GPU Programming

CSInParallel Project

July 24, 2012

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is GPU?

Graphics Processing Unit (GPU), is an electronic circuit that, through rapid memory manipulation and massive parallel data processing, accelerates the building of images intended for output to a display. Right now, GPUs are used in almost all customer end personal computers, game consoles, professional workstations and even the cell phone you are holding.

Before GPU was introduced, CPU did all the graphic processing tasks. In the early 1990s, computer manufactures began to include GPU into computer system with the aim of accelerating common graphics routines. After two decades of development, GPUs eventually outpaced CPUs as they actually had more transistors, ran faster and capable of doing parallel computation more efficiently. GPU these days become so complex that they are basically computers in themselves, with their own memory, buses and processors. Therefore, sometimes GPU is like an extra brain (supplemental processors) to the computer system. As GPU harnessed more and more horsepower, GPU manufactures, such as NVIDIA and ATI/AMD, found a way to use GPUs for more general purposes, not just for graphics or videos. This gave birth to CUDA structure and CUDA C Programming Language, NVIDIA's response on facilitating the development of General Purpose Graphics Processing Unit (GPGPU).

1.2 What is the difference between GPU and CPU?

The major difference between GPU and CPU is that GPU has highly parallel structure which makes it more effective than CPU if used on data that can be partitioned and processed in parallel. To be more specific, GPU is highly optimized to perform advanced calculations such as floating-point arithmetic, matrix arithmetic and so on.

The reason behind the difference of computation capability between CPU and GPU is that GPU is specialized for compute-intensive and highly parallel computation, which is exactly what you need to render graphics. The design of GPU is more of data processing than data caching and flow control. If a problem can be processed in parallel, it usually means that: first, same problem is executed for each element, which requires less sophisticated flow control; second, dataset is massive and problem has high arithmetic intensity, which reduces the need for low latency memory.

The graph above shows the different between CPU and GPU in their structure. **Cache** is designed for data caching; **Control** is designed for flow control; **ALU** (Arithmetic Logic Unit) is designed for data processing.

In the NVISION 08 Conference organized by NVIDIA corporation, employers from NVIDIA used a rather interesting yet straight forward example illustrating the difference between CPU and GPU. You can watch the video by clicking the link below and hope it can give you a better idea what the difference between GPU and CPU is. Video

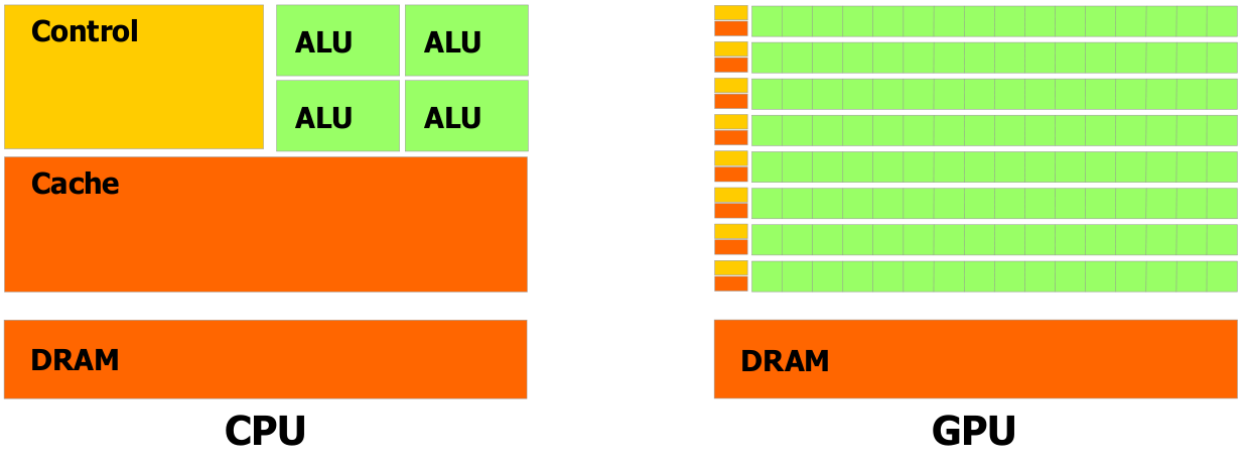

Figure 1.1: This figure is from the NVIDIA CUDA Programming Guide

1.3 What is the advantage of using GPU for computation?

Nowadays, most of the scientific researches require massive data processing. What scientist usually do right now is have all the data being processed on supercomputing clusters. Although most universities have constructed their own parallel computing clusters, researchers still need to compete for time to use the shared resources that not only cost millions to build and maintain, but also consume hundreds of kilowatts of power.

