of a total of 79 blocks, each with 128 threads.

There are metrics for register counts per thread and shared memory per block. We will examine these in more detail shortly because they greatly influence the occupancy metric. The next number is theoretical occupancy. We will also examine occupancy in detail as we progress through this chapter, because of the generated achieved occupancy metric, which tells us the actual occupancy a kernel achieved, not just the highest theoretical occupancy.

The final three properties show the amount of shared memory the SMs requested, how much they were given, and the word size of the banks of shared memory (which is 4 for all current cards).

## Occupancy

Occupancy is a simple metric for determining how busy the device is compared to how busy it could be. Occupancy is the ratio of the number of concurrently running warps divided by the maximum number of warps that could be running with the particular device.

$$\text{Occupancy} = \frac{\text{Concurrently Executing Warps}}{\text{Maximum Possible Concurrent Warps}}$$

Concurrent warps may or may not actually run at the same time. A warp is said to be active (concurrently running) if the device has allocated the resources (shared memory and registers) for the warp. The device will keep a warp active until it has finished executing the kernel. It will switch back and forth between active warps in an attempt to hide the latency of memory and arithmetic caused by the warps as they execute instructions.

Usually, the more warps running concurrently (the higher the occupancy), the better the utilization of the device, and the higher the performance. If there are many warps running concurrently, when one is delayed, the scheduler can switch to some other warp and thus hide this delay or latency.



**Tip:** The CUDA Toolkit supplies an Excel worksheet called the **CUDA Occupancy Calculator**. This can be found at `C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v6.5\tools` or a similar folder, depending on your installation. The worksheet allows you to quickly input the compute capability, shared memory usage, register counts per thread, and block size. It calculates the theoretical occupancy and shows detailed charts depicting the possible outcome of varying the block size, the register counts, the shared memory usage, or all three. Figure 8.4 shows screenshot of the calculator.

## CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click): **3.5** (Help)

1b) Select Shared Memory Size Config (bytes): **49152** (Help)

2.) Enter your resource usage:

Threads Per Block: **256** (Help)

Registers Per Thread: **32** (Help)

Shared Memory Per Block (bytes): **4096** (Help)

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs: (Help)

Active Threads per Multiprocessor: **2048**

Active Warps per Multiprocessor: **64**

Active Thread Blocks per Multiprocessor: **8**

Occupancy of each Multiprocessor: **100%**

Physical Limits for GPU Compute Capability: **3.5**

Threads per Warp: **32**

Max Warps per Multiprocessor: **64**

Max Thread Blocks per Multiprocessor: **16**

Max Threads per Multiprocessor: **2048**

Maximum Thread Block Size: **1024**

Registers per Multiprocessor: **65536**

Max Registers per Thread: **256**

Shared Memory per Multiprocessor (bytes): **49152**

Max Shared Memory per Block: **49152**

Register allocation unit size: **256**

Register allocation granularity: **warp**

Shared Memory allocation unit size: **256**

Warp allocation granularity: **4**

Allocated Resources

Per Block Limit Per SM = Allocatable

Warps (Threads Per Block / Threads Per Warp): **8** **64** **8**

Registers (Warp limit per SM due to per-warp reg count): **8** **64** **8**

Shared Memory (Bytes): **4096** **49152** **16**

Maximum Thread Blocks Per Multiprocessor: **8** **8** **64**

Limited by Max Warps or Max Blocks per Multiprocessor: **8** **8** **64**

Limited by Registers per Multiprocessor: **8** **8** **64**

Limited by Shared Memory per Multiprocessor: **8** **8** **64**

Physical Max Warps/SM = **64**

Occupancy = **64 / 64 = 100%**

CUDA Occupancy Calculator

Version: **6.0**

Copyright and License

Here for detailed instructions on how to use this occupancy calculator, or more information on NVIDIA CUDA, visit <http://developer.nvidia.com/cu>

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

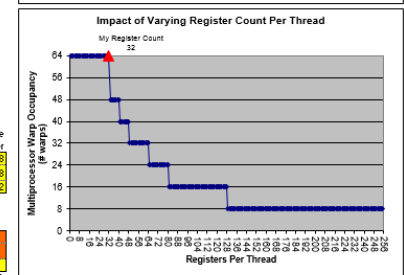
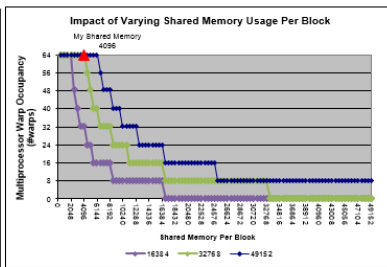
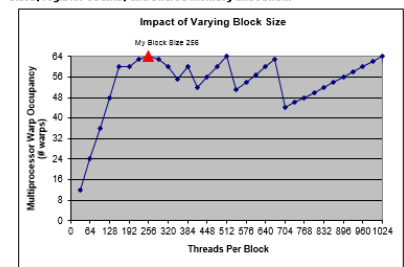


Figure 8.4: CUDA GPU Occupancy Calculator

The maximum number of threads per SM (according to the output Device Query for my device; be sure to check the output for your own hardware) is 1,536 (or  $48 \times 32$ ). This means for 100 percent occupancy, the kernel would need to have a total of 1,536 threads active at any one time; anything less than this is not 100 percent occupancy. There is most likely more than one SM in the device (again, according to Device Query for my hardware there are two), and the total number of active threads is 1,536 multiplied by the number of SMs in the device. For instance, if there are two SMs, the total number of active threads could potentially be 3,072.

This maximum occupancy can only be achieved with very special care. The number of threads per block, registers per thread, and shared memory per block work together to define the number of threads the SM is able to execute concurrently.



**Note:** *Maximizing occupancy is not always the best option for optimizing a particular kernel, but it usually helps. For a very enlightening presentation on optimizing certain kernels with low occupancy, see Vasily Volkov's 2010 GPU Technology Conference (GTC) presentation, *Better Performance at Lower Occupancy*. The slides are available at <http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf>. The presentation was recorded and is available online from GTC on demand at <http://www.gputechconf.com/gtcnew/on-demand-gtc.php>.*

## Register Count

Each SM in the device is able to execute up to eight blocks concurrently, and up to 48 warps, but only if resources permit. One of the governing resources is the register count per thread. The Register/Thread metric in Figure 8.3 shows how many registers are used per thread. The number of registers used is related to the number of variables (and the variable types—for instance, doubles take two registers each) in the kernel's code. Reusing a small collection of variables in the code rather than defining many different variables will generally decrease the number of registers per thread, but usually at the expense of code complexity.

If this value is high (perhaps 40 or 60 registers per thread) and there are a considerable number of threads per block, the SMs will not be able to execute 48 warps concurrently.



**Tip:** *NVCC can return verbose information. This output includes, among other things, a printout of the register usage per kernel to Visual Studio's output window during the project build. If you wish to enable this output, set the Verbose PTXAS Output option to Yes (`--ptxas-options=-v`) in your project's properties in the CUDA/Device section.*



**Note:** *There is a single additional, implicit register used per thread, which is not reported in the profiler or with the verbose PTXAS output. All register counts should have this additional register added to be accurate. Instead of a maximum of 64 registers per kernel, we are limited to 63.*

The maximum number of concurrent threads per SM for this device (GT 430 according to Device Query) is 1,536 (48 warps of 32 threads each). The maximum number of registers (reported by Device Query) is 32,768, or 32k. If each thread uses 24 registers (plus a single implicit register), then 1,536 threads would use  $25 \times 1,536$ ; or 38,400 registers. This number is greater than the 32,768 available registers, so the device cannot possibly perform at 100 percent occupancy. Each warp would require  $32 \times 25$  (since there are 32 threads per warp) registers, or 800. The number of available registers divided by this value and rounded down will yield the approximate number of active warps an SM can execute,  $32,768 / 800 = 40$ . This means the number of active warps the device could execute if it had only a single SM would be 40 instead of the maximum of 48.



**Tip:** *There is a very simple way to reduce the register usage of threads—set the `MaxRegCount` value in the project's properties. In the CUDA C/C++ tab under the Device subheading, one of the options is labeled `Max Used Register`. Set a small number here if your kernel's register usage is limiting occupancy. Be careful! Do not set this to an impractically low value (such as 1 or 2); your code may take a very long time to compile or not compile at all. A setting of 0 (default) means NVCC will decide how many registers to use. Setting values lower than the number NVCC would use will generally mean that local memory is used when registers spill, and may not give you any extra benefit in increasing performance.*

## Shared Memory Usage

The total amount of shared memory on the device is 64k, of which some portion will be used as L1 cache. The kernel in this example used 16k of L1 cache and 48k of shared memory. Each block used an array of `float3` structures, which has a `sizeof` value of 12 bytes each. There were 128 elements in the array, so the total shared memory usage per block for this kernel is  $128 \times 3 \times 4$  bytes, which is 1.5 kilobytes.

The SM will execute up to eight blocks at once if resources permit. Since  $48k / 1.5k = 32$ , our shared memory usage would actually permit 32 simultaneous blocks. This means that shared memory is not a limiting factor on the occupancy for this kernel.

## Unguided Analysis

We can get a lot more detail from the profiler by selecting a particular instance of the kernel in the graphical display window and using the unguided analysis option. In the following screenshot (Figure 8.5), I have selected the first kernel launch.

