# Introduction to CUDA 5.0

CUDA 5

In this article, I will introduce the reader to CUDA 5.0. I will briefly talk about the architecture of the Kepler **GPU** (**G**raphics **P**rocessing **U**nit) and I will show you how you can take advantage of the many **CUDA** (**C**ompute **U**nified **D**evice **A**rchitecture) cores in the **GPU** to create massively parallel programs.

## List of Figures

## List of Tables

## Introduction

Using the power of the NVIDIA GPU, CUDA allows the programmer to create highly parallel applications that can perform hundreds of times faster than an equivalent program that is written to run on the CPU alone. The NVIDIA CUDA Tookit provides several API's for integrating a CUDA program into your C and C++ applications.

CUDA supports a heterogeneous programming environment where parts of the application code is written for the CPU and other parts of the application are written to execute on the GPU. The application is compiled into a single executable that can run on both devices simultaneously.

In a CUDA intensive application, the CPU is used to allocate CUDA memory buffers, execute CUDA kernels and retrieve and analyze the result of running a kernel on the GPU. The GPU is used to synchronously process large amounts of data or to execute a simulation that can easily be split into a large grid where each grid executes a part of the simulation in parallel.

The NVIDIA GPU consists of hundreds (even thousands) of CUDA cores that can work in parallel to operate on extremely large datasets in a very short time. For this reason, the NVIDIA GPU is much more suited to work in a highly parallel nature than the CPU.

The image below shows the computing power of the GPU and how it compares to the CPU. The vertical axis shows the theoretical **GFLOP**/s (**G**iga **F**loating **P**oint **O**perations per **S**econd). The horizontal axis shows the advances in technology over the years[1].

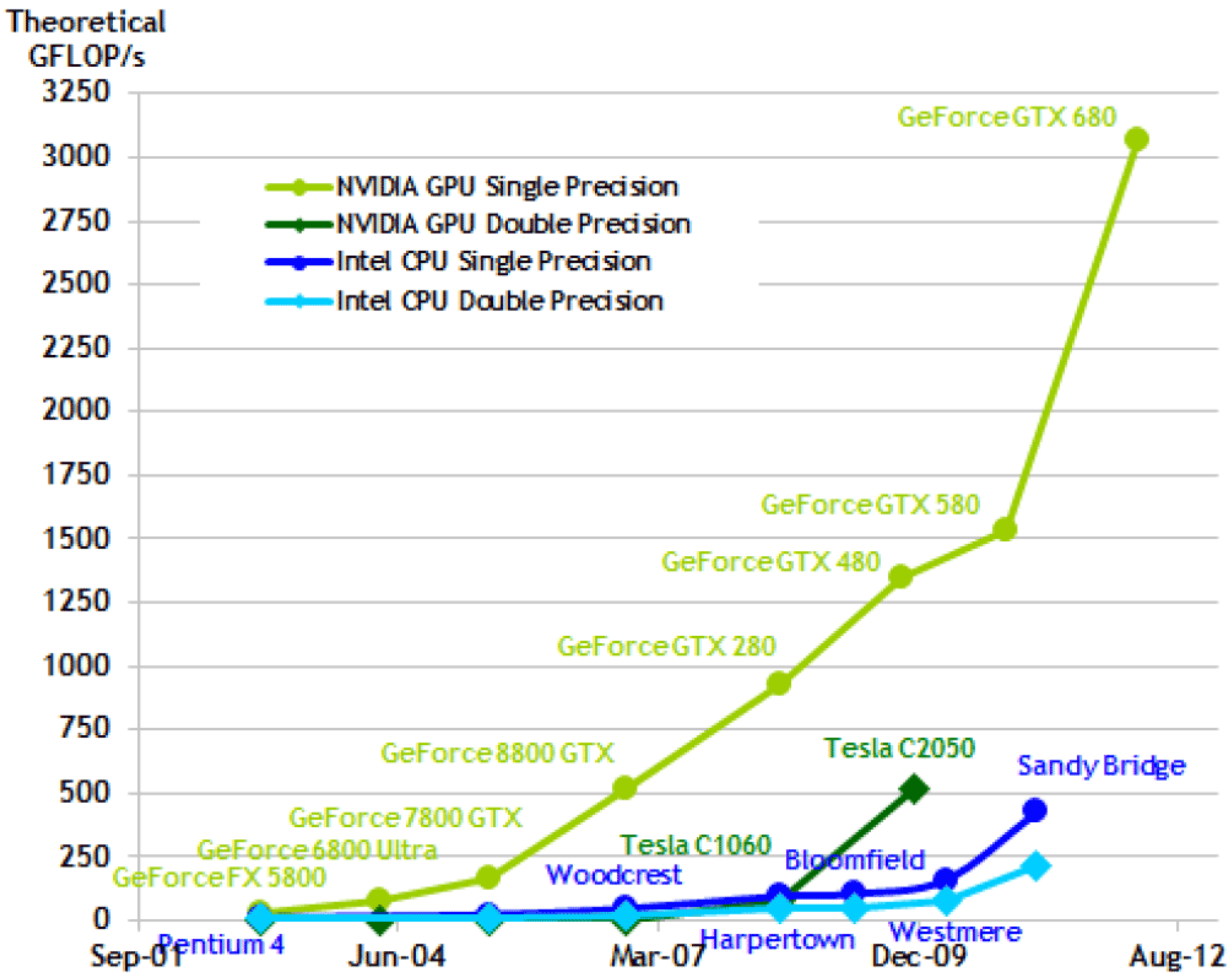


**Figure 1.** Floating Point Operations Per Second

As can be seen from the image, the latest GPU from NVIDIA (The GTX 680 $3 \times 10^{12}$ at the time of this writing) can theoretically perform 3 Trillion () Floating Point Operations per Second (or 3 teraFLOPS)[1].

The GPU is also capable of transferring large amounts of data through the AGP bus. The image below shows the memory bandwidth in GB/s of the latest NVIDIA GPU compared to the latest desktop CPUs from Intel[1].

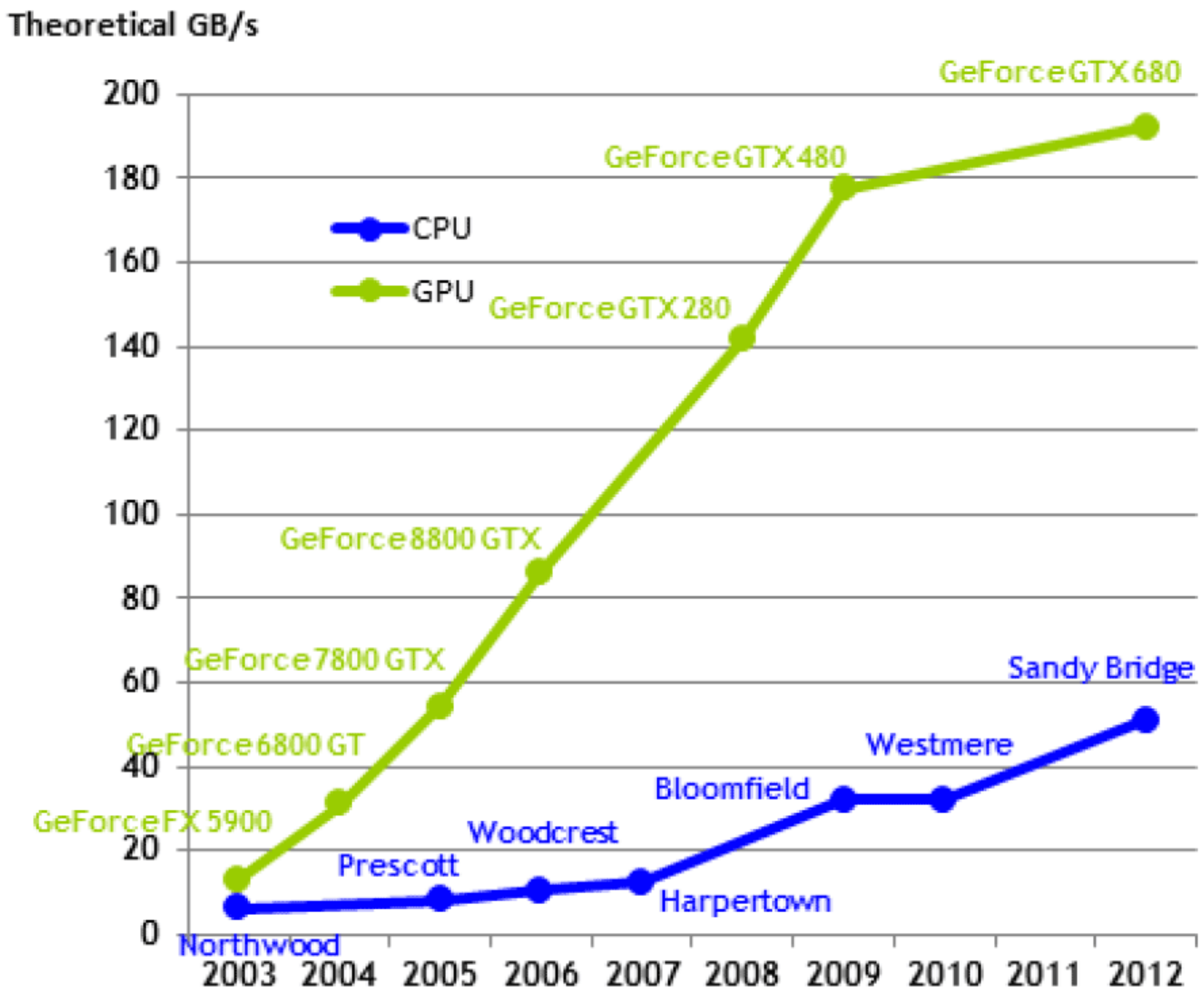


**Figure 2.** Memory Bandwidth

In this article, I will introduce the latest GPU architecture from NVIDIA: **Kepler**. I will also introduce the CUDA threading model and demonstrate how you can execute a CUDA kernel in a C++ application. I will also introduce the CUDA memory model and I will show how you can optimize your CUDA application by making use of shared memory.

## Kepler Architecture

**Kepler** is the name given to the latest line of desktop GPUs from NVIDIA. It is currently NVIDIA's flagship GPU replacing the **Fermi** architecture.

The Kepler GPU consits of 7.1 billion transistors[2] making it the fastest and most complex microprocessor ever built.

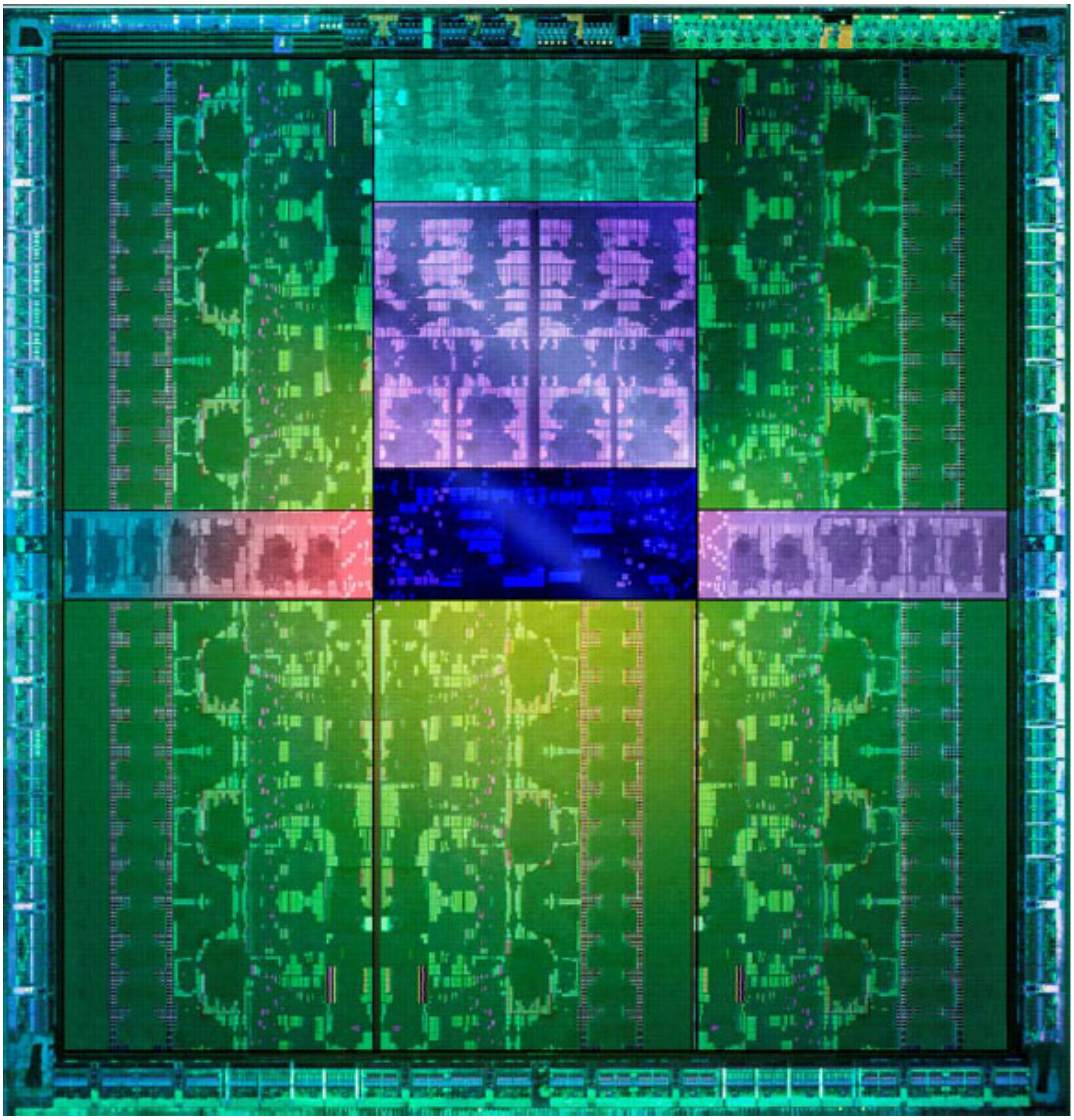


**Figure 3.** Kepler GK110 Die

Despite it's huge transistor count, the Kepler GPU is much more power efficient than its predecessor delivering up to 3x the performance per watt of the Fermi architecture[2].

The Kepler GPU was designed to be the highest performing GPU in the world. The Kepler GK110 consists of 15 **SMX** (streaming multiprocessor) units and six 64-bit memory controllers[2] as shown in the image below.

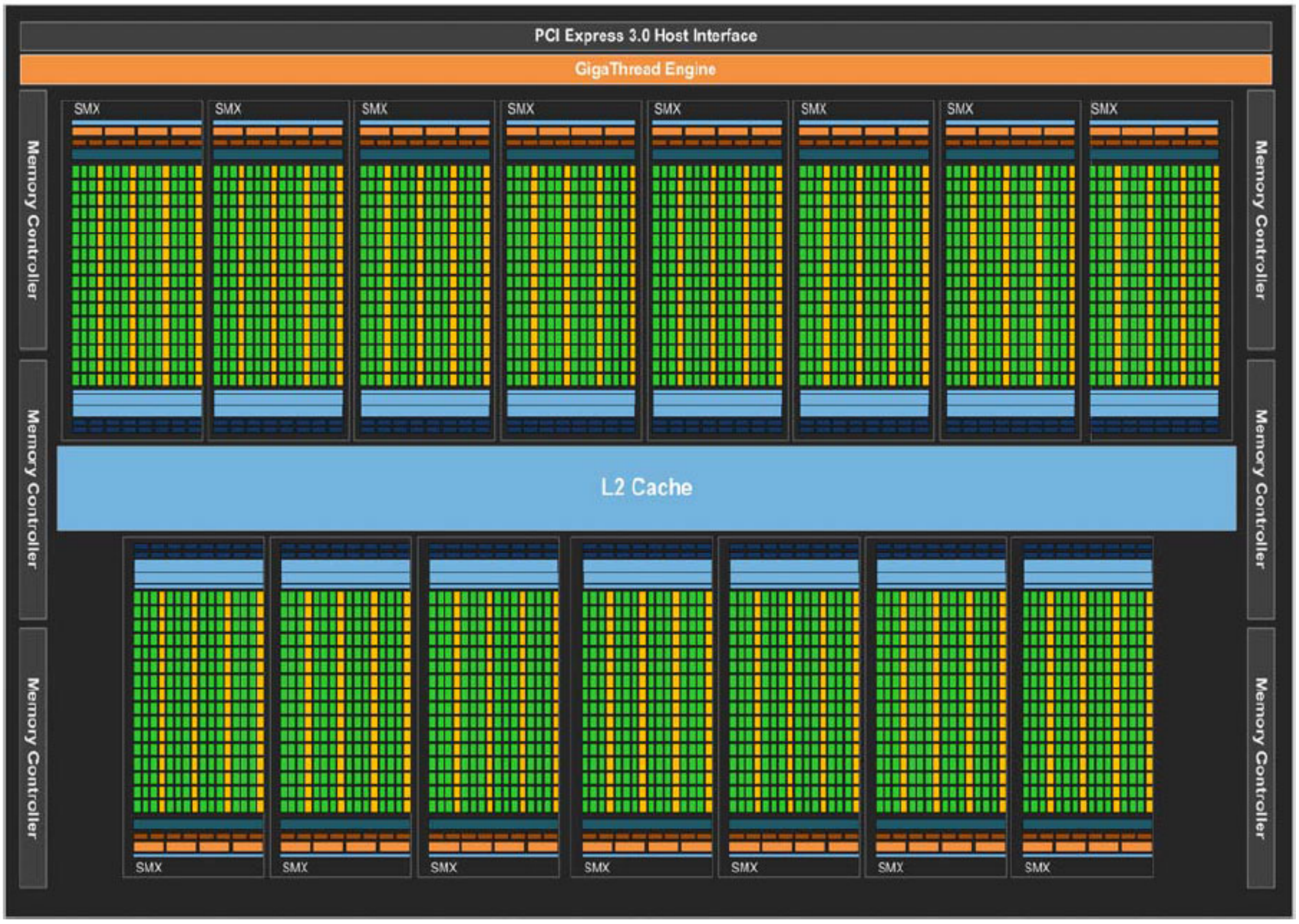


**Figure 4.** Kepler Architecture

If we zoom into a single SMX unit, we see that each SMX unit consists of 192 single-precision CUDA cores, 64 double-precision units, 32 special function units (SFU), and 32 load/store units (LD/ST).