Different from traditional supercomputers that is built with many CPU cores, supercomputers with GPU structure can achieve same level of performance with less cost and lower power consumption. Personal Supercomputer (PSC) based on NVIDIA's Tesla companion processors, was first introduced in 2008. The first generation four-GPU Tesla personal supercomputer have 4 Teraflops of parallel supercomputing performance, more than enough for most small researches. All it takes is 4 Tesla C1060 GPUs with 960 cores and two Intel Xeon processors. Moreover, Personal supercomputer is also very energy efficient as it can even run off a 110 volt wall circuit. Although supercomputer with GPUs cannot match the performance of the top ranking supercomputers that cost millions even billions, it is more than enough for researchers to perform daily research related computations in subjects like bioscience, life science, physics and geology.

1.4 What are the important parts inside a GPU?

Although modern GPU are basically computers themselves, they still serve as a part of computer system. A modern GPU is connected with the host through high speed I/O bus slot, usually a PCI-Express slot. Modern GPUs are extremely energy consuming. Some of the GPUs alone consumes hundreds watts of power, sometimes higher than all other parts of the computer system combined. Part of the reason that GPUs require such power supply is that they have much complex structure and can perform much sophisticated task than other parts of computer system. Owe to its high capability, GPU needs its own memory, control chipset as well as many processors.

GPUs these days usually equipped with several gigabytes of on-board memory for user configuration. GPUs designed for daily use like gaming and video rendering, such as NVIDIA's Geforce series GPUs and ATI/AMD's Radeon series GPUs, have on-board memory capacity ranging from several hundreds megabytes to several gigabytes. Professional GPUs designed for high-definition image processing and General Purpose Computation, such as the Tesla Companion Processor we are using, usually have memory up to 5 or 6 gigabytes. Data is transferred between the GPU on-board memory and host memory through a method called DMA (Direct Memory Access). One thing needs mentioning is that CUDA C programming language supports direct access of the host memory from GPU end under certain restrictions. As GPU is designed for compute intensive operations, device memory usually supports high data bandwidth with not