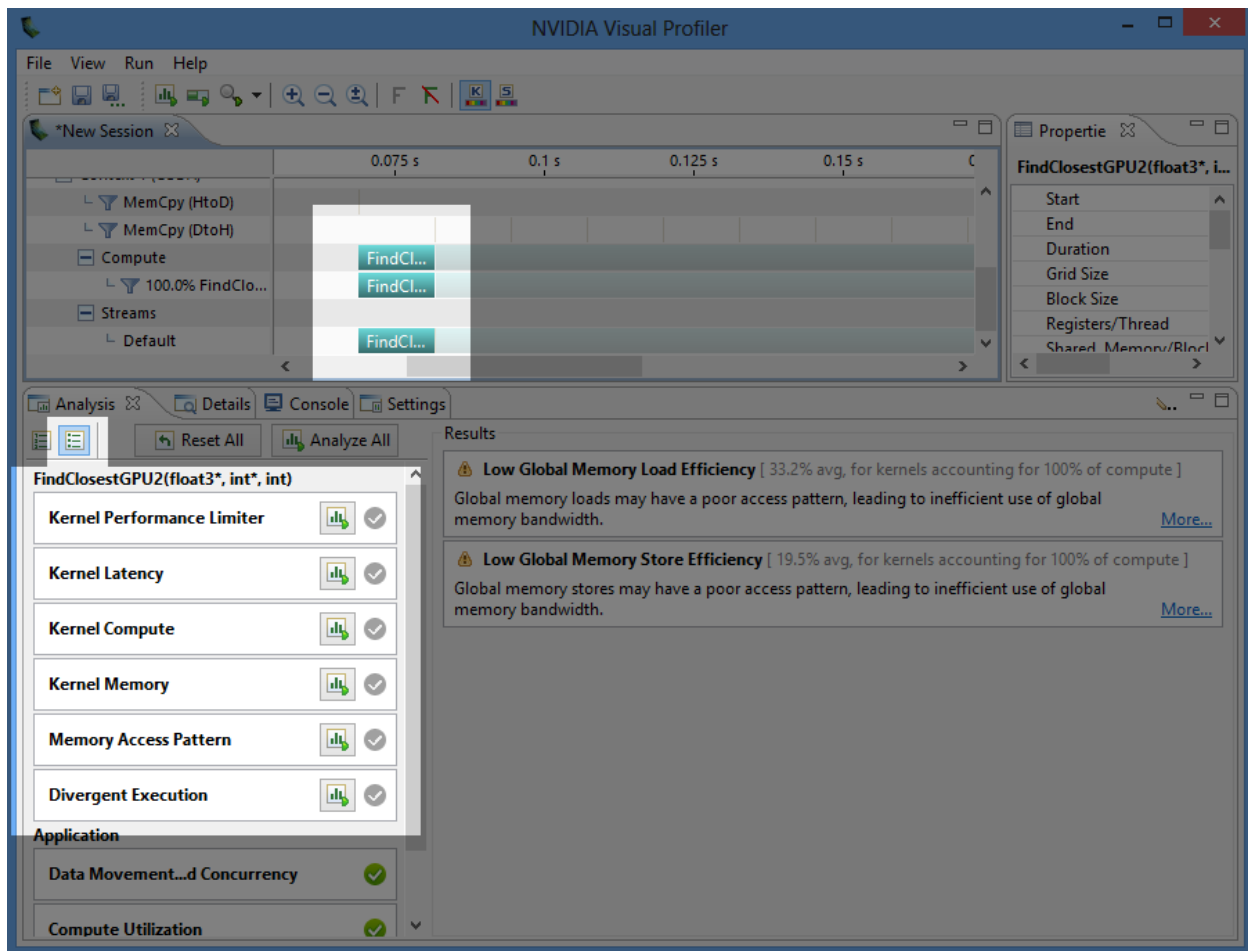


Figure 8.5: Unguided Analysis

Clicking the Run Analysis buttons (these are the buttons with a small chart and green arrow in the Analysis tab of the left panel) will re-run the application in the profiler, but it will collect the specifically requested data on the currently selected kernel. The selected kernel will usually take much longer to run because the profiler collects information as it executes the code.

## Kernel Performance Limiter

The information from an unguided analysis can help you pinpoint exactly where the bottlenecks in your code are. Clicking on the Kernel Performance Limiter button in this example produces the following chart (Figure 8.6).

## Results

### i Kernel Performance Is Bound By Compute And Memory Bandwidth

For device "GeForce GT 430" compute and memory utilization are balanced. These utilization levels indicate that kernel performance is good, but that additional performance improvement may be possible if either of both of compute and memory utilization levels are increased.

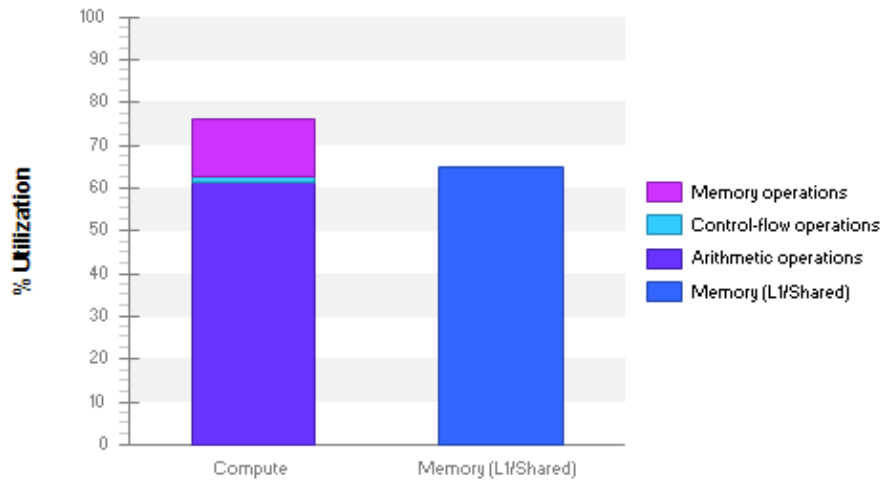


Figure 8.6: Kernel Performance Limiter

Control flow and memory operations are only a small portion of the Compute bar in Figure 8.6. The Memory bar shows that we are relying heavily on shared memory and not using the L1 cache at all. This heavy use of shared memory was intentional; it is the result of the blocking technique we applied in Chapter 7.

The arithmetic operations portion of the Compute bar is glaringly large. The chart indicates that if the kernel is to execute much faster, it will require much less arithmetic. To decrease the amount of arithmetic in a kernel, we would need to either simplify the expressions (which will probably not offer a substantial gain since the expressions were not complicated), or we can use a more efficient algorithm.

Our algorithm, which is essentially just a brute force linear search, was a poor choice. Each point is compared to every other point in the entire array, and if we really wanted to reduce the amount of arithmetic, we would investigate storing our points in a more sophisticated structure where we wouldn't need to perform so many comparisons. The profiler has not directly said this, but it can be gleaned from Figure 8.6. Our kernel requires too many point comparisons, and that is the main reason it is slow.

## Kernel Latency

The Kernel Latency option produces a sophisticated output of charts that indicate what might be limiting the occupancy of the kernel (see Figure 8.7).



## Results

### ⚠ GPU Utilization May Be Limited By Block Size

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

The kernel has a block size of 128 threads. This block size is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GT 430" can simultaneously execute up to 8 blocks on each SM. Because each block uses 4 warps to execute the block's 128 threads, the kernel is using only 32 warps on each SM. Chart "Varying Block Size" below shows how changing the block size will change the number of warps that can execute on each SM.

Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM.

[More...](#)



Figure 8.7: Kernel Latency

This option also produces the achieved occupancy statistic in the Properties tab, just above the theoretical occupancy. The output in Figure 8.7 compares the achieved occupancy with the maximum. It indicates quite clearly that the block size (128 threads) is a possible limiting factor.

For this particular kernel and this particular device, this tip from the profiler makes little to no difference on overall performance. I tested many block sizes and found that although larger block sizes increase the occupancy (for instance, 512 threads per block achieves an occupancy of 92.2 percent), the overall performance remains the same. The suggestion from the profiler to increase the block size is, in this instance, not beneficial, but this will often not be the case.

## Kernel Compute

When we use the Kernel Compute option, we can see again that arithmetic is probably a bottleneck in our code. The chart in Figure 8.8 (the output from the Kernel Compute analysis) also shows that our texture compute usage is non-existent. Storing points as textures and allowing the texture units to perform at least some of the computation might improve our performance.

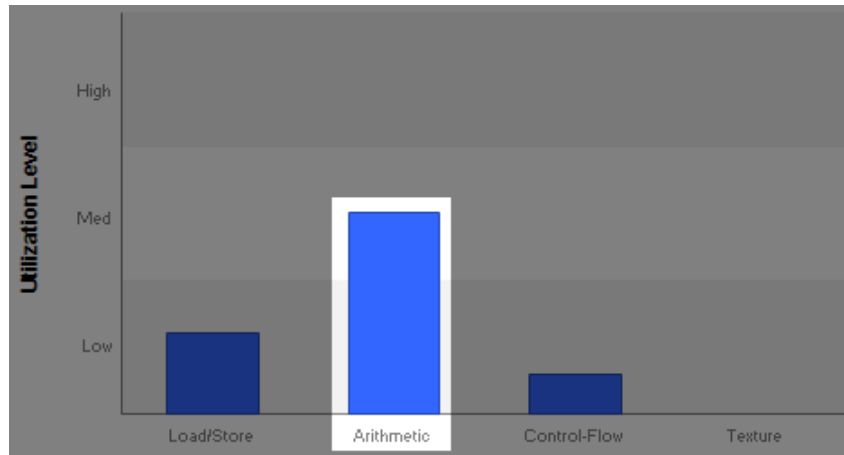


Figure 8.8: Kernel Compute

## Kernel Memory

The Kernel Memory usage profile shows that our shared memory usage is unsurprisingly quite high from the blocking technique we employed in Chapter 7 (Figure 8.9). The texture cache reads are very low since we did not use any texture memory at all. This output again suggests that there may be some benefit gained if we can figure out a way to use texture memory for our current problem.

Results			
i Memory Bandwidth And Utilization			
The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. <a href="#">More...</a>			
	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	1241248	11.62 GB/s	
Local Stores	869004	8.19 GB/s	
Shared Loads	9464832	89.82 GB/s	
Shared Stores	75120	725.37 MB/s	
Global Loads	222300	2.14 GB/s	
Global Stores	312	15.27 MB/s	
L1/Shared Total	11872816	112.5 GB/s	<div><div></div></div>
Texture Cache			
Reads	0	0 B/s	<div><div></div></div>
L2 Cache			
Reads	325339	858.53 MB/s	
Writes	30898	74.8 MB/s	
Total	356237	933.33 MB/s	<div><div></div></div>
Device Memory			
Reads	246266	606.3 MB/s	
Writes	30434	73.54 MB/s	
Total	276700	679.84 MB/s	<div><div></div></div>

Figure 8.9: Kernel memory with shared memory blocking

The “Idle” readings for L2 and device memory are often a good thing since these memories are very slow compared to shared memory. Figure 8.10 shows the Kernel Memory usage profile for the original CUDA version of the algorithm (where we did not use shared memory). I have included it as it makes for an interesting comparison.