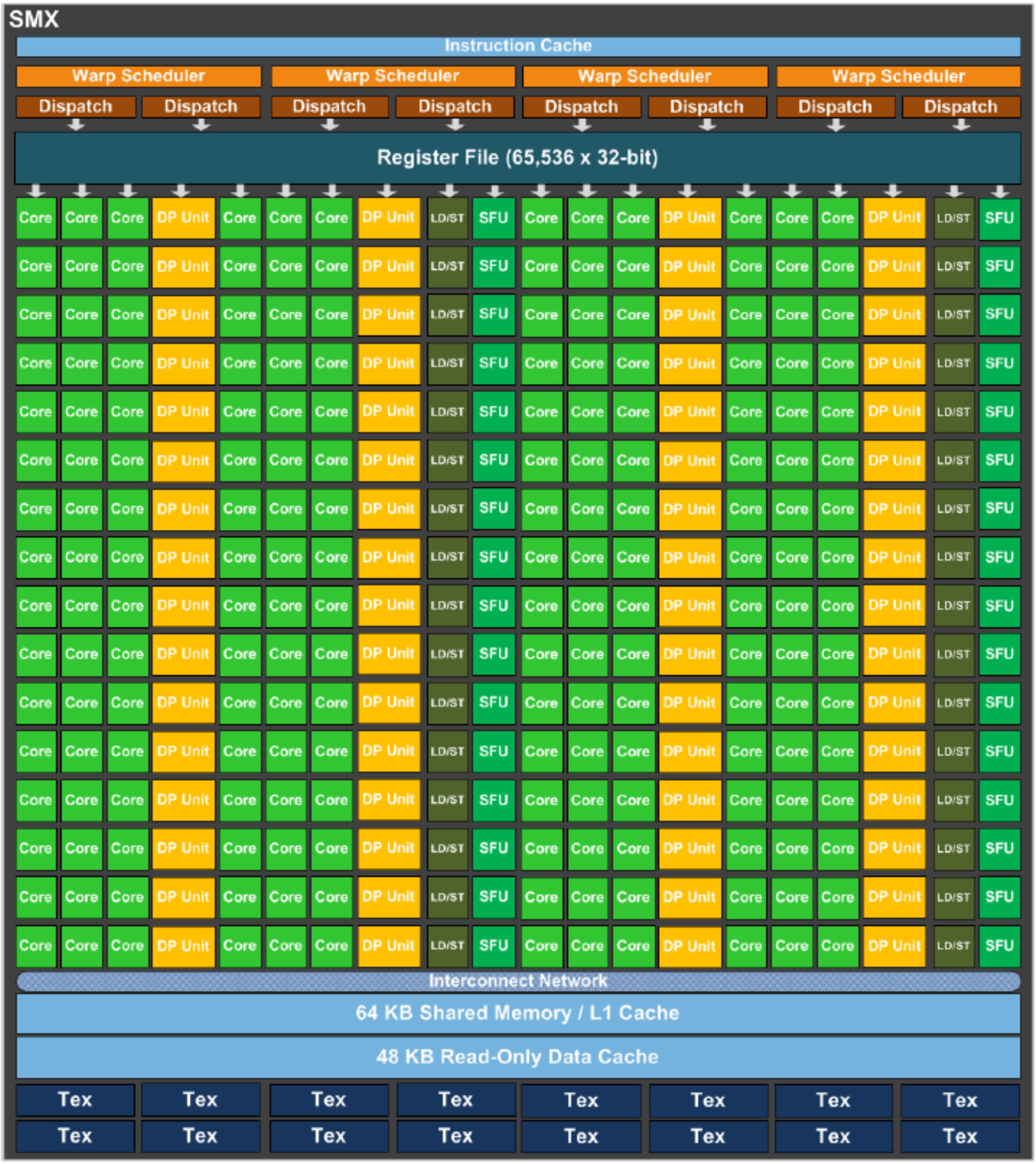

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

**Figure 5.** Kepler Streaming Multiprocessor (SMX)

The 192 single-precision CUDA cores each contain a single-precision floating-point unit (**FPU**) as well as a 32-bit integer arithmetic logic unit (ALU).

Each SMX supports 64 KB of shared memory, and 48 KB of read-only data cache. The shared memory and the data cache are accessible to all threads executing on the same streaming multiprocessor. Access to these memory areas is highly optimized and should be favored over accessing memory in global DRAM.

The SMX will schedule 32 threads in a group called a **warp**. Using compute capability 3.5, the GK110 GPU can schedule 64 warps per SMX for a total of 2,048 threads that can be resident in a single SMX at a time (not all threads will be active at the same time as we will see in the section describing the threading model).

Each SMX has four warp schedulers and eight instruction dispatch units (two dispatch units per warp scheduler) allowing four warps to be issued and executed concurrently on the streaming multiprocessor[2].

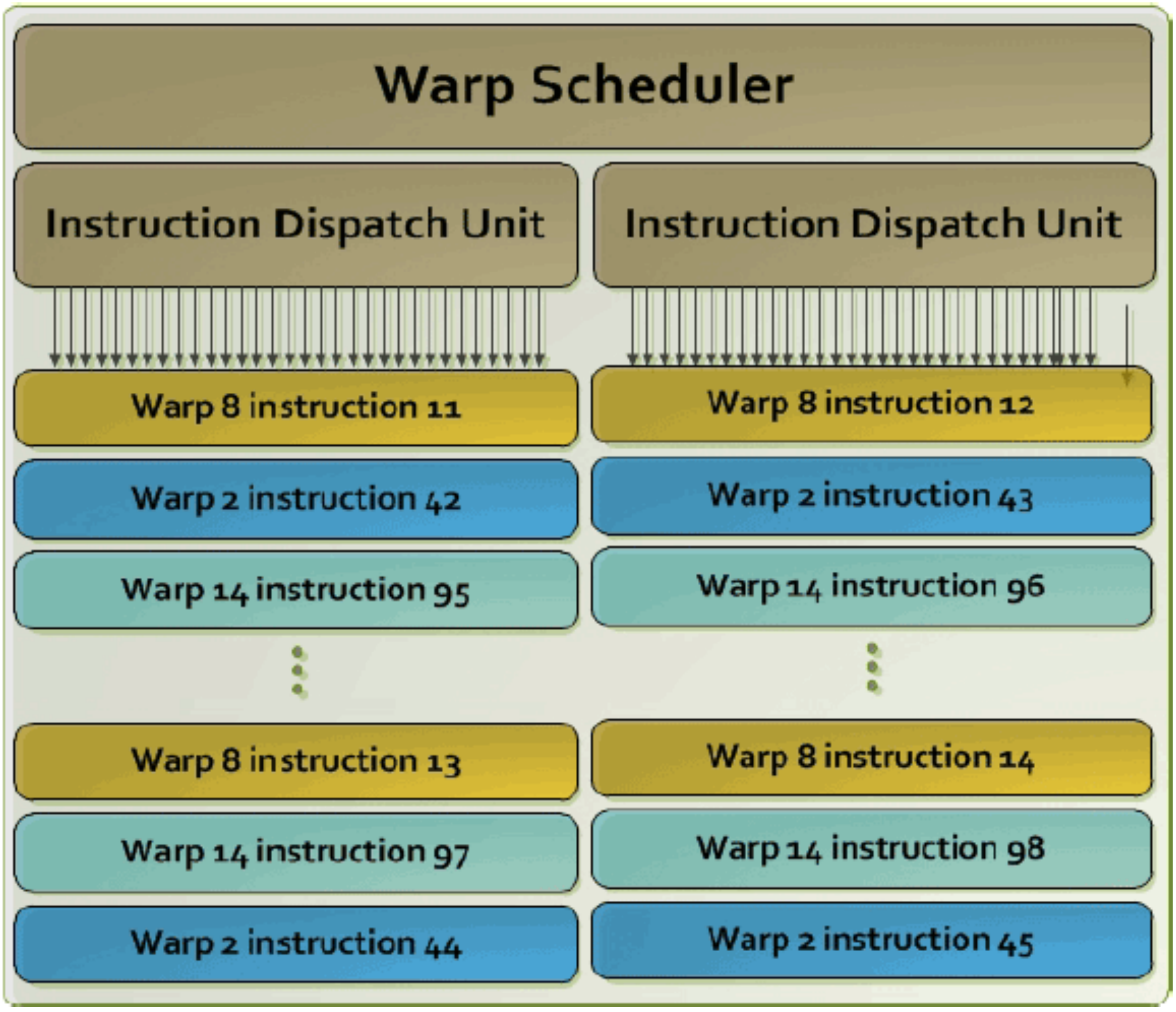


**Figure 6.** Warp Scheduler

## Dynamic Parallelism

The GK110 GPU supports a feature called **Dynamic Parallelism**. Dynamic Parallelism allows the GPU to create new work for itself by creating new kernels as they are needed without the intervention of the CPU.

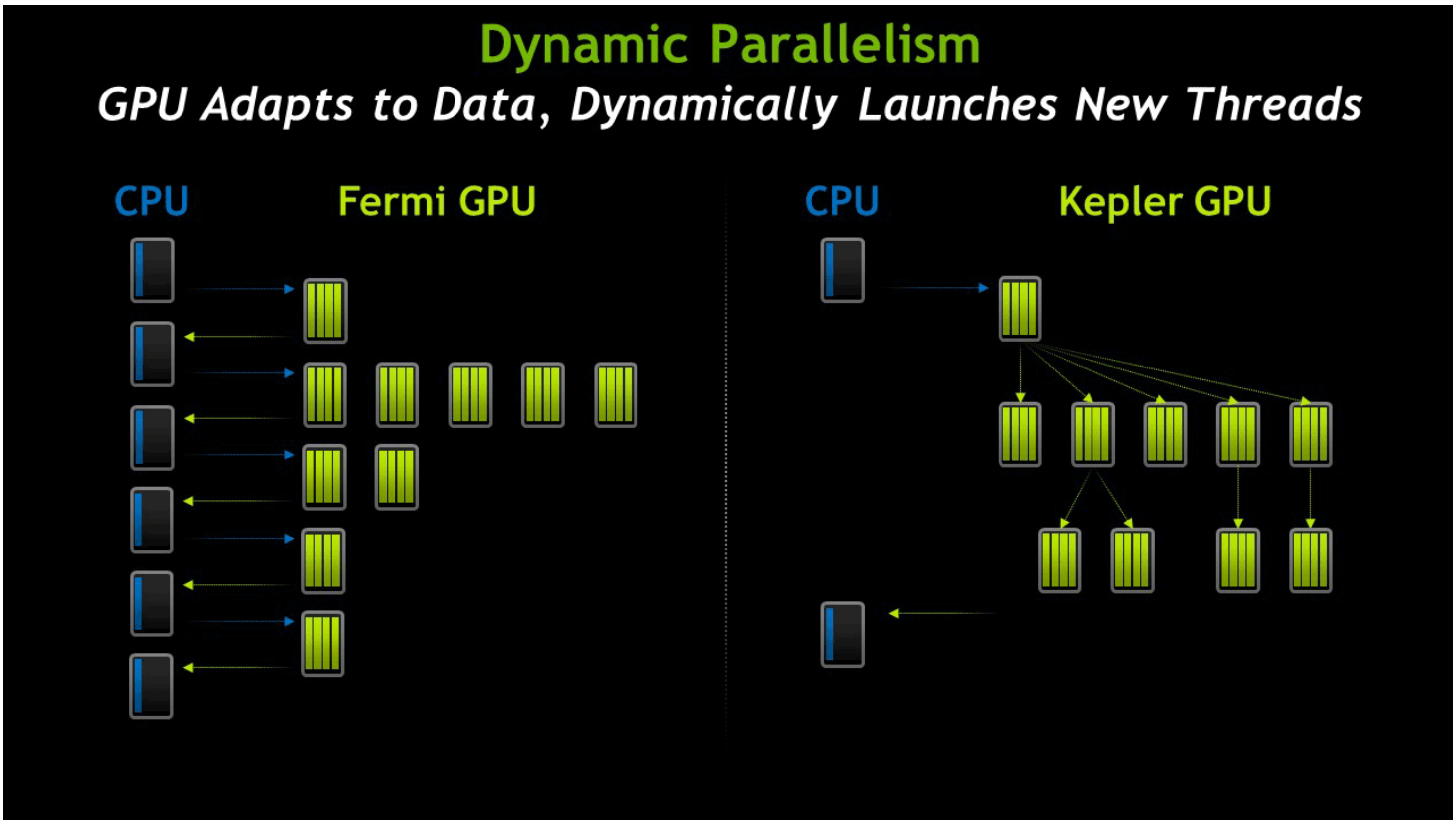


**Figure 7.** Dynamic Parallelism

As can be seen from the image, on the left, the Fermi GPU requires the CPU to execute kernels on the GPU. On the right side of the image, the Kepler GPU is capable of launching kernels from within a kernel itself. No intervention from the CPU is required.

This allows the GPU kernel to be more adaptive to dynamic branching and recursive algorithms which has some impact on the way we can implement certain functions on the GPU (such as Ray Tracing, Path Tracing and other rasterization techniques).

Dynamic Parallelism also allows the programmer to better load-balance their GPU based application. Threads can by dynamically launched based on the amount of work that needs to be performed in a particular region of the grid domain. In this case, the initial compute grid can be very coarse and the kernel can dynamically refine the grid size depending on the amount of work that needs to be performed.

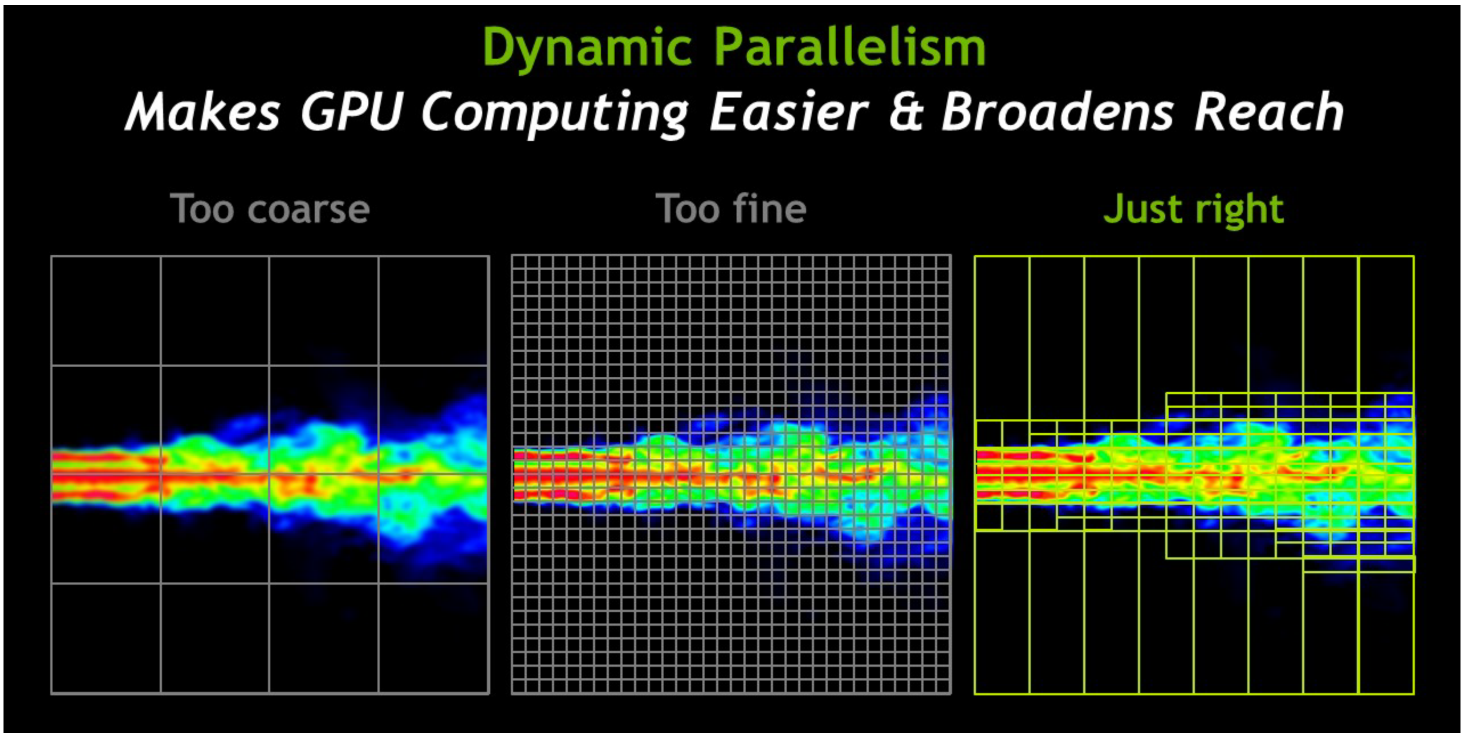


**Figure 8.** Dynamic Parallelism

As can be seen from the image, the left grid granularity is too coarse to produce an accurate simulation. The grid in the center is too fine and many kernels are not performing any actual work. On the right image we see that using dynamic parallelism, the grid can be dynamically refined to produce just the right balance between granularity and workload.

## Hyper-Q

The Fermi architecture relied on a single hardware work queue to schedule work from multiple streams. This resulted in false intra-stream dependencies requiring dependent kernels within one stream to complete before additional kernels in a separate stream could be executed[2].

The Kepler GK110 resolves this false intra-stream dependency with the introduction of the **Hyper-Q** feature. Hyper-Q increases the total number of hardware work-queues to 32 compared to the single work-queue of the Fermi architecture.

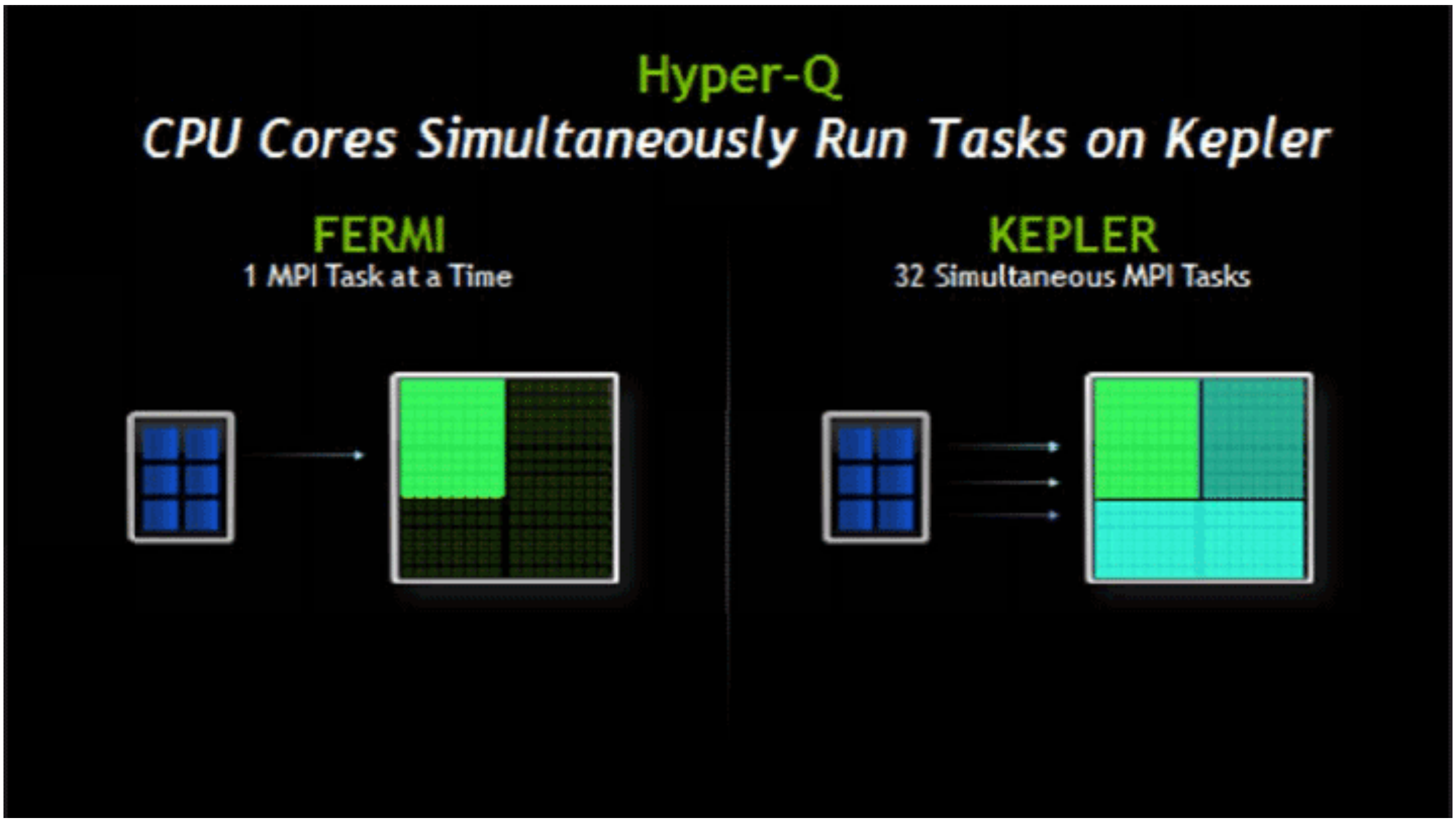

Hyper-Q
CPU Cores Simultaneously Run Tasks on Kepler
FERMI
1 MPI Task at a Time
KEPLER
32 Simultaneous MPI Tasks

**Figure 9.** Hyper-Q

CUDA applications that utilize multiple streams will immeditaly benifit from the multiple hardware work queues offered by the Hyper-Q feature. These stream intensive applications can see a potential increase in performance of up to 32x[2].