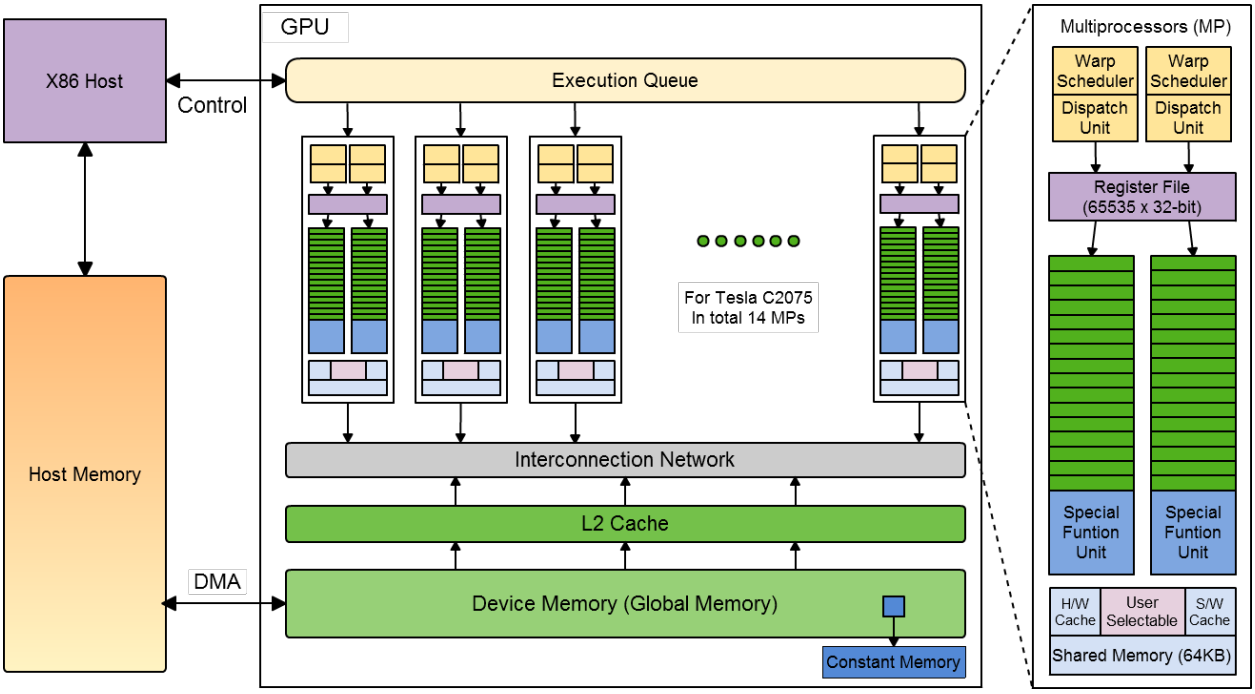

Figure 1.2: This figure is inspired by the figure found in Understanding the CUDA Data Parallel Threading Model: A Primer written by Michael Wolfe from The Portland Group

deeply cached memory hierarchy.

GPUs from NVIDIA has many processors, what they call streaming Processors (SP). Each streaming processor is capable of executing a sequential thread. For a GPU with *Fermi* architecture, like the one we are using, every 32 streaming processors is organized in a Streaming Multiprocessor (SM). A GPU can have one or more multiprocessors on board. For example, the Tesla C2075 GPU card we are using, has 14 multiprocessors built in. Except for 32 streaming processors, each multiprocessor also equipped with 2 warp scheduler, 2 special function units (4 in some GPUs), a set of 32-bit registers and 64KB of configurable shared memory. Warp scheduler is responsible for threads control; SFU handles transcendentals and double-precision operations. For a GPU with *Kepler* architecture, every 192 streaming processors is organized in a multiprocessor. There are also more warp schedulers and SFUs built in.

Shared memory, or L1 cache, is a small data cache that can be configured through software. Shared memory is also shared among all the streaming processors within one multiprocessor. Compared with on-board memory, shared memory is low-latency (usually register speed) and has high bandwidth. Each multiprocessor has 64KB of shared memory that can be configured by user using special commands in host code. Share memory is distributed to software-managed data cache and hardware data cache. User can choose to assign either 48KB to software-managed data cache (SW) and 16KB to hardware data cache (HW) or the other way around.

1.5 How does CUDA connect with hardware?

When the host code invokes a kernel grid through CUDA program, blocks in the grid are distributed to different multiprocessors based on available execution capacity of each multiprocessor. Each multiprocessor is capable of process one or more blocks throughout the kernel execution. However, each block can only be processed by one multiprocessor.

Fermi architecture supports up to 48 active warps on each multiprocessor. The advantage of having many active warps in process at the same time is the significantly reduction of memory latency. Traditionally, memory latency is reduced by adding more cache memory hierarchy into the system. However, by using high degree of multithreading,

GPUs can also effectively reduce memory latency. What happens is that when one warp stalls on memory operation, multiprocessor can select another warp and begin to process that one.

When a block is processed by a multiprocessor, threads in this block is further divided into groups of 32 threads, what NVIDIA calls a warp. Although 32 streaming processors in a block and 32 threads in a warp seems to be a good match for each multiprocessor to process each warp in one clock cycle, the reality is somehow different. As mentioned previously, each multiprocessor has two warp schedulers, which enables it to process two warps simultaneously. After the partition, each warp gets scheduled by a warp scheduler for execution. Each warp scheduler pumps 16 threads (half warp) into a group of 16 streaming processors for execution. Therefore, it would take two clock cycles to process each warp and one multiprocessor can process two warps in two clock cycles. For double-precision operations, each multiprocessor would combine two groups of streaming processors so that they act as a multiprocessor with 16 double-precision streaming processors.