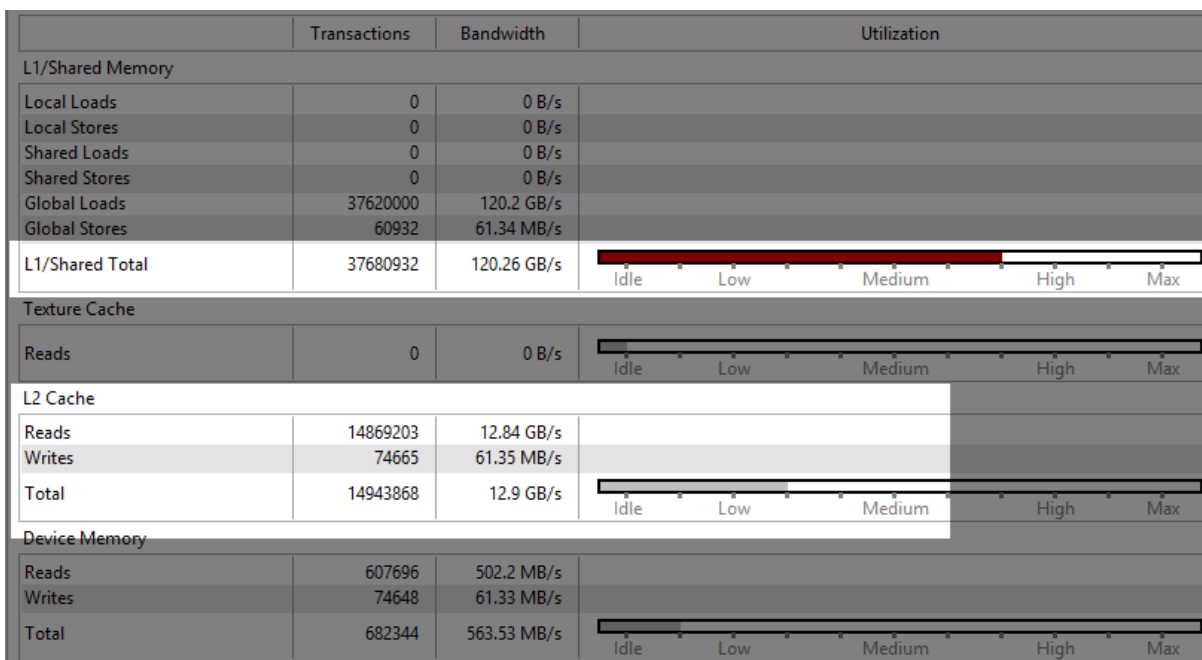


Figure 8.10: Kernel memory without shared memory blocking

The difference in performance (from 37 milliseconds in our Chapter 5 algorithm to 12 milliseconds in our Chapter 7 algorithm) is almost entirely attributed to the L2 cache being employed in the original code. The original version of the code (which leads to Figure 8.9) used L1 cache extensively. When the original kernel’s code executed, points were evicted from L1 to L2 when they were actually going to be read again in the very near future. The difference between the use of L2 and shared memory blocking gave us the healthy 300 percent performance gain.

Data copied to the device using `cudaMemcpy` always goes through the L2. In Figure 8.8, the L2 reads are showing that the points were read from L2 into our shared memory blocks.

## Memory Access Pattern

To perform memory access pattern unguided analysis, you must generate line numbers when you compile your application. This option will examine the efficiency of the usage of global memory. Figure 8.11 is an example output.

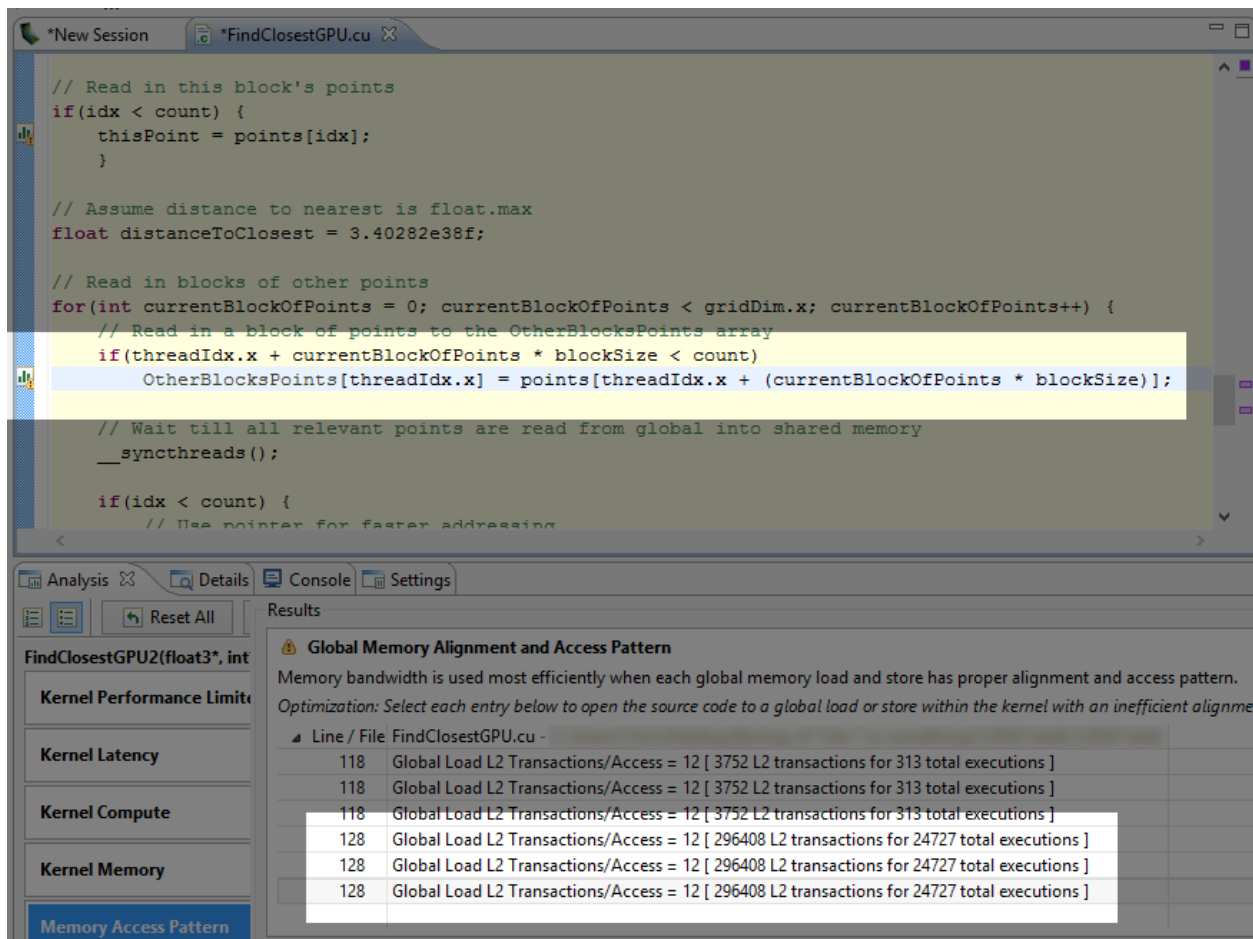


Figure 8.11: Memory Access Pattern

The top three lines (those referencing line 118 in the code) are potentially not as important as the lower three. Line 118 is where we read a single point from global memory into each thread. There are three transactions here because the structure (**float3**) is made of three floats. The following three transactions are potentially much more important to us (they reference line 128 in the code where we read global memory into blocks of shared memory). It is clear from Figure 8.11 and knowing how our blocking worked that the transaction counts at line 118 (3,752) are completely insignificant compared to the line 128 transactions (296,408) seen further down. If we needed to optimize based on this output, we should start by looking at line 128 rather than line 118.

In this particular code, you can see that this point is not making any difference whatsoever on the actual performance of the kernel. If we comment out line 128 in the code and re-run the kernel, it achieves exactly the same performance.

In the original CUDA version of the algorithm (from Chapter 5), the inner loop was causing 37,520,000 transactions (not shown in the screenshot). In the new version (from Chapter 7), the global memory transactions of the inner loop are completely negligible and not shown by the profiler at all in Figure 8.10. What we are looking at instead is the comparatively tiny number of transactions involved in creating the shared memory blocks. The 296,408 transactions from reading the blocks into shared memory are a vast improvement over 37,520,000.

## Divergent Execution

The final option (second to last if you are using CUDA version 6.0 or higher) in the unguided analysis is the divergent execution analysis. In our particular kernel the divergence of threads within a warp is irrelevant and the threads of each warp act in lockstep almost all of the time. This type of execution pattern is ideal for CUDA and it is possible here because our algorithm was embarrassingly parallel. There is almost no communication required between threads whatsoever (except for the occasional block-wide barriers `__syncthreads()`).

Thread divergence is a tricky problem to solve. A good way to think about solving poor performance from thread divergence is to alter the data, the algorithm, or both instead of trying to optimize small sections and maintaining the same algorithm, data structure, or both. Minimize thread communication and synchronization. Organize data such that large groups of consecutive threads (i.e. 32 threads of a warp) will most likely take the same branches in an `if` statement.



**Tip:** When questioning how data might be better be stored and accessed, consider *Array of Structures versus Structure of Arrays*. Known as “AoS vs. SoA,” this concept involves a perpendicular rotation of data in memory to improve access patterns. For example, instead of storing 3-D points as  $[x_1, y_1, z_1, x_2, y_2, z_2, \dots]$ , the data is stored with all the  $x$  values first, followed by the  $y$  and then the  $z$ , e.g.,  $[x_1, x_2, x_3, \dots, y_1, y_2, y_3, \dots, z_1, z_2, z_3, \dots]$ . In other words, instead of storing data as an array of points (AoS), it might help access patterns to store three arrays—one for the  $X$  values, one for the  $Y$  values, and one for the  $Z$  values (SoA).



**Tip:** CUDA can be used to perform one algorithm quickly, but it can also be used to perform the same algorithm multiple times. This seemingly obvious statement is always worth considering when deciding what should be converted to CUDA. For instance, in a string search, we could use CUDA to search quickly for a particular sequence of characters. But a completely different approach would be to use CUDA to simultaneously perform many searches.

## Other Tools

### Details Tab

The Details tab gives a summary of all kernel launches. This information is essentially the same as the information in the Properties window, but it is given for all launches at once in a spreadsheet-like format.



**Tip:** If you have a kernel selected in the graphical view, the details will be displayed for that kernel only. To show all details, right-click in the graphical view and select *Don't Focus Timeline* from the context menu.

## Metrics and Events

Finally, from the **Run** option in the menu bar, you can select to collect or configure metrics and events. This will open a window that allows you to specify the exact statistics you are interested in collecting (see Figure 8.12).

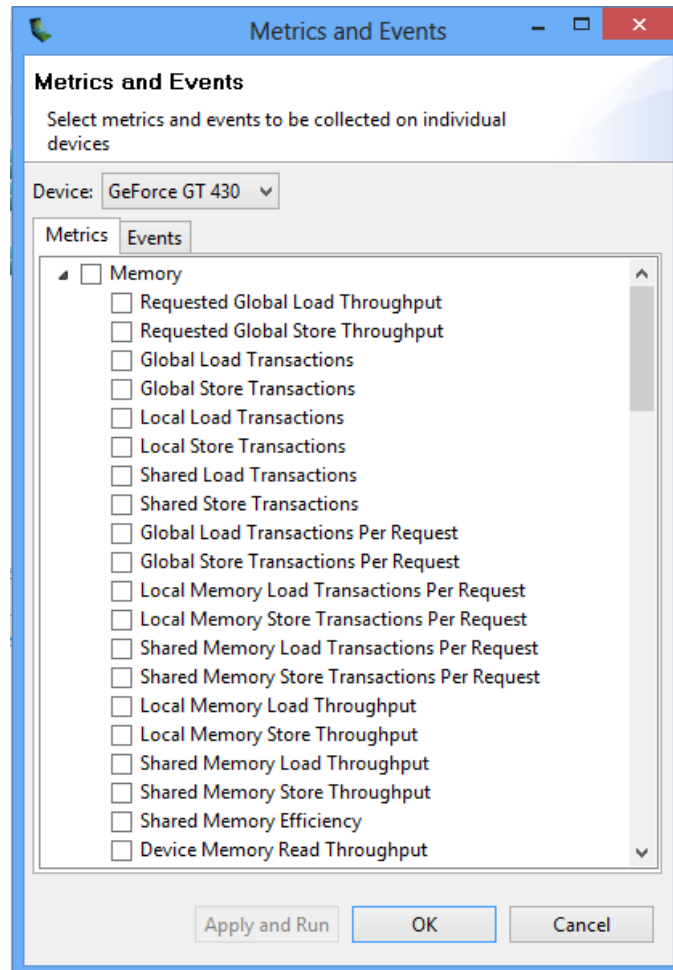


Figure 8.12: Metrics and Events

The benefit of specifically defining the metrics you are interested in is that when you click **Apply and Run**, the metrics will be collected for all the kernels. In the previous sections, the data was collected for the selected kernel alone. The output metrics are not shown in the Properties window but the Details panel.