## Grid Management Unit

In order to facilitate the Dynamic Parallelism feature introduced in the GK110 GPU a new **Grid Managment Unit** (GMU) needed to be designed. In the previous Fermi architecture, grids were passed to the **CUDA Work Distributor** (CWD) directly form the stream queue. Since it is now possible to execute more kernels directly in a running CUDA kernel, a bi-directional communication link is required from the SMX to the CWD via the GMU.

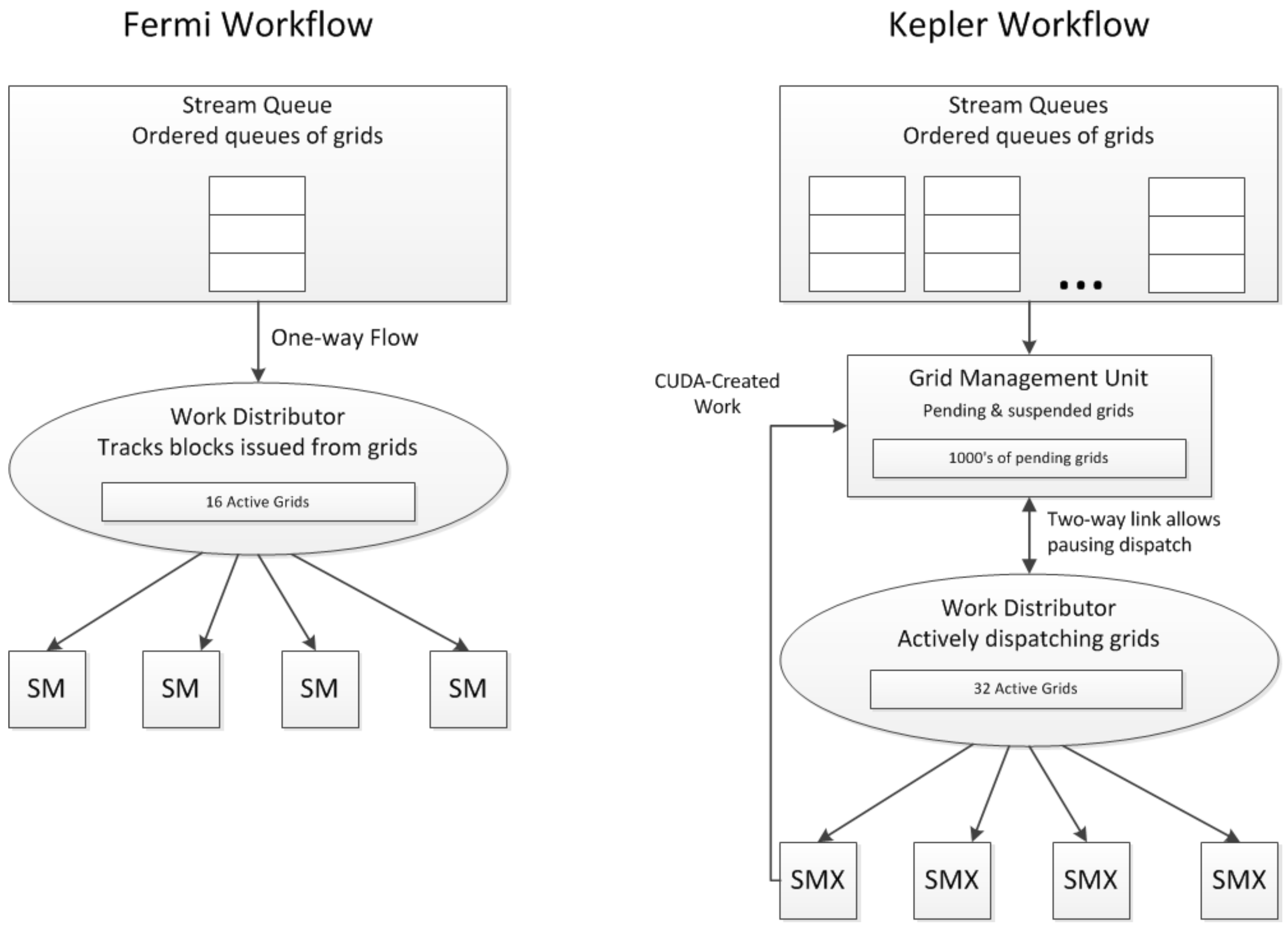


**Figure 10.** Grid Management Unit

## NVIDIA GPUDirect

The Kepler GK110 supports the **Remote Direct Memory Access** (RDMA) feature in NVIDIA GPUDirect[2]. GPUDirect allows data to be transferred directly from one GPU to another via 3rd-party devices such as InfiniBand (IB), Network Interface Cards (NIC), and Solid-State disc drives (SSD).

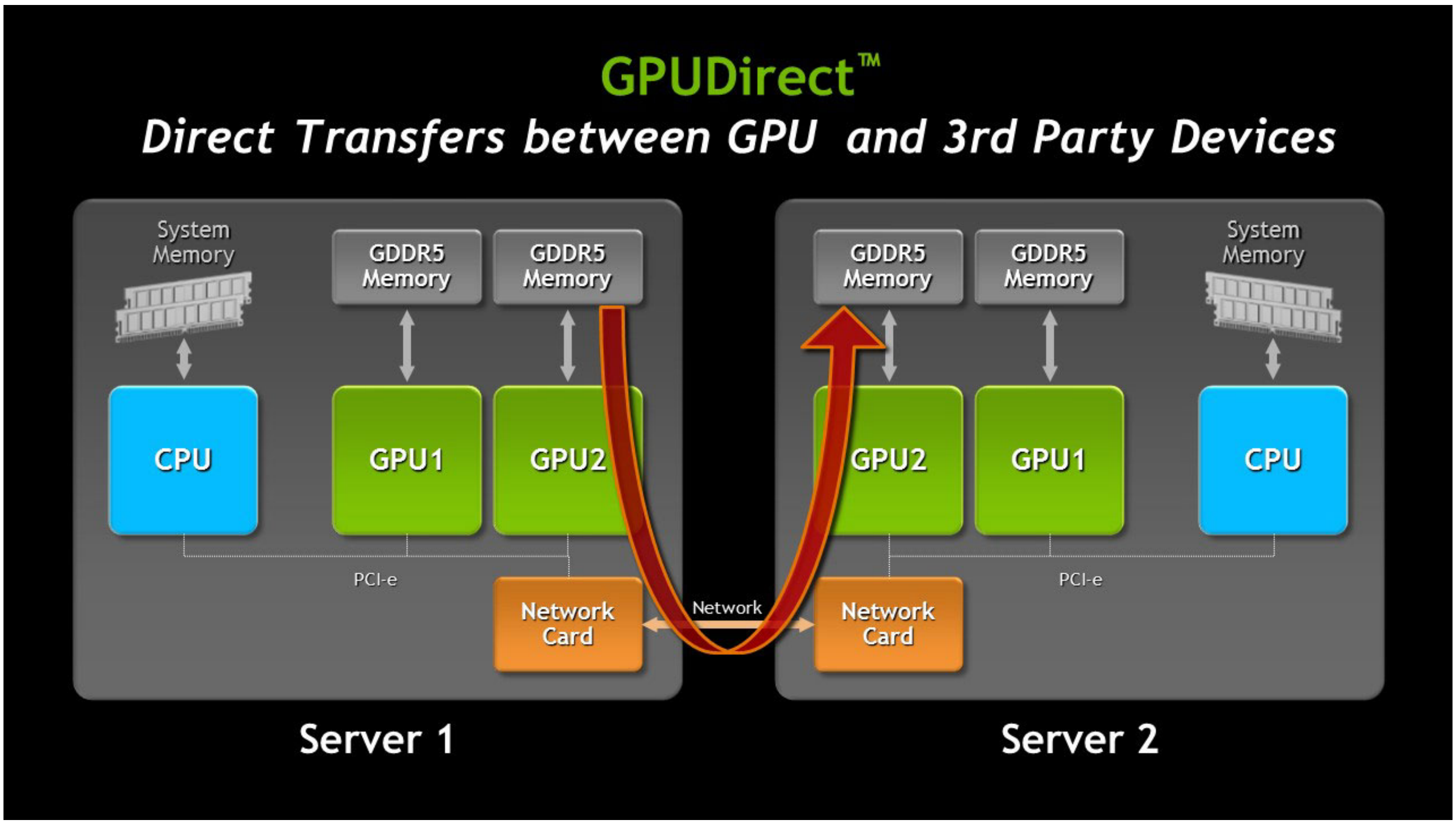


**Figure 11.** GPUDirect

## Getting Started with CUDA

In this article, I will use Visual Studio 2010 to create a CUDA enabled application. The settings and configurations for Visual Studio 2008 will be similar and you should be able to follow along even if you have not yet upgraded to VS2010.

### System Requirements

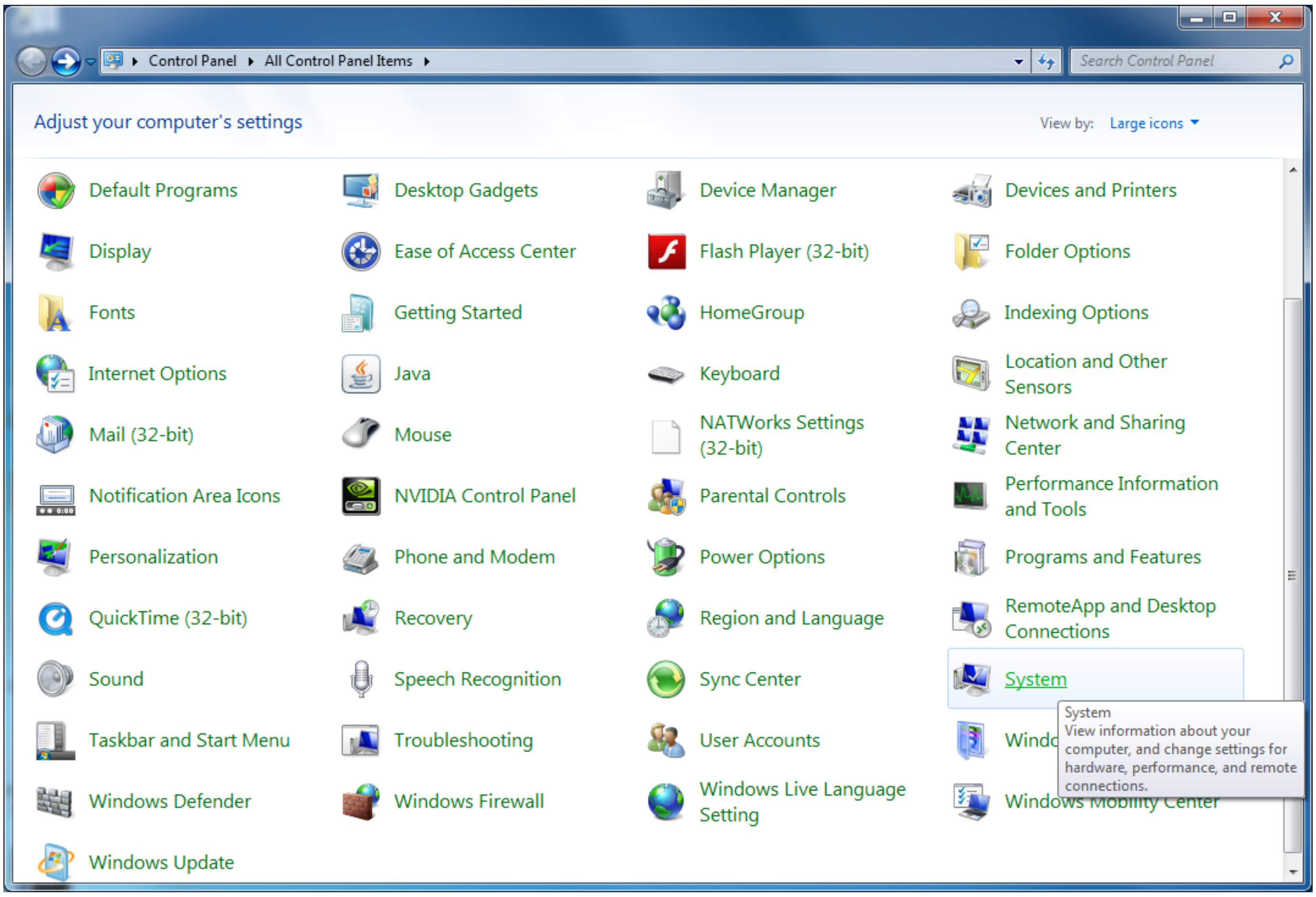
Before you can run a CUDA program, you must make sure that your system meets the minimum requirements.

- CUDA-capable GPU
- Microsoft Windows XP, Vista, 7, or 8 or Windows Server 2003 or 2008
- NVIDIA CUDA Toolkit
- Microsoft Visual Studio 2008 or 2010 or a corresponding version of Microsoft Visual C++ Express

## Verify your GPU

To verify you have a CUDA enabled GPU first check the graphics device you have installed.

1. Open the **Contol Panel** from the **Start Menu**.



Control Panel ▸ All Control Panel Items ▸

Adjust your computer's settings

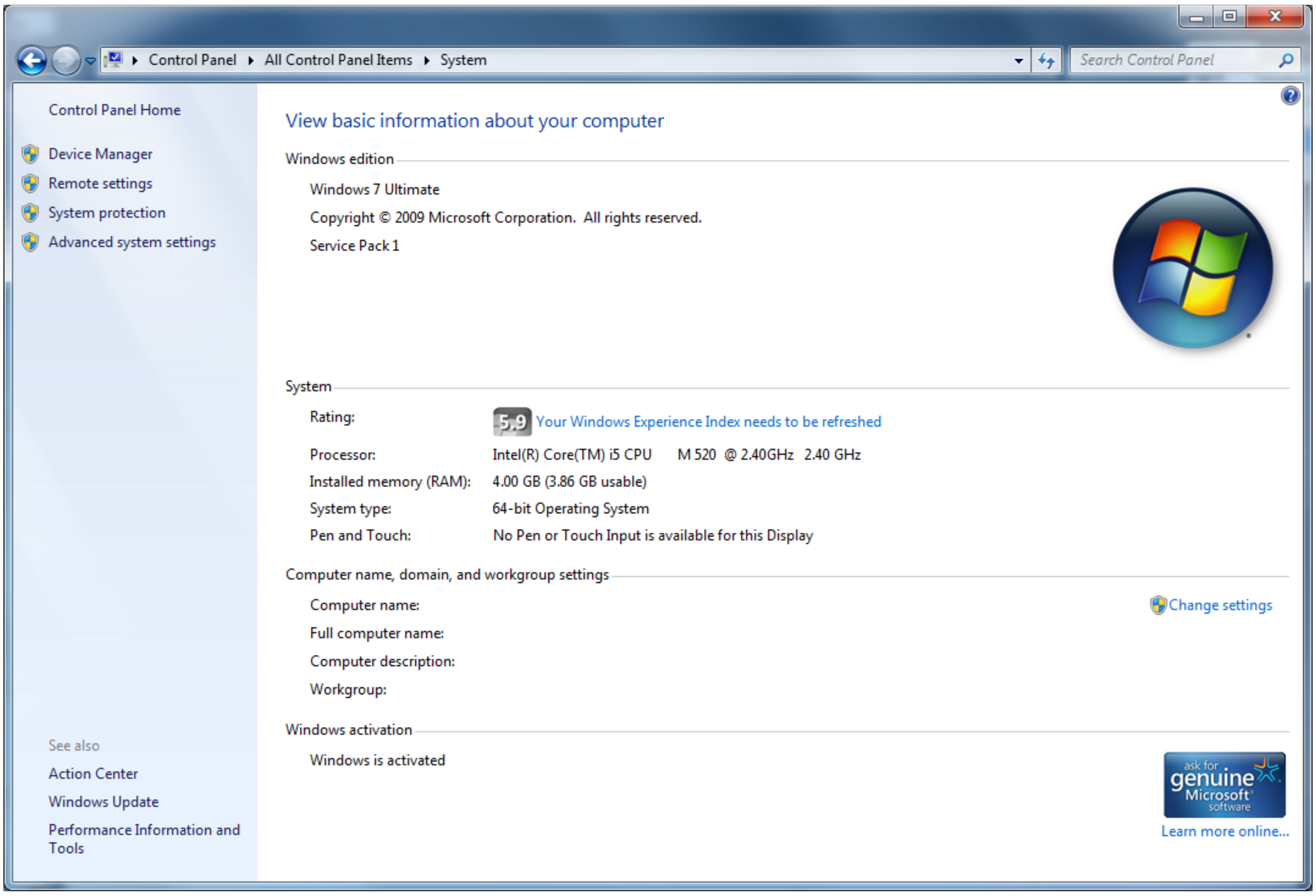
View by: Large icons

| | | | |
|---|---|---|---|
| Default Programs | Desktop Gadgets | Device Manager | Devices and Printers |
| Display | Ease of Access Center | Flash Player (32-bit) | Folder Options |
| Fonts | Getting Started | HomeGroup | Indexing Options |
| Internet Options | Java | Keyboard | Location and Other Sensors |
| Mail (32-bit) | Mouse | NATWorks Settings (32-bit) | Network and Sharing Center |
| Notification Area Icons | NVIDIA Control Panel | Parental Controls | Performance Information and Tools |
| Personalization | Phone and Modem | Power Options | Programs and Features |
| QuickTime (32-bit) | Recovery | Region and Language | RemoteApp and Desktop Connections |
| Sound | Speech Recognition | Sync Center | System |
| Taskbar and Start Menu | Troubleshooting | User Accounts | Windo |
| Windows Defender | Windows Firewall | Windows Live Language Setting | Windows Mobility Center |
| Windows Update | | | |

System
View information about your computer, and change settings for hardware, performance, and remote connections.

**Figure 12.** Control Panel

2. Double-Click the **System** applet to open the **System Control Panel**.
3. In Windows XP, click on the **Hardware** tab then click the **Device Manager** button. In Windows 7 click the **Device Manager** link.