1.6 Is CUDA the only GPU programming language available?

When we are learning CUDA C programming language, it is important for you to know that C programming language is not the only language that can be bind with CUDA structure. NVIDIA also made available other programming languages such as Fortran, Java and Python as binding languages with CUDA.

Furthermore, NVIDIA is not the only company manufacturing GPU cards, which means CUDA is not the only GPU programming MPI available. When NVIDIA are developing CUDA, AMD/ATI responded with ATI Stream, their GPGPU technology for AMD/ATI Radeon series GPUs. ATI Stream technology uses OpenCL as its binding language.

CHAPTER

TWO

CUDA INTRO

Before we proceed to our first example, please follow the following instructions to set up your working environment.

1. Download the follow file. `download common.tar.gz`
2. Extract all the files.
3. Create a empty folder **common** and then put extracted files into it.
4. Put the common folder **outside** of your source code folder.

Note: In side the common folder are source code that contains helper functions. These include some handle error functions and APIs required for the later graphic programs, such as ray-tracing.

2.1 Acknowledgement

The examples used in this chapter are based on examples in CUDA BY EXAMPLE: An Introduction to General-Purpose GPU Programming, written by Jason Sanders and Edward Kandrot, and published by Addison Wesley.

This copy of code is a derivative based on the original code and designed for educational purposes only. It contains source code provided by NVIDIA Corporation.

2.2 An Example of Vector Addition

We will start our CUDA journey by learning a very simple example, the vector addition example. It basically takes two vectors that have same dimensions, adds them together and then returns the new vector back.

Vector Addition source file: `VA-GPU-11.cu`

2.2.1 The Device Code

As you may notice in your background reading about CUDA programming, CUDA programs execute in two separated places. One is called host, another is called device. In our example, the add() function executes on the device (our GPU) and the rest of the C program executes on our CPU.

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = 0;
    // loop over all the element in the vector
    while (tid < N){
        c[tid] = a[tid] + b[tid];
        tid += 1; // we are using one thread in one block
    }
}
```

As shown in the code block above, we need to add a **__global__** qualifier to the function name of the original C code in order to let function add() execute on a device.

You might notice that this code is much like standard C code except for the **__global__** qualifier. We are seeing this because this version of vector addition's device code is utilizing only one core of the GPU. We can see this from the line

```
        tid += 1; // we are using one thread in one block
```

where we only add 1 to the *tid*. In the later examples, where we will be using more cores of the GPU, you will see difference of CUDA C programming language and Standard C programming language.

2.2.2 The Host Code

Before you proceed

Different from device code, the host code is more complicated and require more explanation. We advice you to download the source file provided at the beginning of this page and have it open in a separate window. We divided the host code into several parts for the purpose of easier explanation. However, looking at the host code as a whole might be helpful, especially for CUDA programming, where host codes are usually highly organized and structured.

```
int main( void ) {

    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

    // allocate memory on the CPU
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

    // fill arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
```

As shown in the code block above, similar to standard C programming, we first need to declare pointers. Notice that we declared two sets of pointers, one set is used to store data on host memory, another is used to store data on the device memory.

The Event API

Before we go any further, we need to first learn ways of measuring performance in CUDA runtime. How do we measure performance? Well, in the simplest yet most direct way, how fast can the program run. To be more specific, we will try to time the program.

The tool we use to measure the time GPU spends on a task is CUDA Event API.