In addition to this, you can select many metrics at once by selecting the root nodes in the tree view (Figure 8.12). This allows you to quickly select all metrics and run a gigantic analysis, collecting information on every aspect of every kernel in an application. It takes some time to complete the analysis, depending on the kernel. The output from such a request is a proverbial gold mine for studying possible optimizations. The entire Details panel can be exported to a comma separated values (CSV) file and opened in a standard spreadsheet application. To export the Details panel, click the **Export Details in CSV Format** button in the top right of the panel (see Figure 8.13).

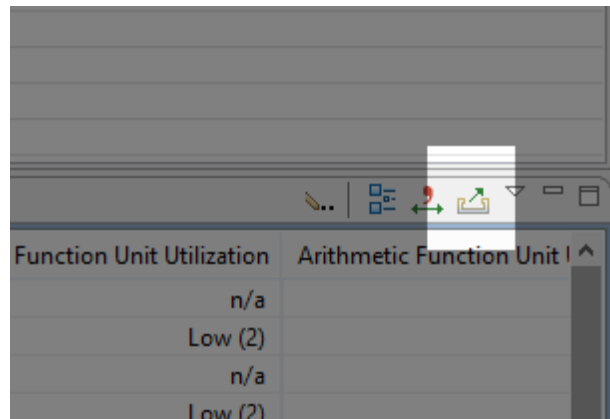


Figure 8.13: Export details to CSV format

# Chapter 9 Nsight

This chapter is mostly relevant to readers who use Visual Studio Professional. The Nsight debugger is a free download from the NVIDIA website. Unfortunately the Express editions of Visual Studio do not allow extensions such as Nsight to be installed.

If you are using the Express edition of Visual Studio, this chapter still may offer some insight into how the device works, but you will not be able to work along with the examples I present.

The Nsight debugger is useful for debugging graphics APIs like DirectX and OpenGL as well as compute APIs. The debugger is available from the following website:

<https://developer.nvidia.com/nvidia-nsight-visual-studio-edition>

To download Nsight from the previous link, you will need to register as an NVIDIA developer and log in to your account. To register as a CUDA developer, visit the following website:

<https://developer.nvidia.com/programs/cuda/register>

The registration and download procedure for Nsight changes over time, and you may need to provide additional details to NVIDIA's staff before you are allowed access the Nsight debugger download.

The download comes with an automatic installer, so to add the extension to Visual Studio you only need to double-click the downloaded file. Once Nsight is installed, you should be able to run Visual Studio Professional and see the Nsight option in the menu bar.

## Generating Profiling Reports

Nsight can generate very detailed reports including all the information that the profiler (NVVP) generates, and a lot more. Choose **Nsight > Start Performance Analysis** from the menu bar. You should see a window that looks like Figure 10.1.



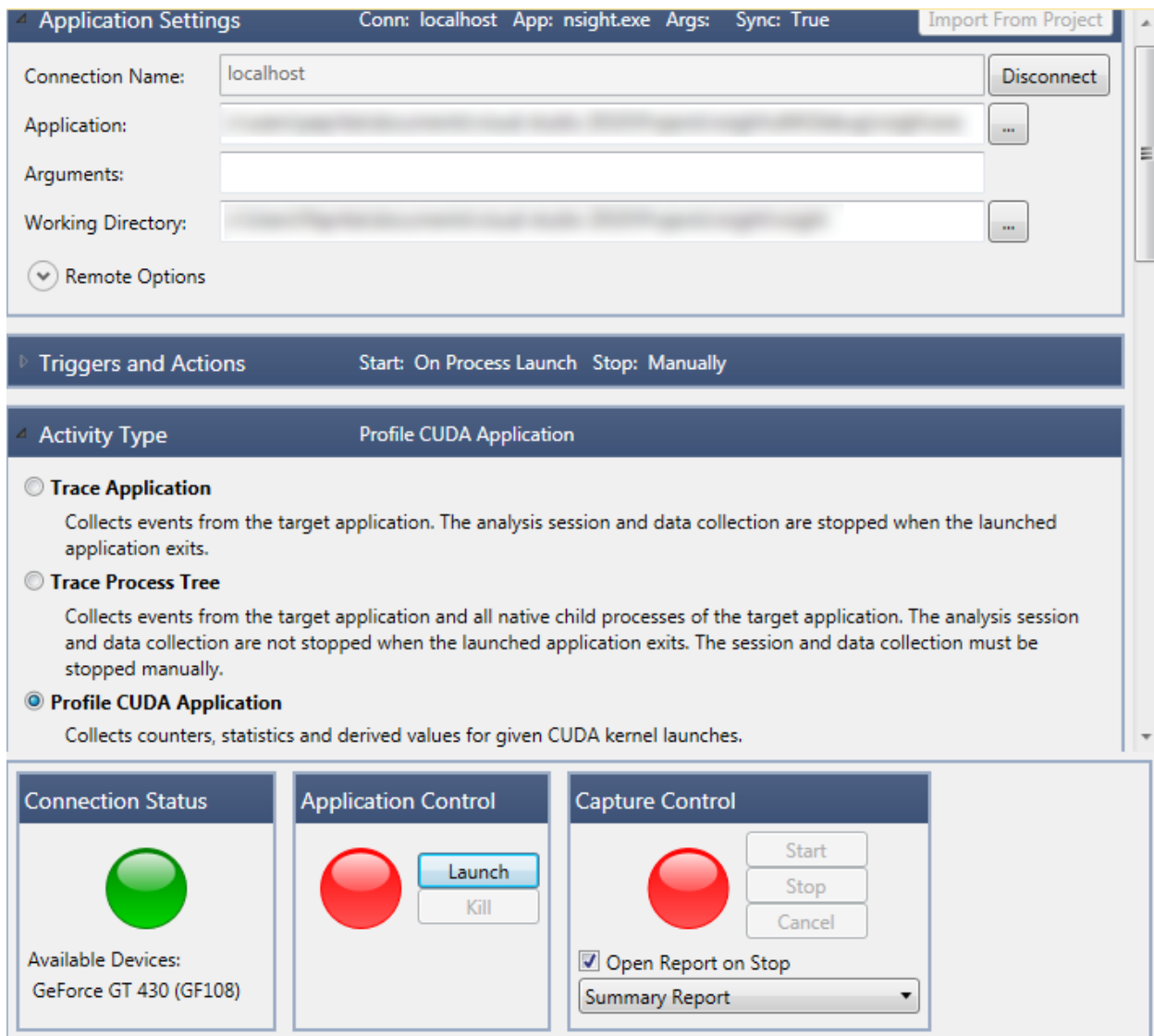


Figure 10.1: Profiling with Nsight

Select **Profile CUDA Application** and click the **Launch** button to run the application and collect metrics just like those from the NVVP. Once the application closes, you will see the Reports tab pictured in Figure 10.2. The **CUDA Launches** selection will show similar data to the NVVP reports.

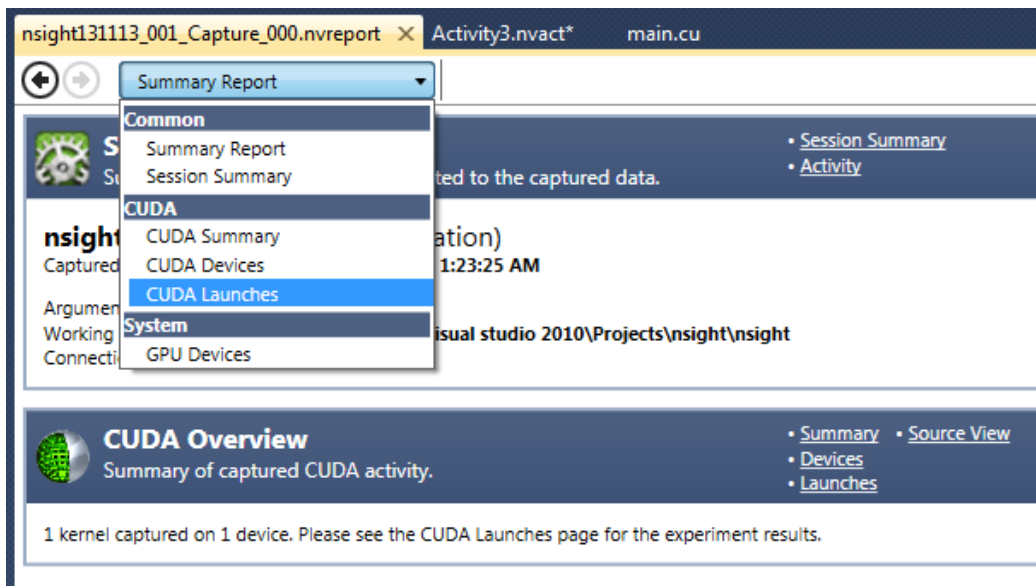


Figure 10.2: Reports from Nsight Profiler

Selecting the **CUDA Devices** or **GPU Devices** reports shows information about the device used in the application. This information is similar to that of Device Query, but much more detailed. There is also a very large number of other reports that can be generated, and which I encourage you to explore.

## Debugging

Using Nsight, we can break into the code of a kernel just as we can in regular application code. To do this, set a breakpoint by clicking in the left margin of the Visual Studio code window, in exactly the same way you would set a host breakpoint. To debug CUDA code, the Nsight monitor must be running. Instead of debugging with the Start Debugging option in the menu bar, you have to select **Start CUDA Debugging** from the **Nsight** menu.

The application will run until the breakpoint is hit; then it will pause and pass control back to Visual Studio. When the program pauses at a breakpoint, you can open two very interesting windows. In the menu bar under the **Nsight** item, select **Windows > CUDA Info**. This will open the CUDA Info window. Or you can open a Warp Watch window by selecting **Window > Warp Watch**. The CUDA Info and Warp Watch windows are only available from the menu bar when the application hits a breakpoint.



**Tip:** You can freeze the warps that you aren't interested in debugging by choosing **Nsight > Freeze** and either **Freeze all except current block** or **Freeze all except current warp**. This will prevent the other warps or blocks from proceeding. When you resume from a break point, the device will break again at the next warp or block, depending on the setting. In this way you can examine each warp or block in turn.