View basic information about your computer

Windows edition

Windows 7 Ultimate
Copyright © 2009 Microsoft Corporation. All rights reserved.
Service Pack 1

System

| | |
|---|---|
| Rating: | 5.9 Your Windows Experience Index needs to be refreshed |
| Processor: | Intel(R) Core(TM) i5 CPU M 520 @ 2.40GHz 2.40 GHz |
| Installed memory (RAM): | 4.00 GB (3.86 GB usable) |
| System type: | 64-bit Operating System |
| Pen and Touch: | No Pen or Touch Input is available for this Display |

Computer name, domain, and workgroup settings

Computer name:
Full computer name:
Computer description:
Workgroup:

Windows activation

Windows is activated

**Figure 13.** System Manager

4. In the Device Manager window that appears, expand the **Display Adapters** node in the device tree.

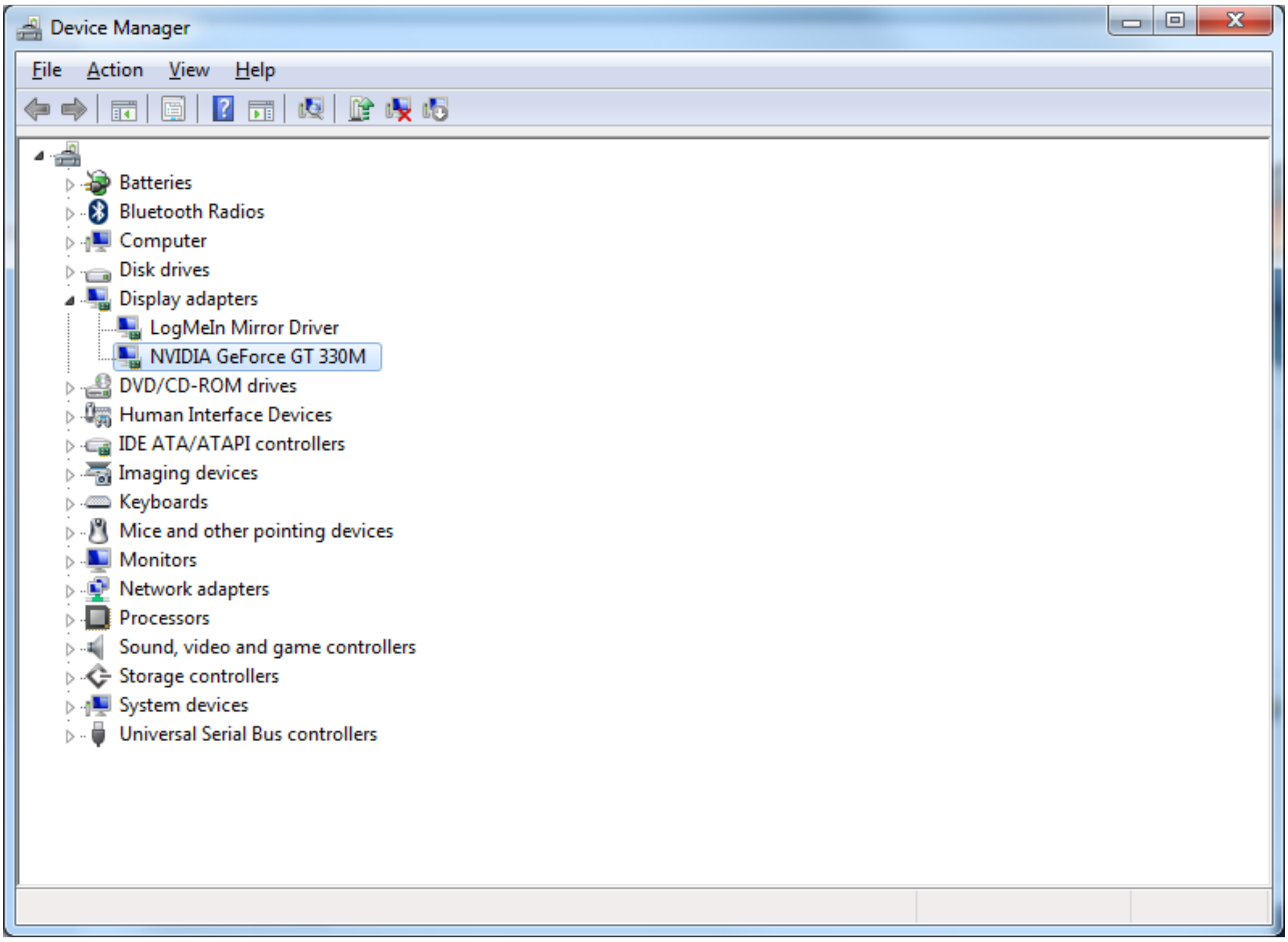


**Figure 14.** Device Manager

If your device is listed at https://developer.nvidia.com/cuda-gpus then you have a CUDA-capable GPU.

## Install CUDA

Download and install the latest NVIDIA CUDA Toolkit. The CUDA Toolkit is available at https://developer.nvidia.com/cuda-downloads.

At the time of this writing, the latest version of the CUDA toolkit is CUDA 5.0 Production Release.

The CUDA Toolkit contains the drivers and tools needed to create, build and run a CUDA application as well as libraries, header files, and CUDA samples source code and other resource[3].

By default, the CUDA toolkit is installed to **C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v#.#**, where #.# refers to the CUDA version you have installed. For the CUDA 5.0 toolkit, the complete path to the CUDA installation will be **C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v5.0**.

The installation will include the following directories:

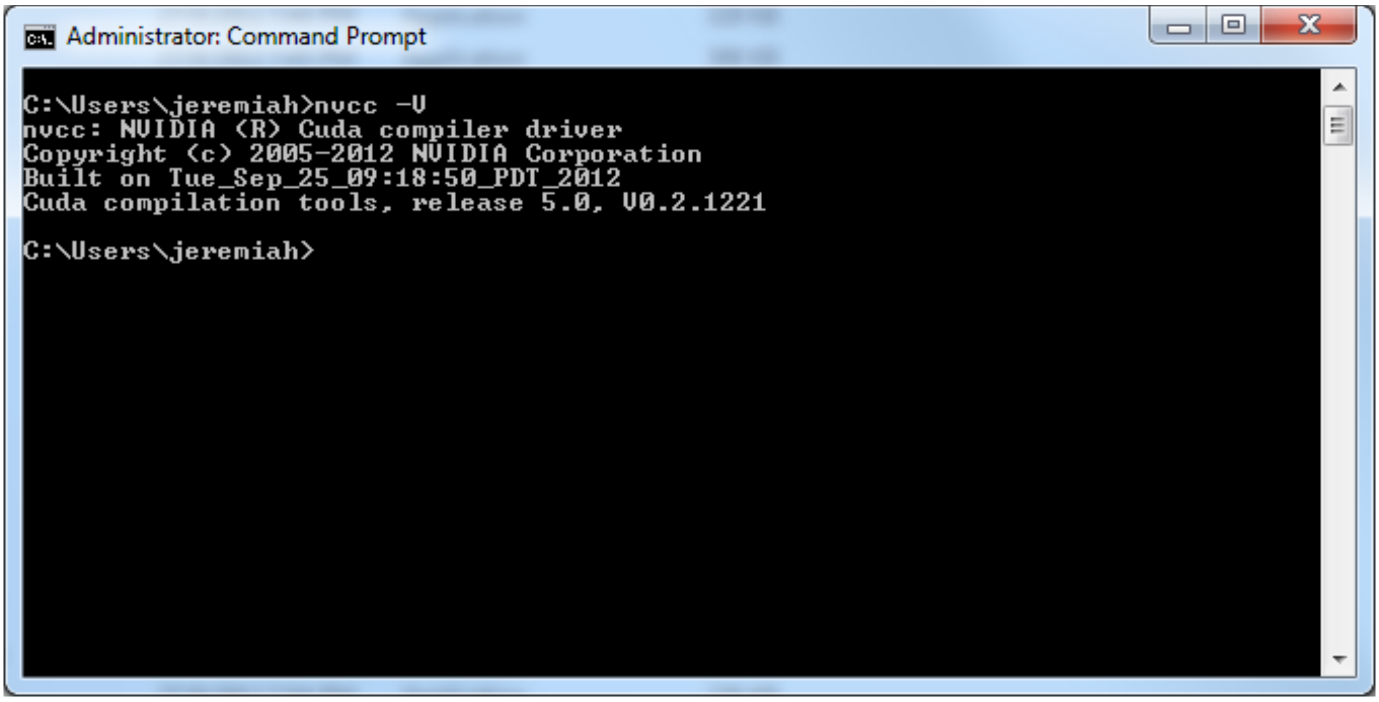
- **bin**: This folder contains the CUDA compiler and runtime libraries (DLLs)
- **include**: The **C** header files that are needed to compile your CUDA programs.
- **lib**: The library files that are needed to link your CUDA programs.
- **doc**: This directory contains the documentation for the CUDA Toolkit such as the **CUDA C Programming Guide**, the **CUDA C Best Practices Guide** and the documentation for the different CUDA libraries that are available in the Toolkit.

The CUDA Samples contain sample source code and projects for Visual Studio 2008 and Visual Studio 2010. On Windows XP, the samples can be found in **C:\Document and Settings\All Users\Application Data\NVIDIA Corporation\CUDA Samples\v#.#** and for Windows Vista, Windows 7, and Windows Server 2008, the samples can be found at **C:\ProgramData\NVIDIA Corporation\CUDA Samples\v#.#** where #.# is the installed CUDA version.

## Verify the Installation

Before you start creating a CUDA application, it is important to verify that your CUDA installation is working correctly.

1. Open a **Command Prompt** window by going to **Start Menu > All Programs > Accessories > Command Prompt**



```
C:\Users\jeremiah>nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2012 NVIDIA Corporation
Built on Tue_Sep_25_09:18:50_PDT_2012
Cuda compilation tools, release 5.0, V0.2.1221

C:\Users\jeremiah>
```

**Figure 15.** Command Prompt

2. In the **Command Prompt** window type:

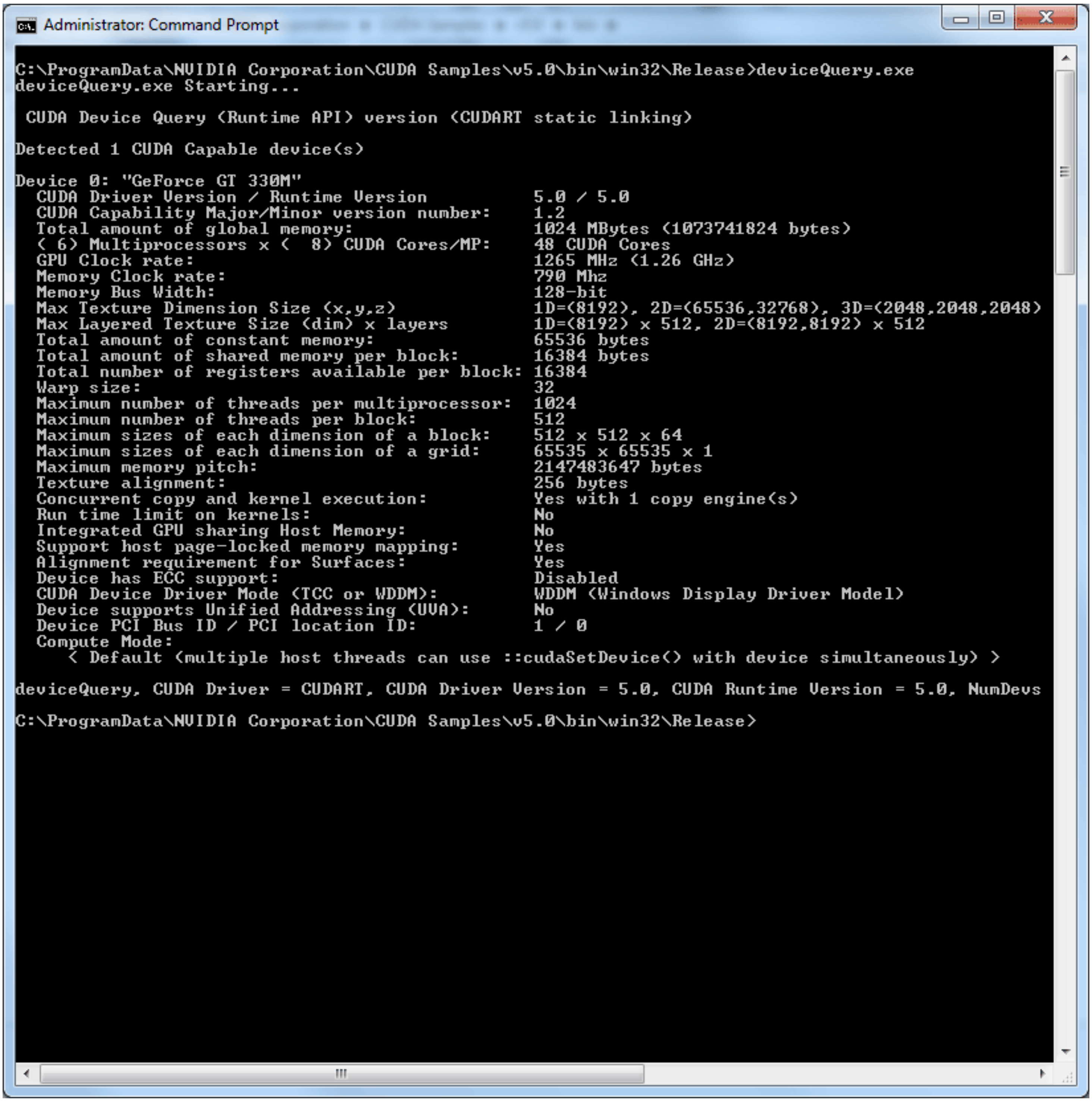
   ```
   nvcc -V
   ```

   You should see something similar to what is shown in the Command Prompt screenshot above. The output may differ slightly depending on the version of the CUDA Toolkit you installed but you should not get an error.

## Run Compiled Sample

The CUDA Toolkit comes with both the source code and compiled executable for the Toolkit samples. On Windows XP the compiled samples can be found at **C:\Document and Settings\All Users\Application Data\NVIDIA Corporation\CUDA Samples\v#.#\bin\win32\Release\** and on Windows 7, Windows 8, Windows Server 2003, and Windows Server 2008 the compiled samples can be found at **C:\ProgramData\NVIDIA Corporation\CUDA Samples\v#.#\bin\win32\Release**. On a 64-bit version of Windows, you can replace the **win32** with **win64** to run the 64-bit version of the samples.

Try to run the **deviceQuery** sample in a **Command Prompt** window. You should see some output similar to the following image:



```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0\bin\win32\Release>deviceQuery.exe
deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GT 330M"
  CUDA Driver Version / Runtime Version          5.0 / 5.0
  CUDA Capability Major/Minor version number:    1.2
  Total amount of global memory:                 1024 MBytes (1073741824 bytes)
  ( 6) Multiprocessors x (  8) CUDA Cores/MP:    48 CUDA Cores
  GPU Clock rate:                                1265 MHz (1.26 GHz)
  Memory Clock rate:                             790 Mhz
  Memory Bus Width:                              128-bit
  Max Texture Dimension Size (x,y,z)             1D=(8192), 2D=(65536,32768), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(8192) x 512, 2D=(8192,8192) x 512
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 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):      No
  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 = 5.0, CUDA Runtime Version = 5.0, NumDevs

C:\ProgramData\NVIDIA Corporation\CUDA Samples\v5.0\bin\win32\Release>
```

**Figure 16.** deviceQuery

Of course the output generated on your system will be different than this (unless you also have a GeForce GT 330M mobile GPU). Of course, the important thing is that your device(s) is(are) found and the device information is displayed without any errors.

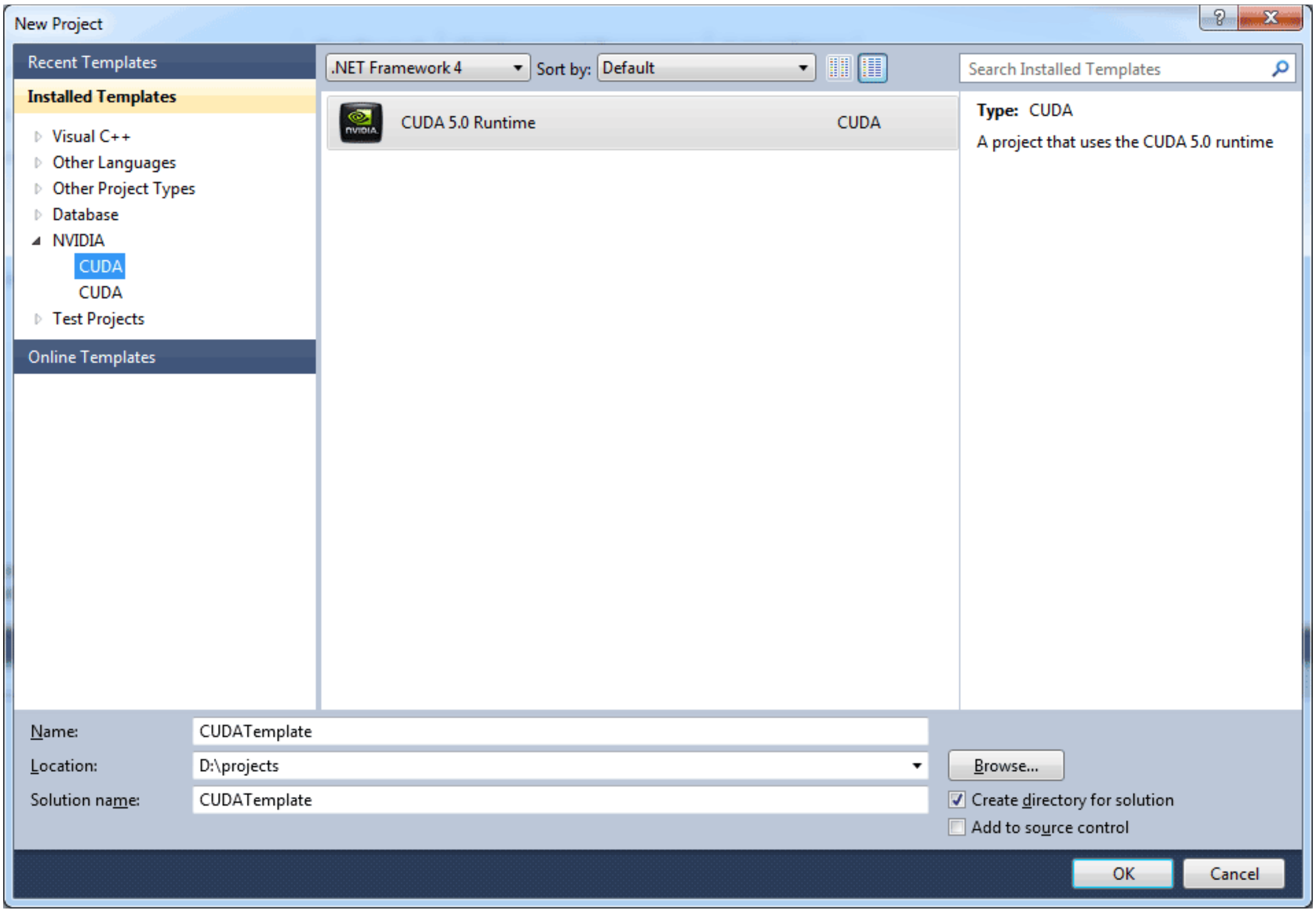
## Creating your First Project

For this article, I will create a CUDA application using Microsoft Visual Studio 2010. If you are still using Microsoft Visual Studio 2008 the steps will be very similar and you should still be able to follow along.

Open your Visual Studio IDE and create a new project.

As of CUDA Toolkit 5.0, Visual Studio project templates will be available that can be used to quickly create a project that is ready for creating a CUDA enabled application. Previous to CUDA Toolkit 5.0, Visual Studio project templates were only available when you installed **NVIDIA Nsight Visual Studio Edition**.