```
// define events in the field
cudaEvent_t     start, stop;
// create two events we need
HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );
// instruct the runtime to record the event start.
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
```

The first step of using event is declaring the event. In this example. we declared two events, one called start, which will record the start event and another called stop, which will record the stop event. After declaring the events, we can use the command cudaEventRecord() to record a event. You can think of recording a event as initializing it. You may also notice that we pass this command a second argument (0 in this case). In our example, this argument is actually useless. However, if you are really interested in this, you can read more about CUDA stream.

You can see we that there is another function HANDLE_ERROR() around each of the commands. For the moment, this function does not do anything than returning the error if CUDA commands run into any.

Why would we use Event API?

If you are C programming language veteran you may ask the question: why don't we use the the timing functions in standard C, such as *clock()* or *timeval* structure, to perform this task? Well, this is a really good question. The fundamental motivation of using Event API instead of timing functions in standard C lies on the difference between CPU and GPU computation. To be more specific, GPU is a companion computation device, which means every time CPU has to call GPU to do computations. However, when GPU is doing computation, CPU does not wait for it to finish its task, instead CPU will continue to execute next line of code while GPU is still working on previous call. This *asynchronous* feature of GPU computation structure leads to possible inaccuracy when measuring time using standard C timing functions. Therefore, Event API become needed.

The cudaMalloc() Function

```
// allocate memory on the GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
```

Just like standard C programming language, you need to allocate memory for variables before you start to use them. The command cudaMalloc(), similar to *malloc()* command in standard C, tells the CUDA runtime to allocate the memory on the device (Memory of GPU), instead of on the host (Memory of CPU). The first argument is a pointer points to where you want to hold the address of the newly allocated memory.

For some reasons, you are not allowed to modify memory allocated on the device (GPU) from host directly in CUDA C programming language. Instead, you need to use two other method to access device memory. You can do it by either using device pointers in the device code, or you can use the cudaMemcpy() method.

The way to use pointers in the device code is exactly the same as we did in the host code. In other words, pointer in CUDA C is exactly the same as Standard C. However, there is one thing you need to pay attention to, host pointers can only access memory (usually CPU memory) from host code, you cannot access device memory directly. On the other hand, device pointers can only access memory (usually GPU memory) from device code as well.

The cudaMemcpy() Function

```
// copy arrays 'a' and 'b' to the GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                          cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                          cudaMemcpyHostToDevice ) );
```

As mentioned in last section, We can also use cudaMemcpy() from host code to access memory on a device. **This command is the typical way of transferring data between host and device.** Again this call is similar to standard C call *memcpy()*, but requires more parameters. The first argument identifies the destination pointer; the second identifies the source pointer. The last parameter to the call is cudaMemcpyHostToDevice, telling the runtime that the source pointer is a host pointer and the destination pointer is a device pointer.

The Kernel Invocation

```
// kernel invocation code
add<<<1,1>>>( dev_a, dev_b, dev_c );
```

The following line is the call for device code from host code. You may notice that this call is similar to a normal function call but has additional code in it. We will talk about what they represent in later examples. At this point all you need to know is that they are they are telling the GPU to use only one thread to execute the program.

More cudaMemcpy() Function

```
// copy array 'c' back from the GPU to the CPU
HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                          cudaMemcpyDeviceToHost ) );
```

In previous section we have seen how CUDA runtime transfer data from Host to Device, this time we will see how to transfer data back to host. Notice that this time device memory is source and host memory is destination. Therefore, we are using argument cudaMemcpyDeviceToHost.

Timing using Event API

```
// get stop time, and display the timing results
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
float   elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Time to generate:  %3.1f ms\n", elapsedTime );
```

We have seen how to declare and record a Event API in CUDA C, but have not elaborate how to use such tool to measure performance. The basic idea is that we first declare event start and event stop. Then at the beginning of the program we record event start and at the end of the program we record event stop. The last step is to calculate the elapsed time between two events.

As shown in the code block above, we again use command cudaEventRecord() to instruct the runtime to record the event stop. Then we proceed to the last step, which is get elapsed time using command cudaEventElapsedTime().

However, there is still a problem with timing GPU code in this way. The CUDA C programming language, though is derived from standard C, has many characteristics that is different from standard C. We have mentioned in previous section that CUDA C is asynchronous. This is a example to jog your memory. Suppose we are running a program

to do matrices multiplication, and host calls the GPU to do the computation. As GPU begins executing our code, the CPU proceeds to the next line of code instead of waiting GPU to finish its work. If we want the stop event to record the correct time, we need to make sure that our event is recorded after the GPU finishes everything prior to the call to cudaEventRecord(). To address this problem, CUDA C calls the function cudaEventSynchronize() to synchronize the stop event.

The cudaEventSynchronize() function is essentially instructing the runtime to create a barrier to block CPU from executing further instructions until the GPU has reached the stop event.

Another caveat worth mentioning is that CUDA events are implemented directly on the GPU. Therefore they cannot be used for timing device code mixed with host code. In other words, you will get unreliable results if you attempt to use CUDA events to time more than kernel executions and memory copies involving the device.

you should include and only include kernel execution and memory copies involving the device in between start event and stop event. Anything more included could lead to unreliable results.

The cudaFree() Function