## CUDA Info Window

Current	Frozen	CUcontext	Grid ID	blockIdx	Warp Index	threadIdx	PC	Active Mask	Status	Exception	Exception Details	Global Stat
➡		0x004596f0	8	( 0, 0, 0)	2	( 64, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 0, 0, 0)	3	( 96, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 0, 0, 0)	4	(128, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 0, 0, 0)	5	(160, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 0, 0, 0)	6	(192, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 0, 0, 0)	7	(224, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 1, 0, 0)	0	( 0, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 1, 0, 0)	1	( 32, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 1, 0, 0)	2	( 64, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 1, 0, 0)	3	( 96, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 1, 0, 0)	4	(128, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 1, 0, 0)	5	(160, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 1, 0, 0)	6	(192, 0, 0)	0x0002b790	0xffffffff	None	None	None	None
		0x004596f0	8	( 1, 0, 0)	7	(224, 0, 0)	0x0002b790	0xffffffff	None	None	None	None

Figure 10.3: CUDA Info Window

From the CUDA Info window, you can view information on many aspects of the application by clicking the drop-down list in the upper-left corner, including warps (the default), textures, and memory allocations. The screenshot in Figure 10.3 displays the warps information. Clicking the values highlighted in blue will take you to the info pages for those items.

When CUDA is debugging you can, as expected, press **F11** to step into and **F10** to step over the code. Figure 10.4 shows more details in the Warp Info window. You can see which line in the code the warp is up to, and whether a warp is active in the device or not.

Exception	Exception Details	Global Status Details	File Name	Source Line	Lanes	Freeze State	At Barrier
None	None	None	main.cu	15	➡	None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	
None	None	None	main.cu	15		None	

Coordinates

blockIdx ( 0, 0, 0)

threadIdx ( 20, 0, 0)

General

Current False

Exception None

ExceptionDetails None

PC 0x0002b790

Status Breakpoint

Location

FlatBlockIndex 0

FlatThreadId 20

LaneIndex 20

Figure 10.4: Lanes, source lines, and status



**Tip:** If you double-click on a warp in the CUDA Info window, it will cause all open Warp Watch windows to display the known information for the selected warp.

You can get a good idea of the way branching splits up your warps by examining the Lanes view of the CUDA Info window. When a warp is split into one or more groups, some threads become inactive while a single branch is executed. After each branch's code is executed, the threads re-join each other and continue as a collection of 32. Figure 10.5 illustrates a warp of threads that has been split into two, giving us even threads and odd threads.

CUDA Info 1							
<div> <span>←</span> <span>→</span> <span>Lanes</span> <span>⌵</span> <span>⌶</span> <span>→</span> <span>⌵</span> <span>Filter:</span> </div>							
Current	Lane Index	threadIdx	Status	PC	Exception	Exception Details	
→	0	( 32, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	1	( 33, 0, 0)	Inactive	0x0002b8f8	None	None	
	2	( 34, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	3	( 35, 0, 0)	Inactive	0x0002b8f8	None	None	
	4	( 36, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	5	( 37, 0, 0)	Inactive	0x0002b8f8	None	None	
	6	( 38, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	7	( 39, 0, 0)	Inactive	0x0002b8f8	None	None	
	8	( 40, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	9	( 41, 0, 0)	Inactive	0x0002b8f8	None	None	
	10	( 42, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	11	( 43, 0, 0)	Inactive	0x0002b8f8	None	None	
	12	( 44, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	13	( 45, 0, 0)	Inactive	0x0002b8f8	None	None	
	14	( 46, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	15	( 47, 0, 0)	Inactive	0x0002b8f8	None	None	
	16	( 48, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	17	( 49, 0, 0)	Inactive	0x0002b8f8	None	None	
	18	( 50, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	19	( 51, 0, 0)	Inactive	0x0002b8f8	None	None	
	20	( 52, 0, 0)	Breakpoint	0x0002b8d8	None	None	
	21	( 53, 0, 0)	Inactive	0x0002b8f8	None	None	

Figure 10.5: Lanes and Branching

## Warp Watch Window

We can set a watch on a variable to track how it changes as a warp executes. Since 32 threads of a warp all execute in step, it is more useful to watch 32 copies of the same variable, one for each thread of a warp. You can open up to four Warp Watch windows from the menu bar under **Nsight > Windows > CUDA Warp Watch**. Set a break point in your code and use **Start CUDA Debugging** to debug your program. Figure 10.6 shows an example Warp Watch window.

CUDA WarpWatch 1						
	Name	idx	j	myPoint	destination	mapWidth
Type	int	float	_local_ float3	_shared_ doubl	_constant_ int	
0	0	0	{x = 512, y = 1850278, z = 8.3024288e+11}	0.774596669241	1024	
1	1	8	{x = 512, y = 4.7532601, z = 2.9257022e+11}	0.774596669241	1024	
2	2	16.585787	{x = 512, y = 1.2572764e-31, z = 9.0318583}	0.774596669241	1024	
3	3	25.26795	{x = 512, y = 4.2869922e-29, z = 1.7181141}	0.774596669241	1024	
4	4	34	{x = 512, y = 1.2868255e-06, z = 3.9136199}	0.774596669241	1024	
5	5	42.763931	{x = 512, y = -3.8774644e-27, z = -1.171121}	0.774596669241	1024	
6	6	51.55051	{x = 512, y = -392.25317, z = -3.7180852e+}	0.774596669241	1024	
7	7	60.354248	{x = 512, y = -7.7972254e-39, z = 1.030019}	0.774596669241	1024	
8	8	69.17157	{x = 512, y = -1.0164626e-14, z = -2.856301}	0.774596669241	1024	
9	9	78	{x = 512, y = -92937512, z = 9.1914978}	0.774596669241	1024	
10	10	86.837723	{x = 512, y = 7.228459e+19, z = 1.5219642}	0.774596669241	1024	
11	11	95.683373	{x = 512, y = 1.4343139e+27, z = 1.232879}	0.774596669241	1024	
12	12	104.5359	{x = 512, y = -1.1615851e-36, z = -8.502001}	0.774596669241	1024	
13	13	113.39445	{x = 512, y = -1.7204545e+12, z = -1.67246}	0.774596669241	1024	
14	14	122.25834	{x = 512, y = 3.4004789e-27, z = 1.4492796}	0.774596669241	1024	
15	15	131.12701	{x = 512, y = -1.2012912e-25, z = -2.12376}	0.774596669241	1024	
16	16	140	{x = 512, y = 2.9419642e-28, z = -207.5214}	0.774596669241	1024	
17	17	148.87689	{x = 512, y = 1.6528151e-07, z = -2.480628}	0.774596669241	1024	
18	18	157.75736	{x = 512, y = 4.0952418e+18, z = 8.389997}	0.774596669241	1024	
19	19	166.6411	{x = 512, y = 3.446885e+25, z = 2.5836271}	0.774596669241	1024	
20	20	175.52786	{x = 512, y = -8.3026146e-23, z = -1.03012}	0.774596669241	1024	

Figure 10.6: Warp Watch

To add a watch to the Warp Watch window, at the point a program breaks, click on the gray **<Add Watch>** cell shown in the upper-right corner of Figure 10.6 and type the name of the variable you would like to observe. You can step through the code line by line using **F10** (step over) or **F11** (step into) and see how the 32 warps' variables change. If you have a CUDA Info window open, you can double-click on any currently displayed warp (each line in the warp's view represents a different warp). The Warp Watch windows will switch to the variables for the selected warp. To load the first warp of a block into the open CUDA warp watches, double-click a block in the **Blocks** view of the CUDA Info window.



**Tip:** To set a watch on variables in shared memory, you should turn on *Verbose PTAXS Output*. This option can be found in the project properties in the **CUDA C/C++** section.



**Tip:** You can also watch variables using Visual Studio's standard watch windows. Right-click on the variable's name in the code when the program breaks, and select *Add Watch* from the context menu. Be aware that only the first thread's values from the current warp are displayed when using this technique. Therefore this is mostly useful for watching constant and shared variables, since they are block-wide.

Be sure to rebuild your application before starting the Nsight debugger (Ctrl+Shift+B). The debugger may not rebuild prior to running the application, so if you have made changes to your code, the project must be rebuilt.



**Tip:** You might notice the variables in a warp window not changing with the cursor. To better match the change of variables with the cursor, navigate to **Nsight > Options** and set the value for the **Default Timeline Axis Mode** to **Relative to Cursor**.

When the application uses the Nsight debugger, it runs on the host. You can use **cout** to quickly write to the console window. If you use **printf**, the Nsight debugger will print to the screen, and you can also debug with the standard Visual Studio debugger. When you debug with Visual Studio, all calls to **printf** are ignored, and calls to **cout** in kernels will result in a compilation error. Remove all **cout** and **printf** calls in production code.

# Chapter 10 CUDA Libraries

The CUDA toolkit comes with a powerful collection of libraries, meaning you do not need to code your own versions to satisfy many common problems. In this chapter we will examine two of these libraries in detail. Even if your only interest is to learn efficient code, it's worth getting to know the other libraries.

## Libraries: Brief Description

The following list briefly describes some of the libraries that are installed with the CUDA toolkit. These libraries are all of a very high quality, and are optimized, well tested, and generally very simple to use. These libraries are actively updated, and each new version of the CUDA toolkit usually includes newer versions of each of these libraries.

**CUFFT**—CUDA Fast Fourier Transform. This is a CUDA-accelerated implementation of the fast Fourier transform algorithm, which is extremely important in countless fields of study.

**CUBLAS**—CUDA Basic Linear Algebra Subprograms. This is a CUDA implementation of the popular BLAS library.

**CUSPARSE**—This library contains CUDA-accelerated algorithms for working with sparse and dense vectors and matrices.

**NPP**—NVIDIA Performance Primitives. This library presently contains implementations of several algorithms for working with video and image data and some signal processing functions. It is designed to closely resemble the Intel Integrated Performance Primitives library, but it is not currently a complete implementation.

**CURAND**—CUDA Random Number Generation. This library contains implementations of many strong and efficient random number generators, which are accelerated by CUDA. We will look at the CURAND library in some detail in examples later in this chapter.

**Thrust**—Thrust is a template library using parallel algorithms and data structures. It is designed as a CUDA-accelerated implementation of the Standard Template Library (STL), which is familiar to most C++ programmers. We will look at Thrust in some detail later in this chapter.

**Others**—There are countless libraries available online, and the collection is constantly expanding and being updated. For additional libraries, check out <https://developer.nvidia.com/gpu-accelerated-libraries>. The NVIDIA website does not offer an exhaustive list but libraries mentioned on the website tend to be of a very high quality.

We will now examine two of the previously mentioned libraries with a little more detail: CURAND and Thrust. The implementation and use of the other libraries (either those mentioned in the previous list or any of the countless libraries) is often similar to CURAND and/or Thrust. The following sections are intended to be a general introduction to using CUDA accelerated libraries with CURAND and Thrust as examples.