In the **New Project** dialog box, select **NVIDIA > CUDA** from the **Installed Templates** pane. In the right pane, select the **CUDA 5.0 Runtime** template.



Type: CUDA
A project that uses the CUDA 5.0 runtime

Name: CUDATemplate
Location: D:\projects
Solution name: CUDATemplate
[✓] Create directory for solution
[ ] Add to source control

**Figure 17.** New Project Dialog

Give your project a meaningful name such as "**CUDATemplate**" or something similar.

Click **OK** to create a new project.

This will create a new Visual Studio C++ project with a single CUDA source file called **kernel.cu**

You should be able to compile and run this sample already at this point to confirm it is working. You should get the following output:

```
{1,2,3,4,5} + {10,20,30,40,50} = {11,22,33,44,55}
```

If you got any errors or something went wrong, then you should check that do have a CUDA enabled GPU and that you installed the CUDA Toolkit prior to installing Visual Studio 2010. Follow the steps in the previous sections again and make sure you did everything correctly.

Using the Visual Studio project template for the **CUDA 5.0 Runtime** will automatically configure the build settings necessary to compile a CUDA enabled application. If you want to know how to add the configure necessary to build CUDA source files to an existing C/C++ project, then you can refer to my previous article titled Introduction to CUDA that I wrote last year. That article focuses on CUDA 4.0 using Visual Studio 2008 but the steps are almost identical for CUDA 5.0 using Visual Studio 2010.

## Threading Model

The CUDA threading model describes how a kernel is executed on the GPU.

### CUDA Threads

Each kernel function is executed in a grid of threads. This grid is divided into blocks also known as thread blocks and each block is further divided into threads.

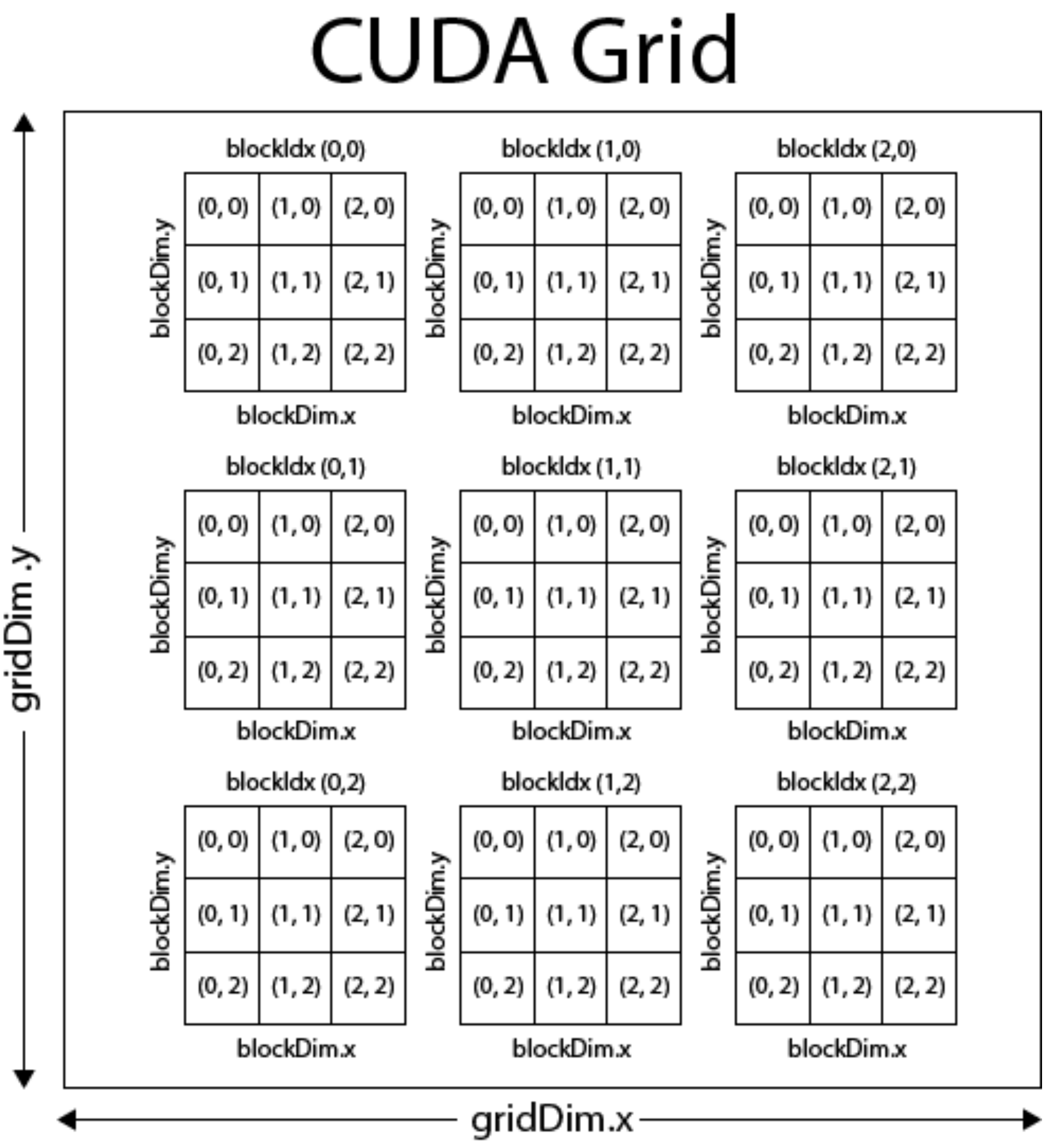


**Figure 18.** Cuda Execution Model

In the image above we see that this example grid is divided into nine thread blocks (3×3), each thread block consists of 9 threads (3×3) for a total of 81 threads for the kernel grid.

This image only shows 2-dimensional grid, but if the graphics device supports compute capability 2.0 or higher, then the grid of thread blocks can actually be partitioned into 1, 2 or 3 dimensions, otherwise if the device supports compute capability 1.x, then thread blocks can be partitioned into 1, or 2 dimensions (in this case, then the 3rd dimension should always be set to 1).

The thread block is partitioned into individual threads and for all compute capabilities, threads in a block can be partitioned into 1, 2, or 3 dimensions. The maximum number of threads that can be assigned to a thread block is 512 for devices with compute capability 1.x and 1024 threads for devices that support compute capability 2.0 and higher.

**Table 1.** Threading Compute Capability

| **Technical Specifications** | **1.0** | **1.1** | **1.2** | **1.3** | **2.x** | **3.0** | **3.5** |
|---|---|---|---|---|---|---|---|
| Maximum dimensionality of a grid of thread blocks. | 2 | | | | 3 | | |
| Maximum x-, dimension of a grid of thread blocks. | 65535 | | | | | $2^{31}$-1 | |
| Maximum y- or z-dimension of a grid of thread blocks. | 65535 | | | | | | |
| Maximum dimensionality of a thread block. | 3 | | | | | | |
| Maximum x- or y-dimension of a block. | 512 | | | | 1024 | | |
| Maximum z-dimension of a block. | 64 | | | | | | |
| Maximum number of threads per block. | 512 | | | | 1024 | | |
| Warp size. | 32 | | | | | | |
| Maximum number of resident blocks per multiprocessor. | 8 | | | | | 16 | |
| Maximum number of resident warps per multiprocessor. | 24 | | 32 | | 48 | 64 | |
| Maximum number of resident threads per multiprocessor. | 768 | | 1024 | | 1536 | 2048 | |

The number of blocks within a gird can be determined within a kernel by using the built-in variable **gridDim** and the number of threads within a block can be determined by using the built-in variable **blockDim**.

A thread block is uniquely identified in a kernel function by using the built-in variable **blockIdx** and a thread within a block is uniquely identified in a kernel function by using the built-in variable **threadIdx**.

The built-in variables **gridDim**, **blockDim**, **blockIdx**, and **threadIdx** are each 3-component structs with members x, y, z.

With a 1-D kernel, the unique thread ID within a block is the same as the x component of the threadIdx variable.

$threadID = threadIdx.x$ and the unique block ID within a grid is the same as the x component of the blockIdx variable:

$blockID = blockIdx.x$ To determine the unique thread ID in a 2-D block, you would use the following formula:

$threadID = (threadIdx.y * blockDim.x) + threadIdx.x$ and to determine the unique block ID within a 2-D grid, you would use the following formula:

$blockID = (blockIdx.y * gridDim.x) + blockIdx.x$ I'll leave it as an exercise for the reader to determine the formula to compute the unique thread ID and block ID in a 3D grid.

## Matrix Addition Example

Let's take a look at an example kernel that one might execute.

Let's assume we want to implement a kernel function that adds two matrices and stores the result in a 3rd.

The general formula for matrix addition is:

$$\begin{aligned} C &= A + B \\ c_{i,j} &= a_{i,j} + b_{i,j} \end{aligned}$$

That is, the sum of matrix **A** and matrix **B** is the sum of the components of matrix **A** and matrix **B**.

Let's first write the host version of this method that we would execute on the CPU.

MatrixAdd.cpp

```
void MatrixAddHost( float* C, float* A, float* B, unsigned int matrixDim )
{
    for( unsigned int j = 0; j < matrixDim; ++j )
    {
        for ( unsigned int i = 0; i < matrixDim; ++i )
        {
            unsigned int index = ( j * matrixDim) + i;
            C[index] = A[index] + B[index];
        }
    }
}
```

This is a pretty standard method that loops through the rows and columns of a matrix and adds the components and stores the results in a 3rd. Now let's see how we might execute this kernel on the GPU using CUDA.

First, we need to think of the problem domain. I this case, the domain is trivial: it is the components of a matrix. Since we are operating on 2-D arrays, it seems reasonable to split our domain into two dimensions; one for the rows, and another for the columns of the matrices.

We will assume that we are working on square matrices. This simplifies the problem but mathematically matrix addition only requires that the two matrices have the same number of rows and columns but does not have the requirement that the matrices must be square.

Since we know that a kernel is limited to 512 threads/block with compute capability 1.x and 1024 threads/block with compute capability 2.x and 3.x, then we know we can split our job into square thread blocks each consisting of 16×16 threads (256 threads per block) with compute capability 1.x and 32×32 threads (1024 threads per block) with compute capability 2.x and 3.x.

If we limit the size of our matrix to no larger than 16×16, then we only need a single block to compute the matrix sum and our kernel execution configuration might look something like this:

main.cpp

```
2dim3 gridDim( 1, 1, 1 );
3dim3 blockDim( matrixDim, matrixDim, 1 );
 MatrixAddDevice<<<gridDim, blockDim>>>( C, A, B, matrixDim );
```

In this simple case, the kernel grid consists of only a single block with *matrixDim* x *matrixDim* threads.

However, if we want to sum matrices larger than 512 components, then we must split our problem domain into smaller groups that can be processed in multiple blocks.

Let's assume that we want to limit our blocks to execute in 16×16 (256) threads. We can determine the number of blocks that will be required to operate on the entire array by dividing the size of the matrix dimension by the maximum number of threads per block and round-up to the nearest whole number:

$blocks = Ceiling\lceil \frac{matrixDim}{16} \rceil$ And we can determine the number of threads per block by dividing the size of the matrix dimension by the number of blocks and round-up to the nearest whole number:

$threads = Ceiling\lceil \frac{matrixDim}{blocks} \rceil$ So for example, for a **4×4** matrix, we would get

$$\begin{aligned} blocks &= \lceil \tfrac{4}{16} \rceil \\ blocks &= \lceil 0.25 \rceil \\ blocks &= 1 \end{aligned}$$

and the number of threads is computed as:

$$\begin{aligned} threads &= \lceil \tfrac{4}{1} \rceil \\ threads &= 4 \end{aligned}$$

resulting in a **1×1** grid of **4×4** thread blocks for a total of **16** threads.

Another example a **512×512** matirx, we would get:

$$\begin{aligned} blocks &= \lceil \tfrac{512}{16} \rceil \\ blocks &= \lceil 32 \rceil \\ blocks &= 32 \end{aligned}$$

and the number of threads is computed as:

$$\begin{aligned} threads &= \lceil \tfrac{512}{32} \rceil \\ threads &= 16 \end{aligned}$$

resulting in a **32×32** grid of **16×16** thread blocks for a total of **262,144** threads.

The host code to setup the kernel granularity might look like this:

main.cpp
```
size_t blocks = ceilf( matrixDim / 16.0f );
dim3 gridDim( blocks, blocks, 1 );
size_t threads = ceilf( matrixDim / (float)blocks );
dim3 blockDim( threads, threads, 1 );

MatrixAddDevice<<< gridDim, blockDim >>>( C, A, B, matrixDim );
```

You may have noticed that if the size of the matrix does not fit nicely into equally divisible blocks, then we may get more threads than are needed to process the array. It is not possible to configure a gird of thread blocks with 1 block containing less threads than the others. The only way to solve this is to execute multiple kernels – one that handles all the equally divisible blocks, and a 2nd kernel invocation that handles the partial block. The other solution to this problem is simply to ignore any of the threads that are executed outside of our problem domain which is generally the easier (and more efficient) than invoking multiple kernels (this should be profiled to be proven).

## The Matrix Addition Kernel Function

On the device, one kernel function is created for every thread in the problem domain (the matrix elements). We can use the built-in variables **gridDim**, **blockDim**, **blockIdx**, and **threadIdx**, to identify the current matrix element that the current kernel is operating on.

If we assume we have a **9×9** matrix and we split the problem domain into **3×3** blocks each consisting of **3×3** threads as shown in the CUDA Grid below, then we could compute the *i*th column and the *j*th row of the matrix with the following formula:

$$\begin{aligned} i &= (blockDim.x * blockIdx.x) + threadIdx.x \\ j &= (blockDim.y * blockIdx.y) + threadIdx.y \end{aligned}$$

So for thread **(0,0)** of block **(1,1)** of our **9×9** matrix, we would get:

$$\begin{aligned} i &= (3 * 1) + 0 \\ i &= 3 \end{aligned}$$

for the column and:

$$\begin{aligned} j &= (3 * 1) + 0 \\ j &= 3 \end{aligned}$$

for the row.

The index into the 1-D buffer that store the matrix is then computed as:

$index = (matrixDim * i) + j$ and substituting gives:

$$\begin{aligned} index &= (matrixDim * 3) + 3 \\ index &= (9 * 3) + 3 \\ index &= 30 \end{aligned}$$

Which is the correct element in the matrix. This solution assumes we are accessing the matrix in row-major order.

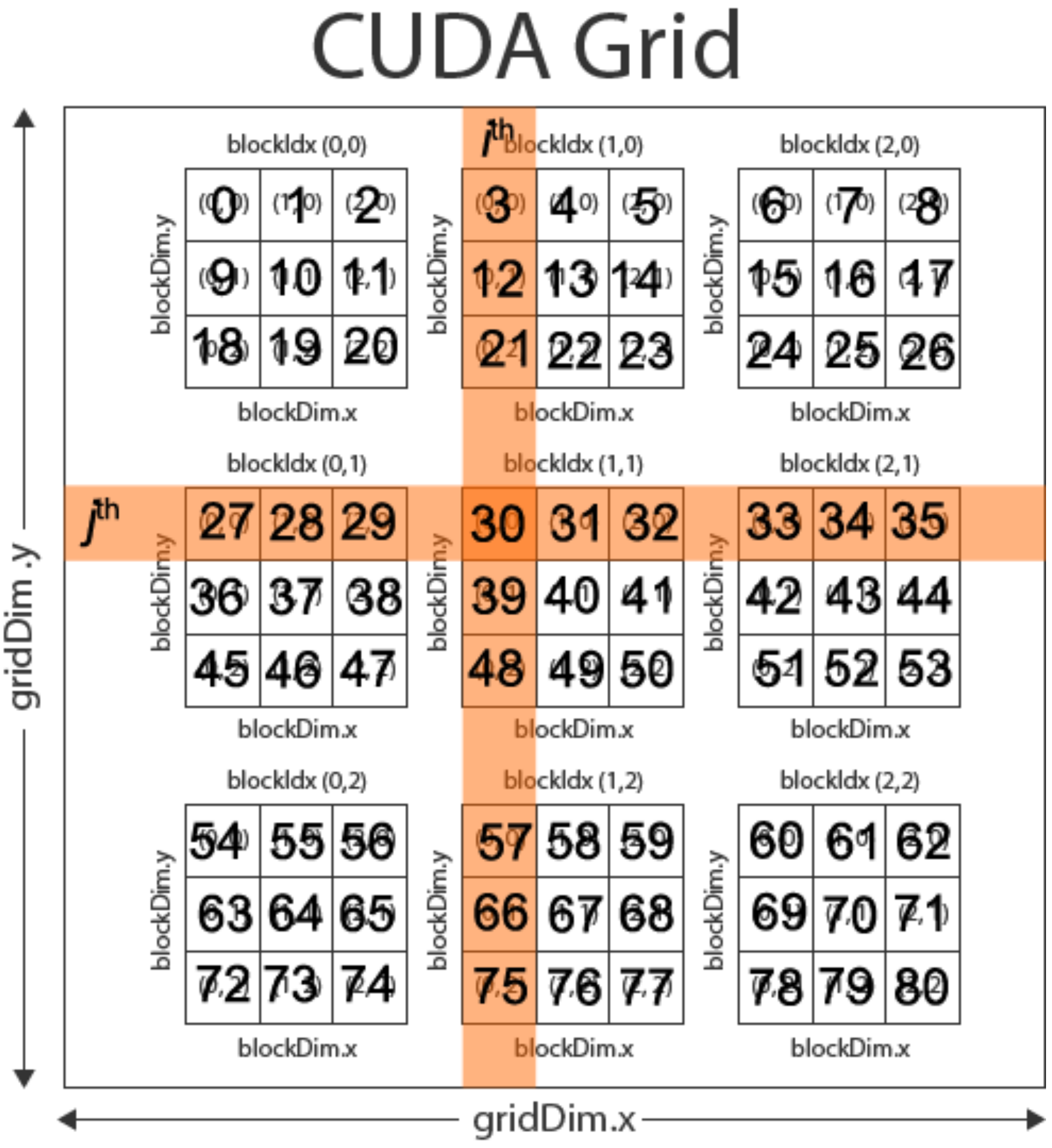


**Figure 19.** CUDA Grid Example

Let's see how we might implement this in the kernel.

MatrixAdd.cu

```
__global__ void MatrixAddDevice( float* C, float* A, float* B, unsigned int matrixDim )
{
    unsigned int column = ( blockDim.x * blockIdx.x ) + threadIdx.x;
    unsigned int row    = ( blockDim.y * blockIdx.y ) + threadIdx.y;

6       unsigned int index = ( matrixDim * row ) + column;
7       if ( index < matrixDim * matrixDim ) // prevent reading/writing array out-of-bounds.
8       {
9           C[index] = A[index] + B[index];
10      }
11  }
```

The kernel function is defined using the **__global__** declaration specifier. This specifier is used to identify a function that should execute on the device. Optionally you can also specify host functions with the **__host__** declaration specifier within a CUDA source file but this is implied if no specifier is applied to the function declaration.

On line 3, and 4 we compute the column and row of the matrix we are operating on using the formulas shown earlier.

On line 6, the 1-d index in the matrix array is computed based on the size of a single dimension of the square matrix.

We must be careful that we don't try to read or write out of the bounds of the matrix. This might happen if the size of the matrix does not fit nicely into the size of the CUDA grid (in the case of matrices whose size is not evenly divisible by 16) To protect the read and write operation, on line 7 we must check that the computed index does not exceed the size of our array.