```
// free memory allocated on the GPU
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );

// destroy events to free memory
HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
```

When you are reading the section about cudaMalloc(), It may occur to you that we might a call different from the call *free()* to free memory on the device. You are absolutely right. To free memory allocated on the device, we need to use command cudaFree() instead of free().

To finish up the code, we need to free memory allocate on the CPU as well.

```
// free memory allocated on the CPU
free( a );
free( b );
free( c );
```

You can add the following code verify whether the GPU has done the task correctly or not. This time we are using CPU to verify GPU's work. We can do this in this problem due to small data size and simple computation.

2.3 Vector Addition with Blocks

We have learned some basic concepts in CUDA C in our last example. Starting from this example, we will begin to learn how to write CUDA language that will explore the potential of our GPU card.

Vector Addition with Blocks source file: `VA-GPU-N1.cu`

2.3.1 Block

Recall that in the previous example, we use the code

```
    add<<<1,1>>>( dev_a, dev_b, dev_c );
```

to call for device kernels and we left those two numbers in the triple angle brackets unexplained. Well, the first number tells the kernel how many parallel blocks we would like to use to execute the instruction. For example, if we launch the kernel <<<16,1>>>, we are essentially creating 16 copies of the kernel and running them in parallel. We call each of these parallel invocations a block.

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid. Why do we need two-dimensional or even three-dimensional grid? why can't we just stick with one-dimensional? Well, it turned out that for problems with two or more dimensional domains, such as matrices multiplication or image processing (don't forget the reason GPU been exist is to process image faster), it is often convenient and more efficient to use two or more dimensional indexing. Right now, nVidia GPUs that support CUDA structure can assign up to 65536 blocks in each dimension of the grid, that is in total $65536 \times 65536 \times 65536$ blocks in a grid.

Note: Some of the books may refer grid in CUDA has top two-dimensions. On the other hand, some books (including the official CUDA Programming Guide provided by NVIDIA) may suggest that grid can be three-dimensional. It turns out that older GPU units are not powerful enough to run grid in three-dimensions. Therefore older books might refer CUDA has only two-dimensional grid. However, as GPUs get more and more powerful, NVIDIA enable newer GPUs to utilize three-dimensional grid.

To see whether your device support three-dimensional grid or not, please run the following source code and see the Compute Capability entry in the output. If it is 2.x or 3.x, then your device supports three-dimensional grid. If it is 1.x or less, you can only use two-dimensional grid. `download enum_gpu.cu`

2.3.2 The Device Code

```
__global__ void add( int *a, int *b, int *c ) {

    // keep track of the index
    int tid = blockIdx.x;

    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += numBlock; // shift by the total number of blocks
    }
}
```

This is the complete device code.

We have mentioned that there are one, two and three-dimensional grids. To index different blocks in a grid, we use the built-in variables CUDA runtime defines for us: blockIdx. blockIdx is a three-component vector, so that threads can be identified using one-dimensional, two-dimensional or three-dimensional index. To access different component in this vector, we use blockIdx.x, blockIdx.y and blockIdx.z.

```
    // keep track of the index
    int tid = blockIdx.x;
```

Since we have multiple blocks doing the same task, we need to keep track of these blocks so that the kernel can pass right data to them and bring right data back. Since we have only 1 thread in each block, we can simply use blockIdx to track index.