# CUDA Random Number Generation Library (CURAND)

CURAND is a CUDA-accelerated collection of pseudo-random number generators. It includes a multitude of different generators, most of which generate extremely high quality random numbers, the sequences of which have very long periods. The library is useful for two reasons: it is able to generate sets of random numbers very quickly because it is CUDA accelerated, but also it contains special generators that are designed to generate random numbers on the fly in device code.

## Host and Device APIs

The beauty of the CURAND library is that it is not only useful for quickly generating blocks of random numbers for consumption by the device (or the host), but it can also generate numbers in kernel code. There are two APIs included; one (the host API) is designed to generate random numbers with the host and store the resulting sequence in global memory (or system memory). The host can then call a kernel, passing the block of random numbers as a parameter for consumption by the device, or the block of random numbers can be consumed by the host. The other API (the kernel or device API) is designed to generate random numbers on the fly within kernels for immediate consumption by CUDA threads.

To use the CURAND library functions you must include the **curand.h** header in your project and link to the **curand.lib** library file in your project's properties. To include the various header and libraries in your project, look in the same folders as you did previously, and find **curand.h** and **curand.lib**. If they are there and you set your paths up correctly at the beginning of the book, then you should be able to just **#include** the headers where needed, and add the .lib to your linker list as you did with **cuda.lib**.

The **curand.h** header is the header for the host API. Additionally, you can include the **curand\_kernel.h** header to use the device API, which we will examine shortly.

## Host API

The host API can be used to generate blocks of random numbers in the device's global memory or in system memory. The following code listing (Listing 10.1) generates two blocks of random numbers using the host API: one in global memory and one in system memory. The host API can be used without the CUDA build customization and in a standard .cpp, instead of .cu files. The following code (Listing 10.1) was created as a new project. This section is not meant as a continuation of what we have previously coded, but is a completely new example project.

```
// main.cpp
#include <iostream>
#include <ctime> // For seeding with time(NULL)
#include <cuda.h>
#include <curand.h> // CURAND host API header

using namespace std;
```



```

int main() {
// Count of numbers to generate
const int count = 20;

// Device generator stores in device global memory
curandGenerator_t deviceGenerator;

// Host generators store in system RAM
curandGenerator_t hostGenerator;

// Define device pointer for storing deviceGenerator results
float *d_deviceOutput;

// Define host arrays for storing hostGenerator's results
float h_deviceOutput[count], hostOutput[count];

// Allocate memory on device for deviceGenerator
cudaMalloc(&d_deviceOutput, sizeof(float)*count);

// Initialize the two generators
curandCreateGenerator(&deviceGenerator, CURAND_RNG_PSEUDO_DEFAULT);
curandCreateGeneratorHost(&hostGenerator, CURAND_RNG_PSEUDO_DEFAULT);

// Set the seed for both generators to 0. Using the same seed will
// cause both generators create exactly the same sequence.
curandSetPseudoRandomGeneratorSeed(deviceGenerator, 0);
curandSetPseudoRandomGeneratorSeed(hostGenerator, 0);

// If you would like the generators to use an unpredictable seed and
// create different sequences, you can use time(NULL) as the seed.
// Alter the value in some way (I've multiplied my hostGenerators by 10
// in the following) otherwise time(NULL) will be exactly the same when
// you seed them.
// curandSetPseudoRandomGeneratorSeed(deviceGenerator, time(NULL));
// curandSetPseudoRandomGeneratorSeed(hostGenerator, time(NULL)*10);

// Generate a random block of uniform floats with both generators:
// The deviceGenerator outputs to global memory:
curandGenerateUniform(deviceGenerator, d_deviceOutput, count);
// The hostGenerator's output is in system RAM
curandGenerateUniform(hostGenerator, hostOutput, count);

// Copy the device results to the host
cudaMemcpy(h_deviceOutput, d_deviceOutput,
           sizeof(float)*count, cudaMemcpyDeviceToHost);

// Free device and CURAND resources
cudaFree(d_deviceOutput);
curandDestroyGenerator(deviceGenerator);
curandDestroyGenerator(hostGenerator);

// Print out the generator sequences for comparison
for(int i = 0; i < count; i++) {

```

```

        cout<<i<<". Device generated "<<h_deviceOutput[i]<<
            " Host generated "<<hostOutput[i]<<endl;
    }

    return 0;
}

```

*Listing 10.1: CURAND Host API*

Listing 10.1 generates two blocks of 20 random, floating point values. There are two types of generators in Listing 10.1: host and device (not to be confused with the two APIs). Both of these generators belong to the host API, and the device API generates its values inside kernels. The difference between the two generators is that the host generator (**hostGenerator**) stores the resulting sequence in system RAM, while the device generator (**deviceGenerator**) stores the sequence in global memory. The two generators in the previous listing should generate exactly the same values for their blocks of random floats, because they use the same seed. In fact, every time the program is executed, the generators will generate exactly the same sequences of numbers. If you comment out the lines that seed the generators to **0**, and uncomment the lines that seed to **time(NULL)**, you should find that the generators generate different numbers from each other, and different values each time the program is executed.



**Note:** You need not include the `cuda.h` header because the CUDA API is already included in the `curand.h` header. Likewise, the `time(NULL)` function can be called without the inclusion of the `ctime` header. These headers were included for clarity.

To generate random numbers with the CURAND library, you first need to create generator handles.

You can create a host generator with **curandCreateGeneratorHost** or a device generator with **curandCreateGenerator**. The first parameter to each of these functions is a **curandGenerator\_t**, which will be passed to subsequent CURAND function calls as a generator handle.

The second parameter is the type of random number generator to use. There are many different algorithms for generating random numbers with computers, and the list of random generators changes with each generation of this library. To see the full list of algorithms included in the version of the CURAND library you have installed, right-click on the **CURAND\_RNG\_PSEUDO\_DEFAULT** identifier and select **Go to definition** from the context menu in Visual Studio, or search for the file **CURAND.h** and examine its contents.

The **d\_deviceOutput** pointer is a device pointer to data in the device's global memory. After the sequence is generated, this pointer could be passed to a kernel for consumption. In Listing 10.1, it is copied back to the host and printed to the screen to be compared with the host's generated values.



**Tip:** When using the host API for consumption by the device, it is best to generate the numbers as infrequently as possible. The numbers should be generated once prior to a

*kernel call. It would be very slow to jump back and forth between the host and device while the host generates new numbers in the middle of an algorithm.*

## Uniformly Distributed Generation

A uniformly distributed random `float` is one that lies between `0.0f` and `1.0f`. In theory, every possible value between these points is as likely as any other to be generated in the sequence. In reality there are an infinite number of real values between any two real numbers, but in computing there are around  $2^{24}$  different floats between `0.0f` and `1.0f`.



***Tip:** Random numbers in any range can be created from a set of random normalized numbers (such as those generated by `curandGenerateUniform`). For instance, to convert the sequences from Listing 10.1 into random floats between `100.0f` and `150.0f`, you could multiply each by `50.0f` (the difference between the lowest value and the highest, `150.0f-100.0f`) and add `100.0f`. Be aware that the `curandGenerateUniform` function generates from `0.0f` to `1.0f`, but `0.0f` is not included, and `1.0f` is.*

## Normally Distributed Generation

Sometimes we need to generate numbers with a normal distribution. This means the values most likely to be generated will be those closest to the mean. Numbers further away from the mean have a decreasing likelihood of being generated, and the frequency of numbers when plotted will look like the familiar bell curve of a normal distribution.

In a normal distribution any number is possible, but the chances of generating numbers far from the mean (in standard scores) becomes very small very quickly. Use the `curandGenerateNormal` function to generate numbers based on a normal curve. The function requires a mean and a standard deviation value to describe the fall off and magnitude of the curve from which to generate numbers. For instance, replace the call to `curandGenerateUniform` in the previous code with a call to `curandGenerateNormal` based on the standard normal curve (this a normal curve with mean `0.0f` and standard deviation of `1.0f`) as per Listing 10.2.

```
// Generate a random block of floats from normal curves
curandGenerateNormal(deviceGenerator, d_deviceOutput, count, 0.0f, 1.0f);
curandGenerateNormal(hostGenerator, hostOutput, count, 0.0f, 1.0f);
```

*Listing 10.2: Generating numbers from normal curves*

You will see that the numbers generated tend towards `0.0f` (the mean of our normal curve and 4<sup>th</sup> parameter of the `curandGenerateNormal` function), but that several are `1.0f` or more away from this mean. Most (theoretically 68.2 percent) of the numbers generated will be within `-1.0f` and `1.0f` because we supplied `1.0f` as our standard deviation (the 5<sup>th</sup> parameter of the `curandGenerateNormal` function) and `0.0f` is the middle of `-1.0` and `1.0`. Using the values `0.0f` for the mean and `1.0f` for the standard deviation means the number will be generated from a standard normal curve.

## Device API

The device API allows for the generation of random values within a kernel, without leaving the device. The device API is a little more complicated than the host API because it needs to set up and maintain the states of multiple generators (one for each thread). Include the `curand_kernel.h` header in your project files to use the device API and, as with the host API, link to the `curand.lib` library in your project's properties. The generation occurs within kernels, so (unlike the host API) you will need to use the CUDA Build Customization to launch kernels and store the code in a `.cu` file. Examine the code in Listing 10.3, in particular the `InitRandGenerator` and `Generate` kernels. Again, this code is intended as a new project; it does not build on anything we have previously coded.

```
// main.cu
#include <iostream>
#include <cuda.h>
#include <curand_kernel.h>
#include <device_launch_parameters.h>

using namespace std;

// Kernel to init random generator states
__global__ void InitRandGenerator(curandStateMRG32k3a *state) {
    int id = threadIdx.x + blockIdx.x * blockDim.x;

    // Init the seed for this thread
    curand_init(0, id, 0, &state[id]);
}

// Generate random numbers using the device API
__global__ void Generate(unsigned int *result, int count, curandStateMRG32k3a
*state) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    // Make a local copy of the state
    curandStateMRG32k3a localState = state[idx];

    // Generate one number per thread
    if(idx < count) {
        // The numbers generated will be integers between 0 and 100
        result[idx] = (unsigned int) curand(&localState)%100;
    }
}

int main() {
    // How many numbers to generate
    const int count = 100;

    // Declare the arrays to hold the results
    unsigned int* h_a = new unsigned int[count];
    unsigned int* d_a;
    cudaMalloc(&d_a, sizeof(unsigned int)*count);
```

```

// Declare the array to hold the state of the generator
curandStateMRG32k3a *generatorState;
cudaMalloc(&generatorState, sizeof(curandStateMRG32k3a) * count);

// Call an init kernel to set up the state
InitRandGenerator<<<(count/64)+1, 64>>>(generatorState);

// Call a kernel to generate the random numbers
Generate<<<(count/64)+1, 64>>>(d_a, count, generatorState);

// Copy the results back to the host
cudaMemcpy(h_a, d_a, sizeof(unsigned int)*count, cudaMemcpyDeviceToHost);

// Print the results to the screen
for(int i = 0; i < count; i++)
    cout<<i<<" ". "<<h_a[i]<<endl;

// Free resources
cudaFree(d_a);
cudaFree(generatorState);
delete[] h_a;

return 0;
}

```

*Listing 10.3: CURAND Device API*

Listing 10.3 generates 100 random unsigned **ints** using the CURAND device API. It stores the numbers in global memory before copying them to the host and printing them out. The important thing to remember here is that the generated numbers can be consumed immediately in the kernel's code; I have copied them back purely for purposes of demonstration.