## Thread Synchronization

CUDA provides a synchronization barrier for all threads in a block through the **__syncthreads()** method. A practical example of thread synchronization will be shown in a later article about optimization a CUDA kernel, but for now it's only important that you know this functionality exists.

Thread synchronization is only possible across all threads in a block but not across all threads running in the grid. By not allowing threads across blocks to be synchronized, CUDA enables multiple blocks to be executed on other streaming multiprocessors (SM) in any order. The queue of blocks can be distributed to any SM without having to wait for blocks from another SM to be complete. This allows the CUDA enabled applications to scale across platforms that have more SM at it's disposal, executing more blocks concurrently than another platforms with less SM's.

Thread synchronization follows strict synchronization rules. All threads in a block must hit the synchronization point or none of them must hit synchronization point.

Give the following code block:

sample.cu
```
if ( threadID % 2 == 0 )
{
    __syncthreads();
}
else
{
    __syncthreads();
}
```

will cause the threads in a block to wait indefinitely for each other because the two occurrences of **__syncthreads** are considered separate synchronization points and all threads of the same block must hit the same synchronization point, or all of them must not hit it.

## Thread Assignment

When a kernel is invoked, the CUDA runtime will distribute the blocks across the SM's on the device. With compute compatibility 1.x and 2.x a maximum of 8 blocks will be assigned to each SM and with compute compatibility 3.x a maximum of 16 blocks will be assigned to each SM as long as there are enough resources (registers, shared memory, and threads) to execute all the blocks. In the case where there are not enough resources on the SM, then the CUDA runtime will automatically assign less blocks per SM until the resource usage is below the maximum per SM.

The total number of blocks that can be executed concurrently is dependent on the device. In the case of the Fermi architecture a total of 16 SM's can concurrently handle 8 blocks for a total of 128 blocks executing concurrently on the device. Kepler devices can handle 16 thread blocks per SMX for a total of 240 thread blocks that can execute concurrently on a single device.

Both the Fermi and Kepler architecture support thread blocks consisting of at most 1024 threads. The Fermi device can support a maximum of 48 warps per SM. The Kepler architecture increases the amount of resident warps per SMX to 64.

The Fermi device can support a maximum of 1,536 resident threads (32×48) per SM. Kepler supports 2,048 threads per SMX (32×64). With 15 SMX units, the Kepler GPU can have a total of 30,720 resident threads on the device. This does not mean that every clock tick the devices is executing 30,720 instruction simultaneously (there are only 2,880 CUDA Cores on the GK110 device). In order to understand how the blocks are actually executed on the device, we must look one step further to see how the threads of a block are actually scheduled on the SM's.

## Thread Scheduling

When a block is assigned to a SMX, it is further divided into groups of 32 threads called a **warp**. Warp scheduling is different depending on the platform, but if we take a look at the Kepler architecture, we see that a single SMX consists of 192 CUDA cores (a CUDA core is also sometimes referred to a streaming processor or **SP** for short).

Each SMX in the Kepler architecture features four warp schedulers allowing four warps to be issued and executed concurrently. Kepler's quad-warp scheduler selects four warps and issues two independent instructions from each warp every cycle[2].

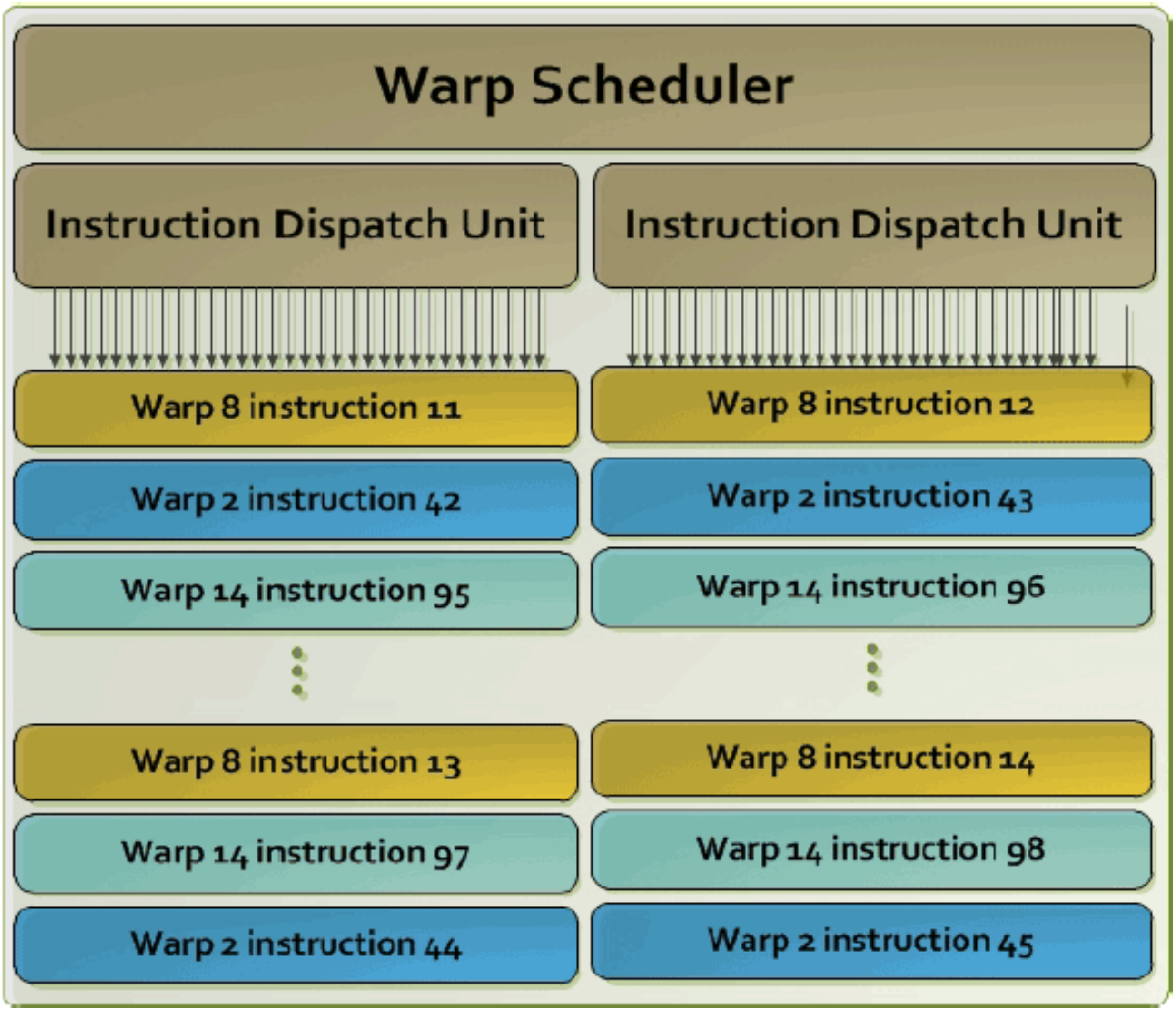


**Figure 20.** Warp Scheduler

You might be wondering why it would be useful to schedule 16 blocks of a maximum of 1024 threads if the SMX only has 192 cuda cores? The answer is that each instruction of a kernel may require more than a few clock cycles to execute (for example, an instruction to read from global memory will require multiple clock cycles). Any instruction that requires multiple clock cycles to execute incurs latency. The latency of long-running instructions can be hidden by executing instructions from other warps while waiting for the result of the previous warp. This technique of filling the latency of expensive operations with work from other threads is often called **latency hiding**.

## Thread Divergence

It is reasonable to imagine that your CUDA program contains flow-control statements like **if-then-else**, **switch**, **while** loops, or **for** loops. Whenever you introduce these flow-control statements in your code, you also introduce the possibility of thread divergence. It is important to be aware of the consequence of thread divergence and also to understand how you can minimize the negative impact of divergence.

Thread divergence occurs when some threads in a warp follow a different execution path than others. Let's take the following code block as an example:

test.cu

```
__global__ void TestDivergence( float* dst, float* src )
{
    unsigned int index = ( blockDim.x * blockIdx.x ) +
threadIdx.x;
    float value = 0.0f;

    if ( threadIdx.x % 2 == 0 )
    {
        // Threads executing PathA are active while threads
        // executing PathB are inactive.
        value = PathA( src );
    }
    else
    {
        // Threads executing PathB are active while threads
        // executing PathA are inactive.
        value = PathB( src );
    }
    // Threads converge here again and execute in parallel.
    dst[index] = value;
}
```

Then our flow control and thread divergence would look something like this:

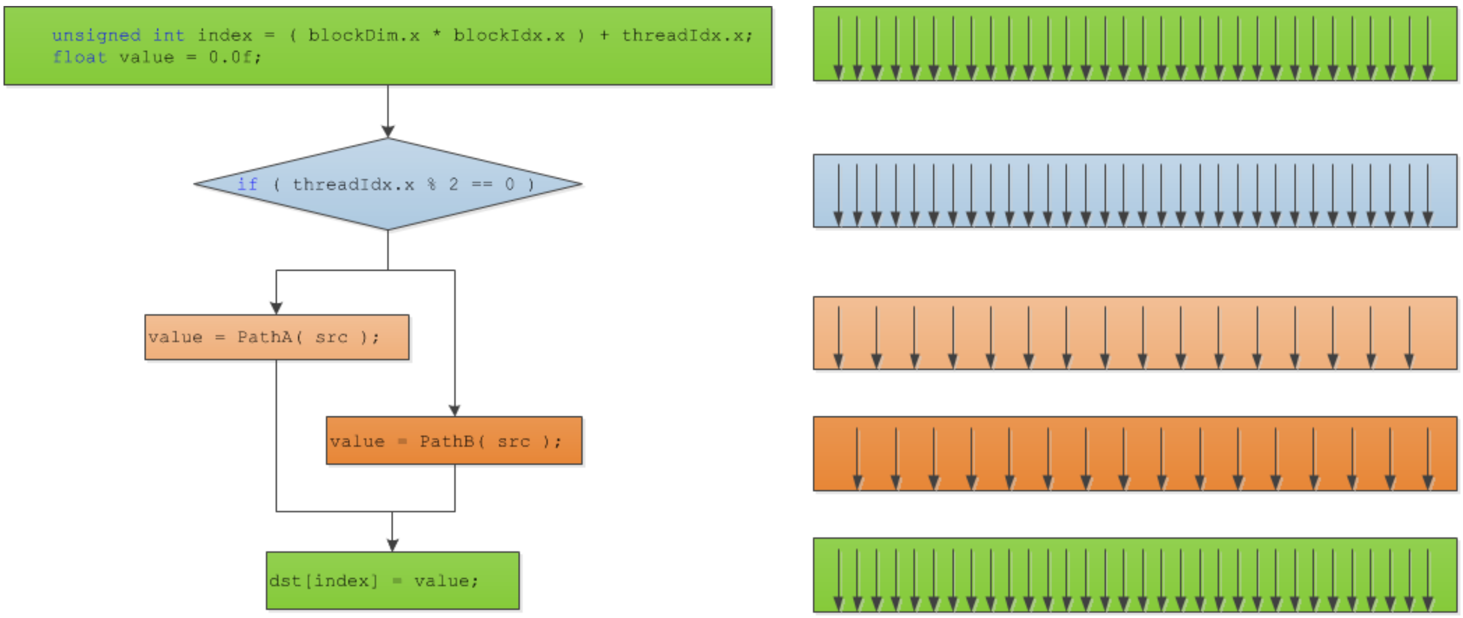

unsigned int index = ( blockDim.x * blockIdx.x ) + threadIdx.x;
float value = 0.0f;
if ( threadIdx.x % 2 == 0 )
value = PathA( src );
value = PathB( src );
dst[index] = value;

**Figure 21.** Thread Divergence

As you can see from this example, the even numbered threads in each block will execute **PathA** while the odd numbered threads in the block will execute **PathB**. This is pretty much the worst-case scenario for simple divergence example.

Both **PathA** and **PathB** cannot be executed concurrently on all threads because their execution paths are different. Only the threads that execute the exact same execution path can run concurrently so the total running time of the warp is the sum of the execution time of both **PathA** and **PathB**.

In this example, the threads in the warp that execute **PathA** are activated if the condition is true and all the other threads are deactivated. Then, in another pass, all the threads that execute **PathB** are activated if the condition is false are activated and the other threads are deactivated. This means that to resolve this condition requires 2-passes to be executed for a single warp.

The overhead of having the warp execute both **PathA** and **PathB** can be eliminated if the programmer takes careful consideration when writing the kernel. If possible, all threads of a block (since warps can't span thread blocks) should execute the same execution path. This way you guarantee that all threads in a warp will execute the same execution path and there will be no thread divergence within a block.

## Memory Model

There are several different types of memory that your CUDA application has access to. For each different memory type there are tradeoffs that must be considered when designing the algorithm for your CUDA kernel.

Global memory has a very large address space, but the latency to access this memory type is very high. Shared memory has a very low access latency but the memory address is small compared to Global memory. In order to make proper decisions regarding where to place data and when, you must understand the differences between these memory types and how these decisions will affect the performance of your kernel.

In the next sections, I will describe the different memory types and show examples of using different memory to improve the performance of your kernel.

## CUDA Memory Types

Every CUDA enabled GPU provides several different types of memory. These different types of memory each have different properties such as access latency, address space, scope, and lifetime.

The different types of memory are **register**, **shared**, **local**, **global**, and **constant** memory.

On devices with compute capability 1.x, there are 2 locations where memory can possibly reside; cache memory and device memory.

The cache memory is considered "on-chip" and accesses to the cache is very fast. Shared memory and cached constant memory are stored in cache memory with devices that support compute capability 1.x.

The device memory is considered "off-chip" and accesses to device memory is about ~100x slower than accessing cached memory. Global memory, local memory and (uncached) constant memory is stored in device memory.

On devices that support compute capability 2.x, there is an additional memory bank that is stored with each streaming multiprocessor. This is considered L1-cache and although the address space is relatively small, it's access latency is very low.

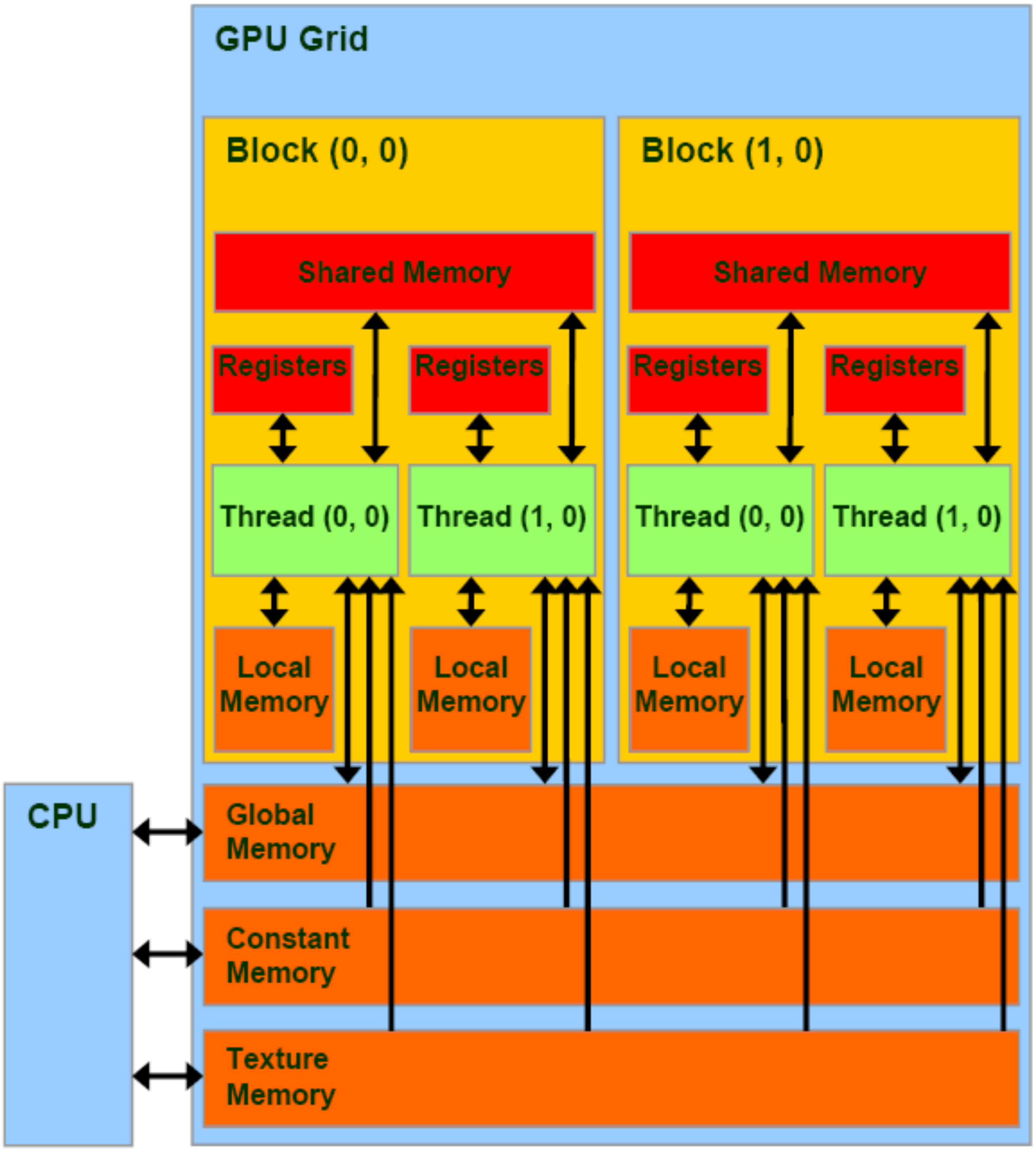


**Figure 22.** CUDA Memory Model

In the following sections I will describe each type and when it is best to use that memory type.

## Register

Scalar variables that are declared in the scope of a kernel function and are not decorated with any attribute are stored in register memory by default.

Register memory access is very fast, but the number of registers that are available per block is limited.

Arrays that are declared in the kernel function are also stored in register memory but only if access to the array elements are performed using constant indexes (meaning the index that is being used to access an element in the array is not a variable and thus the index can be determined at compile-time). It is currently not possible to perform random access to register variables.

Register variables are private to the thread. Threads in the same block will get private versions of each register variable. Register variables only exists as long as the thread exists. Once the thread finishes execution, a register variable cannot be accessed again. Each invocation of the kernel function must initialize the variable each time it is invoked. This might seem obvious because the scope of the variable is within the kernel function, but this is not true for all variables declared in the kernel function as we will see with shared memory.

Variables declared in register memory can be both read and written inside the kernel. Reads and writes to register memory does not need to be synchronized.

## Local

Any variable that can't fit into the register space allowed for the kernel will spill-over into local memory. Local memory has the same access latency as global memory (that is to say, slow). Accesses to local memory is cached only on GPU's with compute capability 2.x or higher[4].

Like registers, local memory is private to the thread. Each thread must initialize the contents of a variable stored in local memory before it should be used. You cannot rely on another thread (even in the same block) to initialize local memory because it is private to the thread.

Variables in local memory have the lifetime of the thread. Once the thread is finished executing, the local variable is no longer accessible.

You cannot decorate a variable declaration with any attribute but the compiler will automatically put variable declarations in local memory under the following conditions:

- Arrays that are accessed with run-time indexes. That is, the compiler can't determine the indices at compile time.
- Large structures or arrays that would consume too much register space.
- Any variable declared that exceeds the number of registers for that kernel (this is called register-spilling).

The only way that you can determine if the compiler has put some function scope variables in local memory is by manual inspection of the PTX assembly code (obtained by compiling with the **-ptx** or **-keep** option). Local variables will be declared using the **.local** mnemonic and loaded using the **ld.local** mnemonic and stored with the **st.local** mnemonic.

Variables in local memory can be both read and written within the kernel and access to local memory does not need to be synchronized.

## Shared

Variables that are decorated with the **__shared__** attribute are stored in shared memory. Accessing shared memory is very fast (~100 times faster than global memory) although each streaming multiprocessor has a limited amount of shared memory address space.

Shared memory must be declared within the scope of the kernel function but has a lifetime of the block (as opposed to register, or local memory which has a lifetime of the thread). When a block is finished execution, the shared memory that was defined in the kernel cannot be accessed.

Shared memory can be both read from and written to within the kernel. Modification of shared memory must be synchronized unless you guarantee that each thread will only access memory that will not be read-from or written-to by other threads in the block. Block synchronization is acheived using the **__syncthreads()** barrier function inside the kernel function.

Since access to shared memory is faster than accessing global memory, it is more efficient to copy global memory to shared memory to be used within the kernel but only if the number of accesses to global memory can be reduced within the block (as we'll see with the matrix multiply example that I will show later).

## Global

Variables that are decorated with the **__device__** attribute and are declared in global scope (outside of the scope of the kernel function) are stored in global memory. The access latency to global memory is very high (~100 times slower than shared memory) but there is much more global memory than shared memory (up to 6GB but the actual size is different across graphics cards even of the same compute capability).

Unlike register, local, and shared memory, global memory can be read from and written to using the C-function **cudaMemcpy**.

Global memory has a lifetime of the application and is accessible to all threads of all kernels. One must take care when reading from and writing to global memory because thread execution cannot be synchronized across different blocks. The only way to ensure access to global memory is synchronized is by invoking separate kernel invocations (splitting the problem into different kernels and synchronizing on the host between kernel invocations).

Global memory is declared on the host process using **cudaMalloc** and freed in the host process using **cudaFree**. Pointers to global memory can be passed to a kernel function as parameters to the kernel (as we will see in the example later).

Reads from global memory is cached only on devices that support compute capability 2.x or higher[4] but any write to global memory will invalidate the cache thus eliminating the benefit of cache. Access to global memory on devices that support compute capability 1.x is not cached.

It is a bit of an art-form to reduce the number of accesses to global memory from within a kernel by using blocks of shared memory because the access latency to shared memory is about 100 times faster than accessing global memory. Later, I will show an example of how we can reduce the global memory access using shared memory.

## Constant

Variables that are decorated with the **__constant__** attribute are declared in constant memory. Like global variables, constant variables must be declared in global scope (outside the scope of any kernel function). Constant variables share the same memory banks as global memory (device memory) but unlike global memory, there is only a limited amount of constant memory that can be declared (64KB on all compute capabilities).

Access latency to constant memory is considerably faster than global memory because constant memory is cached but unlike global memory, constant memory cannot be written to from within the kernel. This allows constant memory caching to work because we are guaranteed that the values in constant memory will not be changed and therefor will not become invalidated during the execution of a kernel.

Constant memory can be written to by the host process using the **cudaMemcpyToSymbol** function and read-from using the **cudaMemcpyFromSymbol** function. As far as I can tell, it is not possible to dynamically allocate storage for constant memory (the size of constant memory buffers must be statically declared and determined at compile-time).

Like global memory, constant memory has a lifetime of the application. It can be accessed by all threads of all kernels and the value will not change across kernel invocations unless explicitly modified by the host process.

## Properties of Memory

The amount of memory that is available to the CUDA application is (in most cases) specific to the compute capability of the device. For each compute capability, the size restrictions of each type of memory (except global memory) id defined in the table below. The application programmer is encouraged to query the device properties in the application using the **cudaGetDeviceProperties** method.

**Table 2.** Memory Compute Capability

| **Technical Specifications** | **1.0** | **1.1** | **1.2** | **1.3** | **2.x** | **3.0** | **3.5** |
|---|---|---|---|---|---|---|---|
| Number of 32-bit registers per thread | 128 | | | | 63 | | 255 |
| Maximum amount of shared memory per SM | 16 KB | | | | 48 KB | | |
| Amount of local memory per thread | 16 KB | | | | 512 KB | | |
| Constant memory size | 64 KB | | | | | | |

The following table summarizes the different memory types and the properties of those types.

**Table 3.** Properties of Memory Types

| **Memory** | **Located** | **Cached** | **Access** | **Scope** | **Lifetime** |
|---|---|---|---|---|---|
| Register | cache | n/a | Host: None<br>Kernel: R/W | thread | thread |
| Local | device | 1.x: No<br>2.x: Yes | Host: None<br>Kernel: R/W | thread | thread |
| Shared | cache | n/a | Host: None<br>Kernel: R/W | block | block |
| Global | device | 1.x: No<br>2.x: Yes | Host: R/W<br>Kernel: R/W | application | application |
| Constant | device | Yes | Host: R/W<br>Kernel: R | application | application |

## Pointers to Memory

You can use pointers to memory in a kernel but you must be aware that the pointer type does not determine where the memory is located.

For example, the following code declares a pointer to constant memory and a pointer to global memory. You should be aware that only the pointer variable is constant – not what it points to.

test.cu

```
__constant__ float* constPtr;
__device__ float* globalPtr;

__global__ void KernelFunction(void)
{
    // Assign the pointer to global memory to a pointer to constant memory.
    // This will not compile because the pointer is constant and you can't change
    // what a const-pointer points to in the kernel.
    constPtr = globalPtr;

    // This will compile because what the const pointer points to is not
    // necessarily const (if it is, you'll probaly get a runtime error).
    *constPtr = *globalPtr;
}
```

Since you can't dynamically allocate constant memory, this example would not be very useful anyways.

Be careful when using pointers like this. It is a best-practice rule to ensure that a declared pointer only points to one type of memory (so a single pointer declaration will only point to global memory and another pointer declaration will only point to shared memory).

## Minimize Global Memory Access

Since access latency is much higher for global memory than it is for shared memory, it should be our objective to minimize accesses to global memory in favor of shared memory. This doesn't mean that every access to data in global memory should first be copied into a variable in shared (or register) memory. Obviously we will not benefit from the low latency shared memory access if our algorithm only needs to make a single access to global memory. But it happens in some cases that multiple threads in the same block will all read from the same location in global memory. If this is the case, then we can speed-up our algorithm by first allowing each thread in a block to copy one part of the global memory into a shared memory buffer and then allowing all of the threads in a block to access all elements in that shared memory buffer.

To demonstrate this, I will show several flavors the classic matrix multiply example. The first example I will show is the standard implementation of the matrix multiply using only global memory access. Then, I will show an optimized version of the algorithm that uses shared memory to reduce the number of accesses to global memory for threads of the same block.

## Matrix Multiply using Global Memory

This version of the matrix multiply algorithm is the easiest to understand however it is also a very naive approach.

MatrixMultiply.cu

```
1  __global__ void MatrixMultiplyKernel_GlobalMem( float* C, const float* A, const float* B, unsigned int matrixDim )
2  {
3      // Compute the row index
4      unsigned int i = ( blockDim.y * blockIdx.y ) + threadIdx.y;
5
6      // Compute the column index
7      unsigned int j = ( blockDim.x * blockIdx.x ) + threadIdx.x;
8
9      unsigned int index = ( i * matrixDim ) + j;
10     float sum = 0.0f;
11     for ( unsigned int k = 0; k < matrixDim; ++k )
12     {
13         sum += A[i * matrixDim + k] * B[k * matrixDim + j];
14     }
15     C[index] = sum;
   }
```

The parameters **A**, **B**, and **C** all point to buffers of global memory.

The fist step is to figure out which row (**i**) and which column (**j**) we are operating on for this kernel.

On line 10, we loop through all of the elements of row **i** of matrix **A** and the column **j** of matrix **B** and compute the summed product of corresponding entries (the dot product of row **i** and column **j**). A visual aid of this algorithm is shown below.

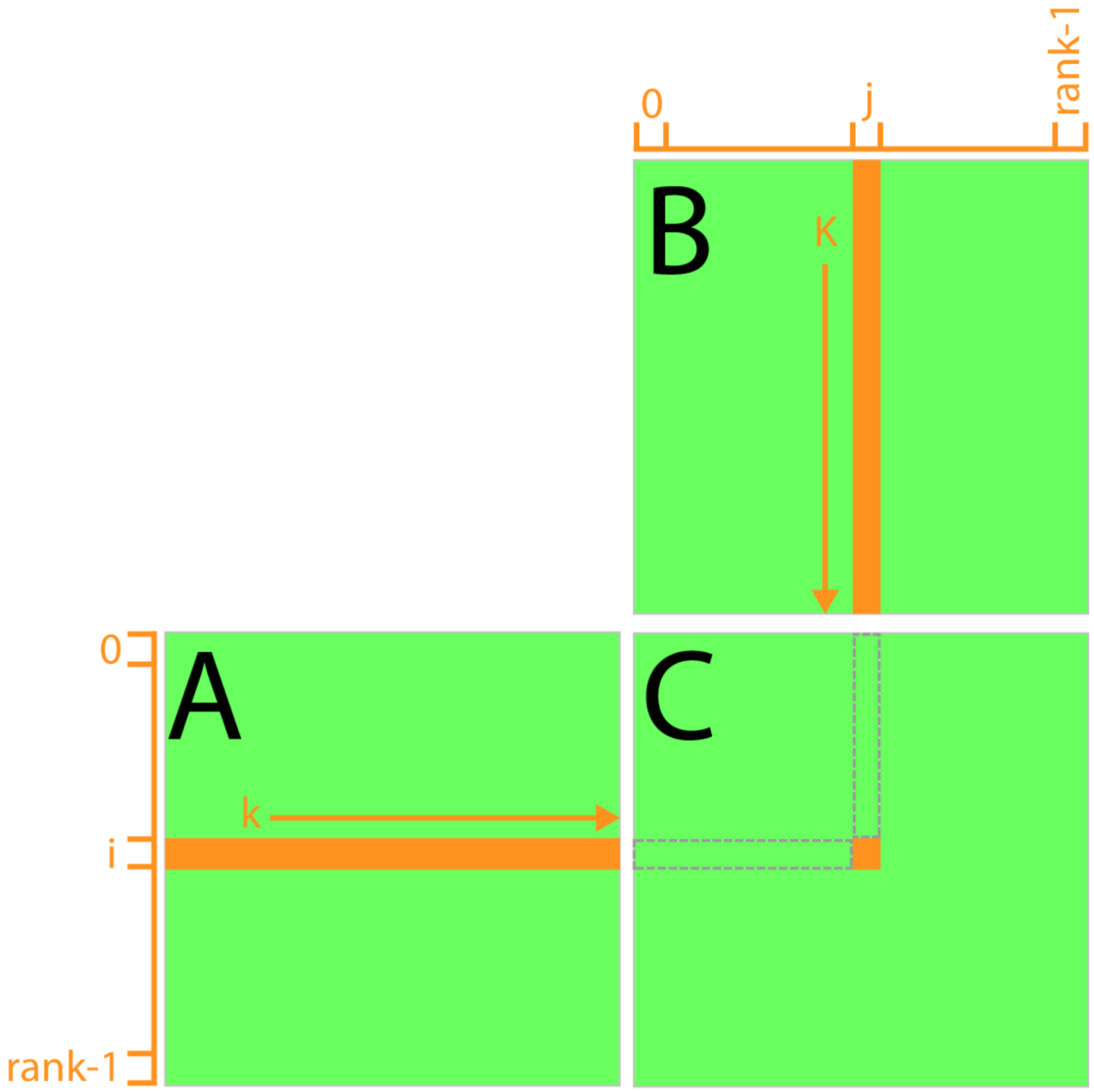


**Figure 23.** Matrix Multiply – Global Memory

If we analyze this algorithm, we may notice that the same row elements of matrix **A** are being accessed for every resulting row element of matrix **C** and all the column elements of matrix **B** are being accessed for every resulting column element of matrix **C**. If we say that the resulting matrix **C** is **N** x **M** elements, then each element of matrix **A** is being accessed **M** times and each element of matrix **B** is being accessed **N** times. That seems pretty wasteful to me.

## Matrix Multiply using Shared Memory

What if we could reduce the number of times the elements of matrix **A** and **B** are accessed to just 1? Well, depending on the size of our matrix, we could just store the contents of matrix **A** and matrix **B** into shared memory buffers then just compute the resulting matrix **C** from those buffers instead. This might work with small matrices (remember that shared memory is local to a single block and with compute capability 1.3, we are limited to matrices of about 20 x 20 because we are limited to 512 threads that can be assigned to a single block).

But what if we had larger matrices to multiply? If we can find a way to split the problem into "phases" then we could simply load each "phase" into shared memory, process that "phase", then load the next "phase" and process that one until we have exhausted the entire domain.

This technique of splitting our problem domain into phases is called "tiling" named because of the way we can visualize the technique as equal sized tiles that represent our problem domain.

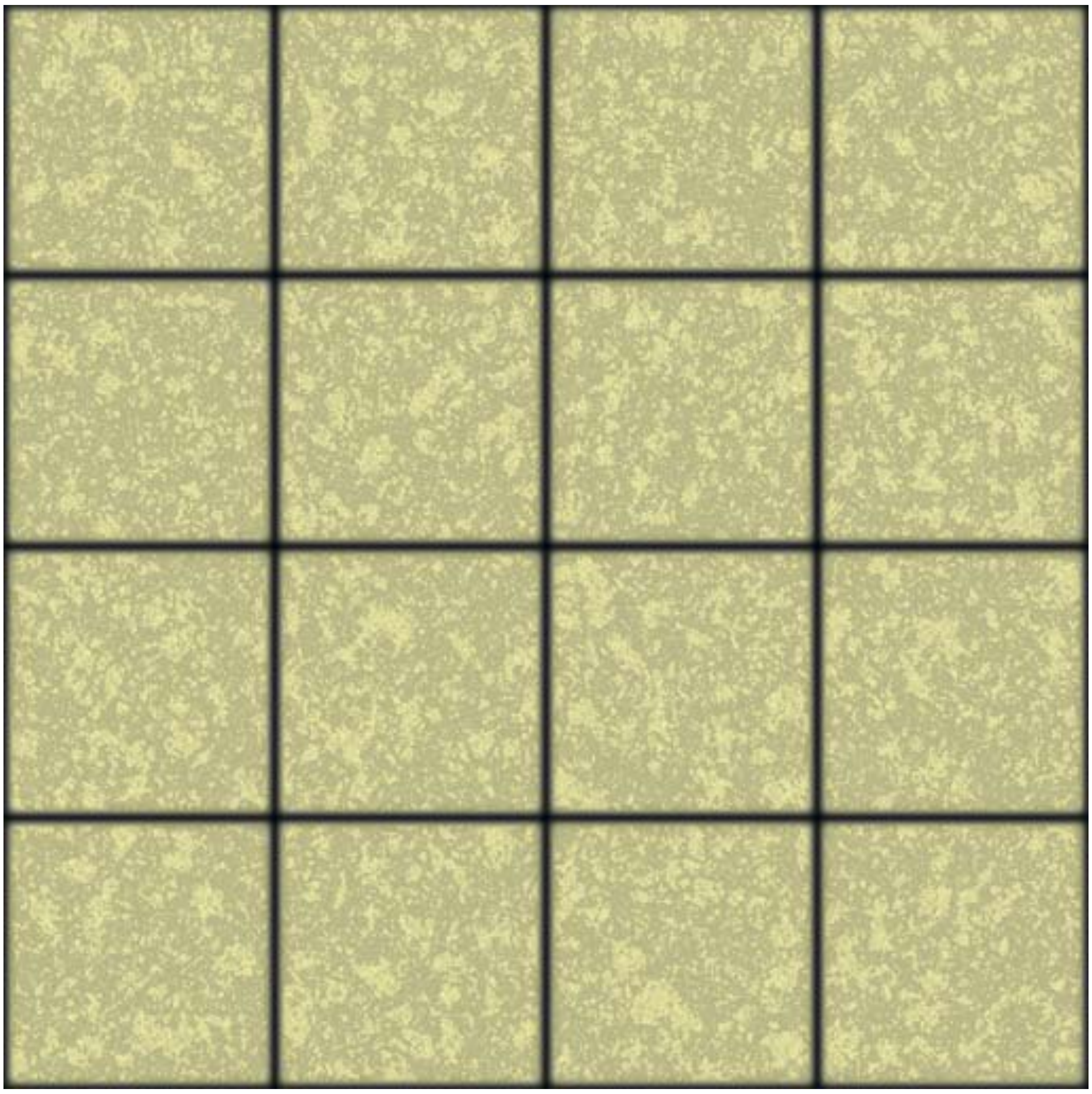


**Figure 24.** Tiles

For this particular problem, the best partitioning of the problem domain is actually the same as partitioning of the grid of threads that are used to compute the result.

If we split our grid into blocks of 16 x 16 threads (which I showed previously in the section about CUDA thread execution to be a good granularity for this problem) then we can create two buffers in shared memory that are the same size as a single thread block in our kernel grid, one that holds a "tile" of matrix **A**, and other to store a "tile" of matrix **B**.

Let's see how this might look:

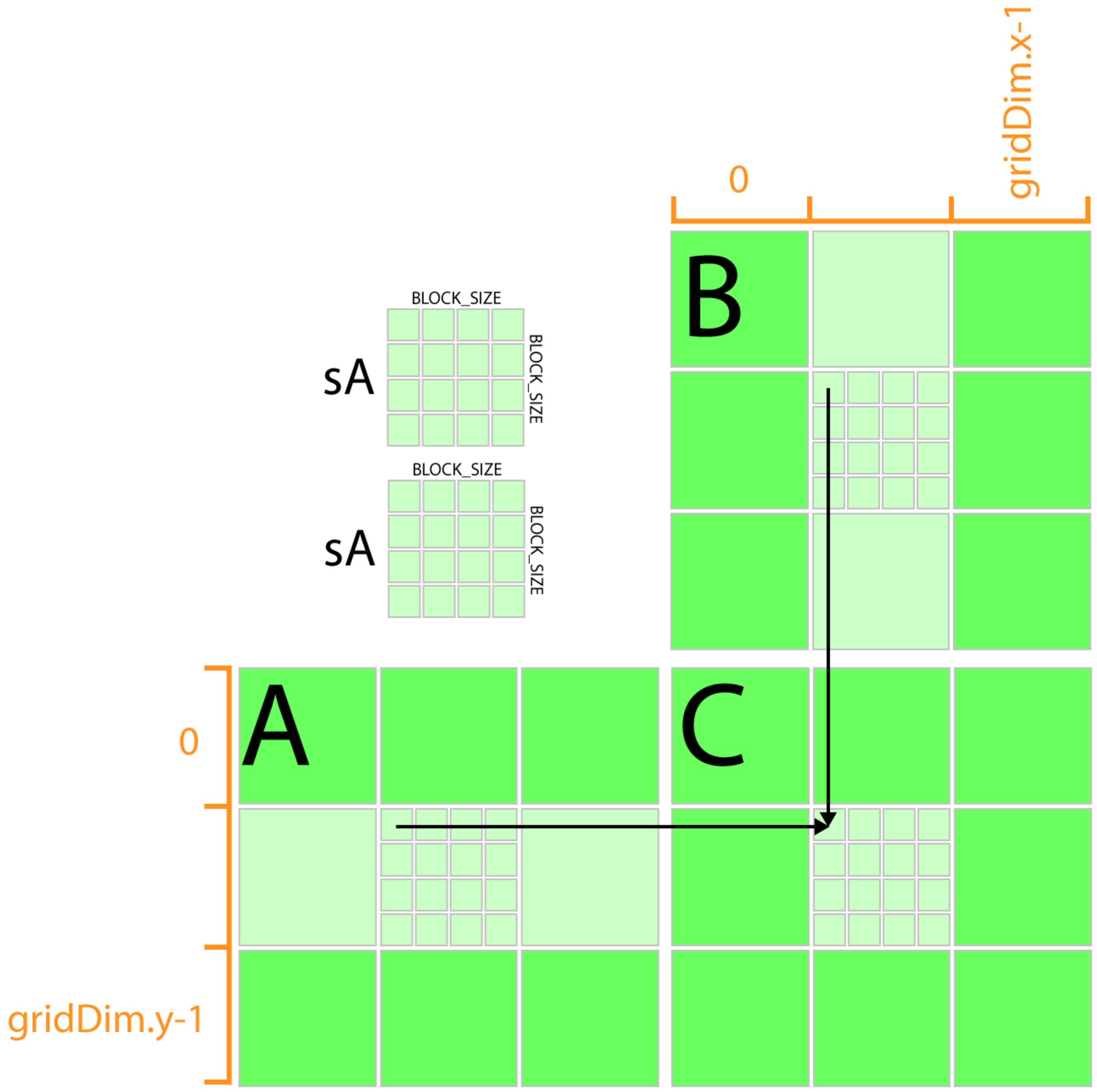


**Figure 25.** Matrix Multiply – Tiles

So the idea is simple, each thread block defines a pair of shared memory buffers that are used to "cache" a "tile" of data from matrix **A** and matrix **B**. Since the "tile" is the same size as the thread block, we can just let each thread in the thread block load a single element from matrix **A** into one of the shared memory buffers and a single element from matrix **B** into the other. Using this technique, we can reduce the number of global memory access to **matrixDim / BLOCK_SIZE** per thread (where **BLOCK_SIZE** is the size of the thread block and shared memory buffer in a single dimension).