An area of device memory needs to be allocated to hold the state of the generator; this can be done using **cudaMalloc**. I selected the **curandStateMRG32k3a** state, which will use a multiple recursive generator algorithm (this is the CUDA version of the **MRG32k3a** algorithm available in the Intel MKL Vector Statistical Library).

The seed and current position of the numbers generated so far for each thread must be known prior to generating subsequent random numbers. This state maintains this information.

Each thread sets up its own seed based on the **threadIdx** by calling the **curand\_init** device function (in the **InitRandGenerator** kernel).

Since **threadIdx.x** is calculated to be unique within this grid, each thread will generate a different seed. Once the state has been seeded, the threads can generate a random number by calling **curand**. The **curand** function returns the next sequence of 32 bits from the state for each thread. We can take the remainder after division with the modulus operator to get a number within a certain range (in Listing 10.3 the numbers will range from 0 to 99 because of the **%100**).

The line `curandStateMRG32k3a localState = state[idx];` creates a local copy of the state for each thread. It makes no difference to the performance of Listing 10.3, but if your kernel generates many numbers per thread, it will be quicker to use a local copy. Creating a local copy means that we do not need to use array subscripts, which are slower.



*Tip: Creating generators is a slow process and they should, if possible, be created once and reused to avoid the overhead of setting up the states each time.*

## Thrust

The second library we will look at in detail is the Thrust library. The Thrust library is a CUDA-accelerated version of the Standard Template Library (STL) for C++. The library provides data types and common algorithms optimized with CUDA.



*Tip: Many of the identifiers included in the Thrust library have the same names as those from the STL. When mixing Thrust and STL code it may be helpful to avoid using the using directive to include the entire namespaces. Instead, qualify references fully. For instance, use `std::sort` and `thrust::sort`.*

There are two different types of basic vectors included in the Thrust library: **host\_vector** and the **device\_vector**. Host vectors are stored in system memory and controlled by the host, while device vectors are stored in device memory and controlled by the device.

The Thrust headers are located in the **thrust** subdirectory of the main CUDA toolkit headers directory. To use host or device vectors in your code, include the appropriate header, **thrust/host\_vector.h** or **thrust/device\_vector.h**.

Thrust does not have its own library (i.e. there is no `thrust.lib` like there was for CURAND). But it does use the standard CUDA Runtime library, so make sure your project links to **CUDART.lib** before using Thrust.

## Basic Vector Manipulations

To define a Thrust vector, use `thrust::host_vector<type>` or `thrust::device_vector<type>` where **type** is the data type for the vector. There are various constructors you can use to initialize the vectors, including defining the size and initial values for each of the elements in the vector. Listing 10.4 shows some basic examples of defining vectors, accessing elements, and copying data to and from the various vector library types.

```
// main.cu
#include <iostream>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <vector>    // STL vector header
```

```

int main() {
// Defining vectors

// STL vector, 100 ints set to 18
std::vector<int> stdV(100, 18);

// Host vector of 100 floats
thrust::host_vector<float> h(100);

// Host vector, 100 ints set to 67
thrust::host_vector<int> hi(100, 67);

// Empty device vector of doubles
thrust::device_vector<double> dbv;

// Device vector, 100 ints set to 0
thrust::device_vector<int> di(100, 0);

//
// Adding and removing elements
//

// Print initial element count for h
std::cout<<"H contains "<<h.size()<<" elements"<<std::endl;

// Remove last element
h.pop_back();
std::cout<<"H contains "<<h.size()<<" elements"<<std::endl;

// Adding and removing elements from device vector is the same
dbv.push_back(0.5);
std::cout<<"Final elements of dbv is "<<dbv.back()<<std::endl;
dbv.pop_back();

//
// Copying data between vectors
//

// Print out the first element of h
std::cout<<"First element of hi is "<<hi[0]<<std::endl;

// Copying host vector to device vector
di = hi;
std::cout<<"First element of di is "<<di[0]<<std::endl;

// Copy from STD vector to device vector
di = stdV;
std::cout<<"First element of di is "<<di[0]<<std::endl;

return 0;
}

```

*Listing 10.4: Basic Vector Manipulations*

Listing 10.4 shows that vectors can be copied from host to device and even STL vectors using the equal operator, `=`. Copying to or from a device vector uses `cudaMalloc` behind the scenes. Bear this in mind when coding with Thrust, or code can drop dramatically in performance, especially when performing many small copies.

Elements can be accessed using the array subscript operators `[ ]`. The standard STL vector manipulations like `push_back`, `pop_back`, and `back` (and many others) are all available for Thrust vectors. Vectors local to a function, device and host, are disposed of automatically when the function returns.

## Sorting with Thrust

Listing 10.5 shows an example of how to use the sorting algorithm available in Thrust. It creates and sorts three lists of 20,000,000 integers. One is sorted using STL, one is a Thrust host vector, and the final one is a Thrust device vector. Each vector type sorts the integers five times, regenerating a random list between sorts. Note that, in this example, the device vector does not actually generate the random numbers, but copies them from the host vector.

```
// main.cu
// Standard headers
#include <iostream>
#include <ctime>

// CUDA and Thrust headers
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

// STL headers
#include <vector>
#include <algorithm>

// Helper defines and variable for timing
long startTime;
#define StartTimer() startTime=clock()
#define ElapsedTime (clock()-startTime)

int main() {
    int count = 20000000; // 20 million elements to sort

    // Vectors
    std::vector<int> stlVec(count); // STL vector
    thrust::host_vector<int> hostVec(count); // Thrust host vector
    thrust::device_vector<int> devVec(count); // Thrust device vector

    std::cout<<"Comparison of Thrust and STL sort times on vectors of size "<<
        count<<"\n\n";

    // Sort 5 times using STL vector
    std::cout<<"STL vector:"<<std::endl;
    for(int i = 0; i < 5; i++) {
```



```

        std::generate(stlVec.begin(), stlVec.end(), rand);
        StartTimer();
        std::sort(stlVec.begin(), stlVec.end());
        std::cout<<"Run["<<i<<"] Sorted in "<<ElapsedTime<<" millis"<<
            std::endl;
    }

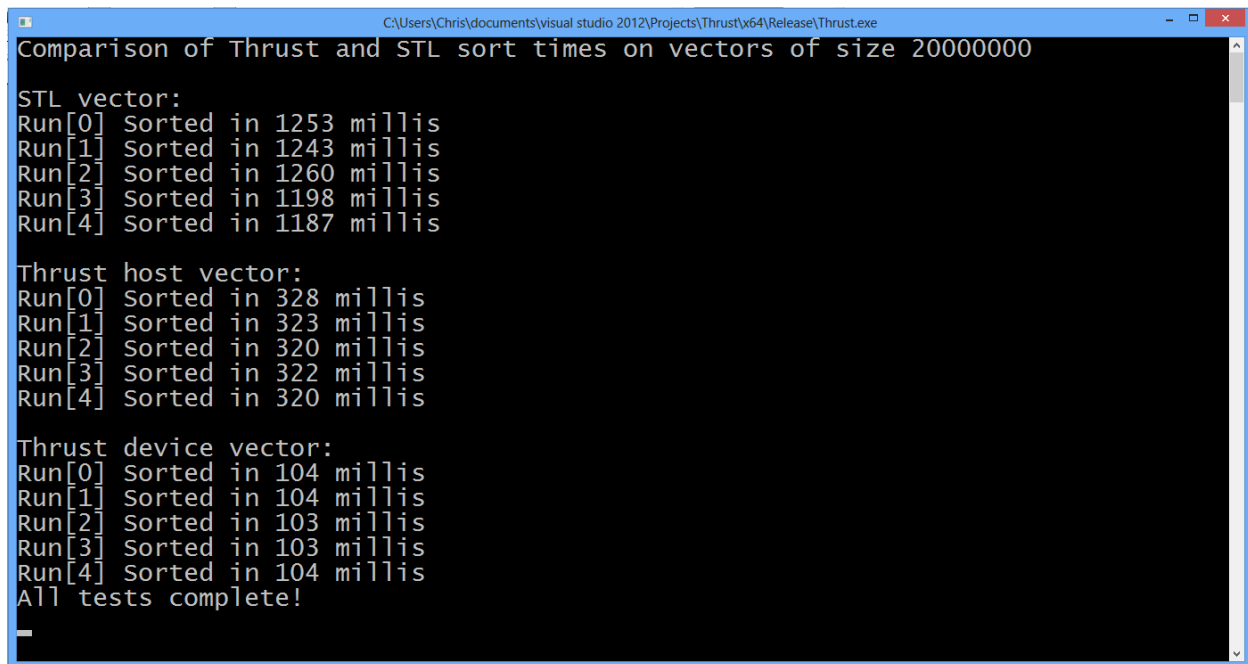
    // Sort 5 times using Thrust host vector
    std::cout<<"\nThrust host vector:"<<std::endl;
    for(int i = 0; i < 5; i++) {
        thrust::generate(hostVec.begin(), hostVec.end(), rand);
        StartTimer();
        thrust::sort(hostVec.begin(), hostVec.end());
        std::cout<<"Run["<<i<<"] Sorted in "<<ElapsedTime<<" millis"<<
            std::endl;
    }

    // Sort 5 times using Thrust device vector
    std::cout<<"\nThrust device vector:"<<std::endl;
    for(int i = 0; i < 5; i++) {
        // Host generates values for devVec
        thrust::generate(hostVec.begin(), hostVec.end(), rand);
        devVec = hostVec; // Copy from host since there's no rand()

        StartTimer();
        thrust::sort(devVec.begin(), devVec.end());
        std::cout<<"Run["<<i<<"] Sorted in "<<ElapsedTime<<" millis"<<
            std::endl;
    }
    std::cout<<"All tests complete!"<<std::endl;
    return 0;
}

```

*Listing 10.5: Vector Sorting Comparison*



```
C:\Users\Chris\documents\visual studio 2012\Projects\Thrust\x64\Release\Thrust.exe
Comparison of Thrust and STL sort times on vectors of size 20000000

STL vector:
Run[0] Sorted in 1253 millis
Run[1] Sorted in 1243 millis
Run[2] Sorted in 1260 millis
Run[3] Sorted in 1198 millis
Run[4] Sorted in 1187 millis

Thrust host vector:
Run[0] Sorted in 328 millis
Run[1] Sorted in 323 millis
Run[2] Sorted in 320 millis
Run[3] Sorted in 322 millis
Run[4] Sorted in 320 millis

Thrust device vector:
Run[0] Sorted in 104 millis
Run[1] Sorted in 104 millis
Run[2] Sorted in 103 millis
Run[3] Sorted in 103 millis
Run[4] Sorted in 104 millis
All tests complete!
```

Figure 10.1: Example output from Listing 10.5

The Thrust device vector is clearly very fast (Figure 10.1 shows the device vector is around 10 times faster than STL and around 3 times faster than the host vector). What is perhaps surprising is that the host vector is considerably faster than the STL. When the lists are sorted already (or nearly), the STL sort has a special trick; it notices the elements are in order and quits the sort early. This gives the STL a peculiar advantage with nearly sorted lists. Placing the lines that regenerate the random list outside the `for` loops shows this. The STL sorts at around 168 milliseconds on the tested machine when the list is sorted.