But will this work? We only have access to 16 KB (16,384 Bytes) of shared memory per streaming multiprocessor for devices of compute capability 1.x. If our **BLOCK_SIZE** is 16 then we need $16^2$ floating point values (4-bytes each) per shared memory buffer. So the size in bytes of each shared memory buffer is:

$$\begin{aligned} bufferSize &= 16^2 \cdot 4 \\ bufferSize &= 256 \cdot 4 \\ bufferSize &= 1024 \end{aligned}$$

And we need 2 buffers, so we will need 2,048 Bytes of shared memory per block. If you remember from the previous article about the CUDA thread execution model,
thread blocks of size 16 x 16 will allow 4 resident blocks to be scheduled per streaming multiprocessor. So 4 blocks each requiring 2,048 Bytes gives a total requirement of 8,192 KB of shared memory which is 50% of the available shared memory per streaming multiprocessor. So this this tiling strategy will work.

So let's see how we might implement this in the kernel.

MatrixMultiply.cu

```
#define BLOCK_SIZE 16



__global__ void MatrixMultiplyKernel_SharedMem( float* C,
const float* A, const float* B, unsigned int matrixDim )
{
    unsigned int tx = threadIdx.x;
    unsigned int ty = threadIdx.y;
    unsigned int bx = blockIdx.x;
    unsigned int by = blockIdx.y;

    // Allocate share memory to store the matrix data in tiles
    __shared__ float sA[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float sB[BLOCK_SIZE][BLOCK_SIZE];

    // Compute the column index
    unsigned int j = ( blockDim.x * bx ) + tx;
    // Compute the row index
    unsigned int i = ( blockDim.y * by ) + ty;

    unsigned int index = ( i * matrixDim ) + j;
    float sum = 0.0f;

    // Loop through the tiles of the input matrices
    // in separate phases of size BLOCK_SIZE
    for( unsigned int phase = 0; phase < matrixDim/BLOCK_SIZE; ++phase )
    {
        // Allow each thread in the block to populate the shared memory
        sA[ty][tx] = A[i * matrixDim + (phase * BLOCK_SIZE + tx)];
        sB[ty][tx] = B[(phase * BLOCK_SIZE + ty) * matrixDim + j];
```

```
31          __syncthreads();
32
33          for( unsigned int k = 0; k < BLOCK_SIZE; ++k )
34          {
35              sum += sA[ty][k] * sB[k][tx];
36          }
37          __syncthreads();
38      }
39

        C[index] = sum;
    }
```

On line 5-8, we just store some “shorthand” versions of the thread and block indexes into private thread variables (these are stored in registers).

On line 11, and 12 the two shared memory buffers are declared to store enough values that each thread in the thread block can store a single entry in the arrays.

On line 15, the index of the column is computed and stored in another registry variable **j** and on line 16, the row is computed and stored in registry variable **i**.

On line 20, the 1-D index into the result matrix **C** is computed and the sum of the products is stored in the float variable **sum**.

On line 25, we will loop over the “tiles” (called phases here) of matrix **A** and matrix **B**. You should note that this algorithm assumes the size of the matrix is evenly divisible by the size of the thread block.

On lines 28 and 29 is where the magic happens. Since the shared memory is accessible to every thread in the block, we can let every thread in the block copy 1 element from matrix **A** and one element from matrix **B** into the shared memory blocks.

Before we can access the data in the shared memory blocks, we must ensure that all threads in the entire block have had a chance to write their data. To do that we need to synchronize the execution of all the threads in the block by calling the **__syncthreads()** method.

Then the **for** loop on line 32 will loop through the elements of shared memory and sum the products.

Before we leave this loop and start filling the next "tile" into shared memory, we must ensure that all threads are finished with the shared memory buffers. To do that, we must execute **__syncthreads()** again on line 36.

This will repeat until all phases (or tiles) of the matrix have been processed.

Once all phases are complete, then the value stored in **sum** will contain the final result and it is written to the destination matrix **C**.

Running the global memory version of the matrix multiply on my laptop with a 512 x 512 matrix runs in about 45 milliseconds. Running the shared memory version on the same matrix completes in about 15 milliseconds (including copying memory from host to device and copying the result back to host memory). This provides a speed-up of 300%!

## Resources as a Limiting Constraint

It is entirely possible to allocate more shared memory per block than 2,048 bytes, but the block scheduler will reduce the number of blocks scheduled on a streaming multiprocessor until the shared memory requirements are met. If you want to allocate all 16 KB of shared memory in a single block, then only a single block will be resident in the streaming multiprocessor at any given moment which will reduce the occupancy of the streaming multiprocessor to 25% (for a 16 x 16 thread block on compute capability 1.x).

This reduced thread occupancy is not ideal, but it is conceivable to imagine that a single block might have this requirement. In most cases the GPU will still out-perform the CPU if the benefit of using the low-latency memory is fully realized.

This is also true for the number of registers that can be allocated per block. If a single kernel declares 32 32-bit variables that must be stored in registers and the thread block consists of 16 x 16 threads, then the maximum number of blocks that can be active in a streaming multiprocessor on a device with compute capability 1.3 is 2 because the maximum number of 32-bit registers that can be used at any moment in time is 16,384.

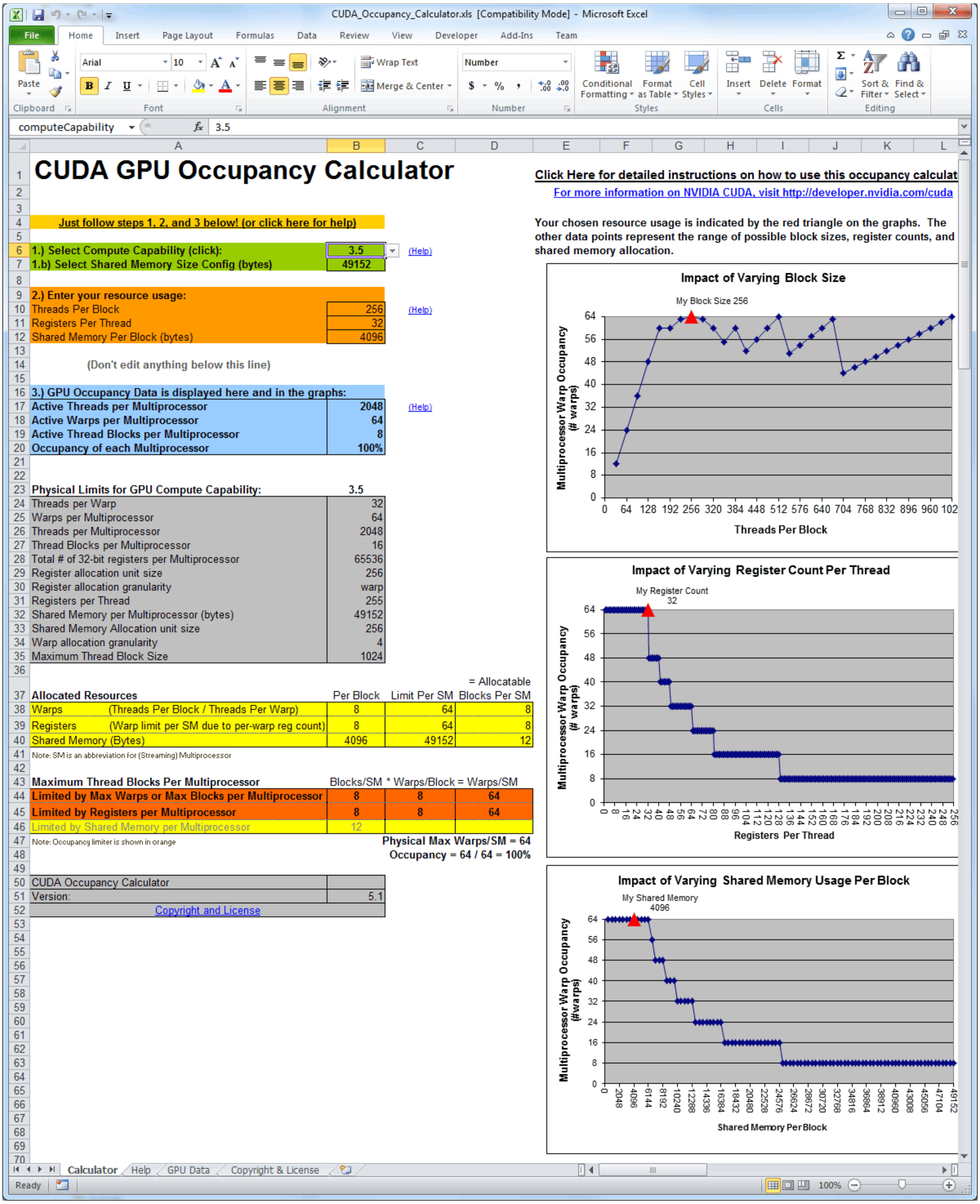
$$\begin{aligned} numRegisters &= 16^2 \cdot 32 \\ numRegisters &= 256 \cdot 32 \\ numRegisters &= 8192 \end{aligned}$$

So the number of 32-bit registers/block is 8,192. So the streaming multiprocessor can accommodate a maximum of 8,192 / 16,384 = 2 blocks.

## CUDA GPU Occupancy Calculator

Since version 4.1, the CUDA Toolkit comes with a tool called the **CUDA GPU Occupancy Calculator**. This tool is a Microsoft Excel file that can be used to compute the maximum thread occupancy of the streaming multiprocessor given a set of limiting constraints (threads per block, registers per thread, and shared memory (bytes) per block). This tool is provided in the following folder:

**C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\vX.X\tools**



computeCapability | 3.5

# 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) |
| 1.b) Select Shared Memory Size Config (bytes) | 49152 | |
| **2.) Enter your resource usage:** | | |
| Threads Per Block | 256 | (Help) |
| Registers Per Thread | 32 | |
| Shared Memory Per Block (bytes) | 4096 | |

(Don't edit anything below this line)

| | | |
|---|---|---|
| **3.) GPU Occupancy Data is displayed here and in the graphs:** | | |
| Active Threads per Multiprocessor | 2048 | (Help) |
| 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 |
| Warps per Multiprocessor | 64 |
| Threads per Multiprocessor | 2048 |
| Thread Blocks per Multiprocessor | 16 |
| Total # of 32-bit registers per Multiprocessor | 65536 |
| Register allocation unit size | 256 |
| Register allocation granularity | warp |
| Registers per Thread | 255 |
| Shared Memory per Multiprocessor (bytes) | 49152 |
| Shared Memory Allocation unit size | 256 |
| Warp allocation granularity | 4 |
| Maximum Thread Block Size | 1024 |

| Allocated Resources | | Per Block | Limit Per SM | = Allocatable Blocks Per SM |
|---|---|---|---|---|
| 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 | 12 |

Note: SM is an abbreviation for (Streaming) Multiprocessor

| Maximum Thread Blocks Per Multiprocessor | Blocks/SM | * Warps/Block | = Warps/SM |
|---|---|---|---|
| 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 | 12 | | |

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

| | |
|---|---|
| CUDA Occupancy Calculator | |
| Version: | 5.1 |
| Copyright and License | |

Click Here for detailed instructions on how to use this occupancy calculat

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

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.

Impact of Varying Block Size
My Block Size 256
Multiprocessor Warp Occupancy (# warps)
Threads Per Block
Impact of Varying Register Count Per Thread
My Register Count 32
Registers Per Thread
Impact of Varying Shared Memory Usage Per Block
My Shared Memory 4096
Multiprocessor Warp Occupancy (#warps)
Shared Memory Per Block

Figure 26. CUDA Occupancy Calculator

The **CUDA Occupancy Calculator** allows you to compute the best thread granularity for your thread blocks given a specific compute capability and resource constraints.

You can refer to the second worksheet titled "Help" to learn how to use the **CUDA GPU Occupancy Calculator**.

## Exercises

**Q1.** Would the **MatrixAddDevice** kernel function shown in this article benefit from the use of **shared memory**? Explain your answer.

**A1.** No, it would not benefit from the use of shared memory because each matrix element is only accessed once. You would still need to access each matrix component to store it in shared memory only to require an access from shared memory to access it again. In this case, store the data in shared memory will only increase the time to execute the kernel because more load/store operations will need to be performed.

**Q2.** In almost all of the examples shown here, I decided to use a 16×16 thread granularity for the thread blocks. Can you explain why this is a good choice for thread granularity on devices of compute capability (you can assume that register use and shared memory allocation are within the limits in each case):

1. 1.3?
2. 2.0?
3. 3.0?

**A2.** To answer this question, let's take a look at each individual compute capability.

**a.** For Compute Capability 1.3 threads are split into groups of **32** threads called **warps**. The maximum number of warps/SM is **32**. If we create a 16×16 thread block, then we have a total of 256 threads/block. Each block will be split into 8 warps to be scheduled on the SM. Since we know that the maximum number of warps/SM for devices with compute capability 1.3 is 32, then **4 thread blocks** will be scheduled on each SM. Each SM can support up to 8 resident blocks per SM and 4 is still within our limit. Also with a maximum resident thread limit of **1024 threads** and we exactly meet this requirement (4×256) so we also achieve **100% thread occupancy** on the SM! So yes, a 16×16 thread block is a good choice for devices with compute capability 1.3.

**b.** For devices with compute capability 2.0 threads are also split into groups of **32** threads called **warps**. In this case, the maximum number of warps/SM is **48**. Again, we have 256 threads per block which are split into 8 warps to be scheduled on the SM then **6 thread blocks** will be scheduled per SM (48/8). 6 blocks is within the 8 block limit, so we haven't exceeded the block limit. And with a maximum resident thread limit of **1536 threads**, we exactly meet this requirement (6×256) so we also achieve a **100% thread occupancy** on the SM! So yes, a 16×16 thread block is also a good choice for devices with compute capability 2.0.

**c.** For devices with compute capability 3.0 the threads are also split into groups of **32** threads called **warps**. So again, each block will be split into 8 warps. The maximum number of warps that can be active in a SM is **64**. This allows for **8 thread blocks** to be scheduled per SM. This is within the limit of 16 blocks/SM and again matches exactly the maximum number of threads of **2048 threads** (8×256) that can be scheduled for each SM so we also achieve **100%** thread occupancy. So yes, a 16×16 thread block is also a good choice for devices with compute capability 3.0 (and consequently this is also true for devices of compute capability 3.5).

**Q3.** Assuming we have a block of 256 threads each, what is the maximum amount of shared memory we can use per block and still maintain 100% thread occupancy for devices of compute capability (assume the register count is not a limiting resource):

1. 1.3?
2. 2.0?
3. 3.0?

**a.** In the previous exercise we already established that with a thread blocks of 256 threads, we will have 4 resident blocks per SM. Since devices of compute capability 1.3 have a maximum **16 KB (16,384 bytes)** of shared memory then each block can use a maximum of **4,096 bytes** (16,384/4) of shared memory while still maintaining 100% thread occupancy.

**b.** In the previous exercise we saw that we could schedule 6 blocks of 256 threads. Devices of compute capability 2.0 have a maximum of **48 KB (49,152 bytes)** of shared memory per SM. This means that we can allocate a maximum of **8,192 bytes** (49,152/6) of shared memory while still maintaining 100% thread occupancy.

**c.** In the previous exercise we saw that we could schedule 8 blocks of 256 threads to get 100% thread occupancy. Devices with compute capability 3.0 also have a maximum of **48 KB (49,152 KB)** of shared memory per SM. In this case, we can only allocate **6,144 bytes** (49,152/8) of shared memory while still maintaining 100% thread occupancy.

**Q4.** In the case (c) above, what would happen if we created thread blocks of 1024 threads? Would we still have 100% thread occupancy? How much shared memory could we allocate per thread block and maintain 100% thread occupancy? Explain your answer.

**Q5.** Answer question (3) and (4) again but this time compute the number of registers you have available **per thread** while still maintaining 100% thread occupancy. In this case, you can assume that shared memory is not a limiting resource.

**Hint:** To answer Q5 correctly, you must also take the register allocation granularity and unit size into consideration. For compute capability 1.3, the register allocation granularity is at the **block** level and the register allocation unit size is **512**. For compute capability 2.x register allocation granularity is at the **warp** level and the register allocation unit size is **64**. For compute capability 3.x, the register allocation granularity is at the **warp** level and the register allocation unit size is **256**.

## References

1. NVIDIA Corporation (2012, October). *CUDA C Programming Guide*. (PG-02829-001_v5.0). USA. Available from: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Accessed: October 2012.

2. NVIDIA Corporation (2012, October). *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*. (V1.0). USA. Available from: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf. Accessed: October 2012.

3. NVIDIA Corporation (2012, October). *NVIDIA CUDA Getting Started Guide For Microsoft Windows*. (DU-05349-001_v5.0). USA. Available from: http://developer.download.nvidia.com/compute/cuda/5_0/rel/docs/CUDA_Getting_Started_Guide_For_Microsoft_Windows.pdf. Accessed: October 2012.

4. NVIDIA Corporation (2012, October). *CUDA C Best Practices Guide*. (DG-05603-001_v5.0). USA. Available from: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf. Accessed: October 2012.

5. Kirk, David B. and Hwu, Wen-mei W. (2010). *Programming Massively Parallel Processors*. 1st. ed. Burlington, MA 01803, USA: Morgan Kaufmann Publishers.

This entry was posted in CUDA, General Purpose GPU Programming and tagged 3.0, 3.5, C++, compute capability, CUDA, introduction, NVIDIA, Programming, tutorial, Visual Studio by Jeremiah van Oosten. Bookmark the permalink.