Replacing all the calls to `sort` with `stable_sort` implements a stable sort. The indices of identical elements are guaranteed to maintain order with respect to one another through a stable sort. A stable sort slows the STL vector down while the Thrust vectors are relatively unaffected.

## Transformations, Reductions, and Stream Compaction

As a final example, Listing 10.6 shows more of the Thrust library's abilities. It finds the factors of an integer up to its square root. This is a very inefficient method for factorizing integers (it is just a brute force search with a marvelously undesirable memory footprint), but it shows how to use several other important functions of the Thrust library.

The following code will not work on devices with compute capability 1.x. Remember to set the SM and compute values in your project to 2.0 or higher.

```
// main.cu

#include <iostream>
```

```

// Thrust headers
#include <thrust\device_vector.h>
#include <thrust\transform.h>
#include <thrust\sequence.h>
#include <thrust\copy.h>

// Helper function, returns true if x is zero
// else return false
struct isZero {
    __device__ bool operator()(const int x) {
        return x == 0;
    }
};

int main() {

    int x = 600; // The integer to factorize

    //
    // Define Thrust device vectors
    //
    // dx vector will be filled with multiple copies of x
    thrust::device_vector<int> dx((int)(std::sqrt((double)x)+1));
    // dy vector will be filled with sequence (2, 3, 4... etc.)
    thrust::device_vector<int> dy((int)(std::sqrt((double)x)+1));
    // factors vector is used to store up to 20 factors
    thrust::device_vector<int> factors(20);

    // Fill dx with multiple copies of x
    thrust::fill(dx.begin(), dx.end(), x);

    // Fill dy with a sequence of integers from 2 up to
    // the square root of x
    thrust::sequence(dy.begin(), dy.end(), 2, 1);

    // Set elements of dx[] to remainders after division
    // of dx[] and dy[]
    thrust::transform(dx.begin(), dx.end(), dy.begin(), dx.begin(),
        thrust::modulus<int>());

    // Multiply all remainders together
    int product = thrust::reduce(dx.begin(), dx.end(), 1,
        thrust::multiplies<int>());

    // If this product is 0 we found factors!
    if(!product && x >= 4) {
        std::cout<<"Factors found!"<<std::endl;

        // Compact dx to factors where dy is 0
        thrust::copy_if(dy.begin(), dy.end(), dx.begin(),
            factors.begin(), isZero());

        // Print out the results
    }
}

```

```

        for(int i = 0; i < factors.size(); i++) {
            if(factors[i] != 0)
                std::cout<<factors[i]<<" is a factor of "<<
                    x<<std::endl;
        }
    }
else { // Otherwise x is prime
    std::cout<<x<<" is prime, factors are 1 and "<<x<<std::endl;
}

return 0;
}

```

*Listing 10.6: Factorizing Integers*

After the headers in Listing 10.6, you will see the definition of a structure (**isZero**). The structure consists of a single Boolean function, which is used as a predicate later in the code.

Three vectors are used in the **main** method. **dx** and **dy** (nothing to do with calculus) are used as two temporary device storage vectors. **dx** holds multiple copies of the integer being factorized, which is the value of the variable **x**. **dy** holds all of the integers from 2 up to the square root of **x**.

To fill the **dx** vector with multiple copies of the **x** variable, the **thrust::fill** function is employed. It takes two iterators, which specify the start and end of the area to fill, and a third parameter, which is the value to fill with.

To create a list of all integers from 2 up to the square root of **x** and store them in the **dy** vector, I have used the **thrust::sequence** function. There are overloaded versions of **sequence**, but the version used here takes four parameters. The first two parameters are the start and end iterators for the area being filled. The third parameter is the value to start with, and the final parameter is the step or gap between the values generated.

The next step is to divide the **xs** stored in **dx** with the sequential integers in **dy**, taking the remainder with the modulus operator. In the code this is achieved with the **thrust::transform** function. Transforms perform the same operation on many elements of vectors. The transform function used in the code takes four parameters. The first two are the start and end iterators of the input. The next is the start iterator for the second input (modulus requires two operands). The third parameter is the output iterator to which the results will be stored, and the final parameter is the function to be performed between the elements of the **dx** and **dy** vectors. The result of this function call is that each of the **xs** in **dx** is divided by the corresponding element in **dy**, and the remainder after the division replaces the original values in **dx**.

The next function call is an example of using a **thrust::reduce**. A **reduce** algorithm (or parallel reduction) reduces the elements of a vector (or array or any other list) to a single value. It could, for instance, produce the sum of elements in an array, the standard deviation, average, or any other single value. In the code it is used to multiply all the elements of **dx** together. If any of them were 0, the result of this multiplication would be 0, which means we found factors (i.e. some integers were found that evenly divide **x**). If the result of this multiplication is not 0, it means none of the integers in **dy** are factors of **x**. If no integer from 2 to the square root of **x** is a factor of **x**, then **x** is prime.

The `thrust::reduce` function used here takes four parameters. The first two are the start and end iterators to the data that is to be reduced. The third parameter is the start value for the reduction. If you were summing a list of numbers you might use 0 for the initial value, but here we are multiplying, so I have used 1. The final parameter is the operation to use. I have used integer multiplication with `thrust::multiplies<int>()`. The resulting `float (product)` from this reduction is the product of all the elements in the `dx` vector. The reduction, as used here, is a complete waste of time, and it is only supplied for illustration of `thrust::reduce`.

The final task (if we found the factors and `product` was 0) is to copy the factors we found into another vector (the `factors` vector). To do this we want to copy elements from `dy` to `factors` where `dx`'s elements are 0. It is a complicated operation to describe in English, but it is easily achieved by a call to `thrust::copy_if`. This operation is called a stream compaction. We are compacting the elements in `dy` to another vector based on some predicate (i.e. that the corresponding element in `dx` is 0). The function (`thrust::copy_if`) as used here takes five parameters; the first two are the start and end iterators of the stream being compacted, which is `dy`. The next is called the stencil parameter; it is the vector of the values to compare against the predicate. Here, we are checking if `dx`'s values are 0, so the stencil is `dx`. The fourth parameter is the start iterator to store the results, `factors.begin()`. The final parameter is the predicate, a function returning `bool`, which decides which elements are copied and which are not.

The `isZero()` function was defined at the top of the application; it returns `true` when its parameter is 0 and `false` for any other values. This predicate (`isZero`) is a `__device__` function. Every element in the `dx` vector will be passed through the predicate function concurrently; when the results are true, the corresponding elements of `dy` are copied to subsequent positions in `factors`. The end result of the stream compaction is that the factors of `x` will be stored in the `factors` vector, from which they are printed to the screen using a `for` loop and `cout`.

This has been a very quick look into some aspects of the Thrust library. Each of the categories of algorithm (transforms, reductions, and stream compactions) are far more flexible than this simple example illustrates. In addition, the Thrust library has a collection of very powerful and flexible iterators, which we have not looked at in these examples. For further reading on the topic of the Thrust library, consult the Thrust Quick Start Guide, which comes with the CUDA Toolkit. The exact path of the PDF documentation will differ for each version of the toolkit; the following is the default path for version 6.5:

C:\Program Files\NVIDIA GPU Computing  
Toolkit\CUDA\v6.5\doc\pdf\Thrust\_Quick\_Start\_Guide.pdf

The Thrust Quick Start Guide is also available online from NVIDIA at <http://docs.nvidia.com/cuda/thrust/index.html>.

# Conclusion

Throughout this book we have looked at many aspects which together form the CUDA programming architecture. I have necessarily left out the vast majority of information and concentrated only on a few key aspects of the technology. We have seen that even a modest modern graphics card is an extremely powerful piece of hardware, and that CUDA offers potential performance gains that are very exciting to anybody interested in pushing modern hardware to its limits. A few hours spent porting an algorithm to a GPU may yield a speed-up that would be completely impossible to gain with a CPU, regardless of the time spent optimizing.

Aside from the obvious gains in performance, I hope I have shown that studying the CUDA SDK gives programmers a completely new set of skills.

The sheer number of concurrent parallel threads makes for a very different programming experience than one might get from programming a handful of threads with a CPU.

I have spoken only about CUDA in this book. There are other graphics cards manufacturers, and they make equal hardware. They have their own languages and ecosystems. There is also OpenCL, which is an open source compute library (very similar to the CUDA driver API) that has the great benefit of working on many different graphics cards, including on-board graphics.

Thank you very much for reading, and I hope you have enjoyed this short adventure into the world of general purpose programming using NVIDIA hardware and CUDA. If you would like to explore further, I recommend looking into graphics APIs, DirectX or OpenGL, and other GPGPU languages, such as OpenCL.

# More Information

## GPU Technology Conference (GTC)

<http://www.gputechconf.com/page/home.html>

If you are interested in finding more information on any topics involving CUDA, GPGPU, or graphics cards in general, watch relevant video presentations from the GTC.

## GTC On Demand

<http://www.gputechconf.com/gtcnew/on-demand-gtc.php>

The GTC On-Demand site offers presentations from past years of the GTC. The sheer amount of information presented here is incredible. Every aspect of programming graphics cards has been discussed in detail and most of the older presentations are still relevant.

## CUDA Library Documentation

<https://developer.nvidia.com/gpu-accelerated-libraries>

The CUDA libraries are very powerful. We looked at only two, and rather quickly. Full references for each library can be found at the bottom of the library's webpage. In addition, PDF versions of these documents are installed with the SDK and can be found in the /doc/ subfolder of the CUDA toolkit (depending on the CUDA version you have installed the exact path may be differ, but it will likely be something similar to the following: C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v6.5\doc\pdf).

You will find many PDF documents here, including:

- CUDA\_C\_Programming\_Guide.pdf, a general reference to CUDA C.
- CUDA\_C\_Best\_Practices\_Guide.pdf, a reference full of NVIDIA's recommended practices when programming with CUDA.
- CUDA\_Runtime\_API.pdf, a reference to the runtime API.

## The Little Book of Semaphores by Allen B. Downey

<http://www.greenteapress.com/semaphores/>

This book is a true gem—it is a must-read for all who wish to dip into the baffling but very beautiful world of parallel problem solving using semaphores and mutexes. It is the best book on the topic available, and Mr. Downey's talk (available from the same site) is an excellent supplement. The simple algorithms we programmed did not require any of these synchronization primitives, but as the complexity of thread interaction increases, a solid knowledge of semaphores becomes indispensable.