```
        tid += numBlock; // shift by the total number of blocks
```

Although we have multiple blocks (1 thread per block) working simultaneously after one block finish one computation, this does not necessary mean block will only perform one time of computation. Normally, we could have problem size that is larger than the number of blocks we have. Therefore, we need each block to perform more than one time of

computation. We do this by adding a stride to the tid after the while loop finish one round. In this example, we want tid to shift to the next data point by the total number of blocks.

2.3.3 The Host Code

```
add<<<numBlock,numThread>>>( dev_a, dev_b, dev_c );
```

Except kernel invocation part of the host code, everything else is the same. However, as we are calling **numBlock** and **numThread** in the code, we need to define them at the very beginning of the source code file.

```
#define numThread 1 // in this example we keep one thread in one block
#define numBlock 128 // in this example we use multiple blocks
```

2.4 Vector Addition with Blocks and Threads

Vector Addition with Blocks and Threads source file: `VA-GPU-NN.cu`

2.4.1 Threads

In the last example, we learned how to launch multiple blocks in CUDA C programs. This time, we will see how to split parallel blocks. CUDA runtime allow us to split block into threads. Recall that in the previous example, we use the code

```
add<<<numBlock,numThread>>>( dev_a, dev_b, dev_c );
```

to call for device kernels where numBlock is 128 and numThread remain as 1, the second number represents how many threads we want in each block.

Here comes the question, why do we need two sets of parallel organization system? Why do we need not only blocks in grid, but also threads in blocks? Is there any advantages in one over the other? Well, there are advantages that we will cover in later examples, so for now, please bear with us.

Just like blocks is organized in up to three-dimensional grid, threads can also be organized in one, two or three-dimensional blocks. Just like there is a limit on number of blocks in a grid, there is also a limit on number of threads in a block. Right now, for most of the high-end nVidia GPUs, this limit is 1024. Be really careful here, 1024 is the total number of threads in a block, **not** the limit **per dimension** like in the grid. Most of the nVidia GPUs that is two or three year old, the limit might be 512. You can query the maxThreadsPerBlock field of the device properties structure to find out which number you have.

2.4.2 The Device Code

```
__global__ void add( int *a, int *b, int *c ) {

    // keep track of the index
    int tid = threadIdx.x + blockIdx.x * numBlock;

    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += numThread * numBlock;// shift by the total number of thread in a grid
    }
}
```

This is the complete device code.

Just like we use CUDA built-in variables to index blocks in a grid, we use variable threadIdx to index threads in a block. threadIdx is also a three-component vector and you can access each of its element using threadIdx.x, threadIdx,y and threadIdx.z.

```
// keep track of the index
int tid = threadIdx.x + blockIdx.x * numBlock;
```

The thread handles the data at its thread id. Recall that earlier we are using tid = blockIdx.x only. Now, as we are using multiple threads per block, we have to keep track of not only blockId, but also the threadId as well.

```
tid += numThread * numBlock;// shift by the total number of thread in a grid
```

Since we have multiple threads in multiple blocks working simultaneously, after one thread in one block finish one computation, we want it to shift to the next data point by the total number of threads in the system. in this example, total number of threads is number of blocks times threads per block.

2.4.3 The Host Code

```
add<<<numBlock,numThread>>>( dev_a, dev_b, dev_c );
```

Except kernel invocation part of the host code, everything else is the same. However, as we are calling **numBlock** and **numThread** in the code, we need to define them at the very beginning of the source code file.

```
#define numThread 128 // in this example we use multiple threads
#define numBlock 128 // in this example keep on using multiple blocks
```

CHAPTER

THREE

THREAD ADVANCE

3.1 Acknowledgement

The examples used in this chapter are based on examples in CUDA BY EXAMPLE: An Introduction to General-Purpose GPU Programming, written by Jason Sanders and Edward Kandrot, and published by Addison Wesley.

3.2 Vector Dot Product

In this example we will see how to perform a dot product using GPU computation. We know that the result of vector addition is a vector, but the result of vector dot product is a number. However, we can divide the vector dot product process into two steps. We first use CUDA to the multiplication process. After this step, the device will return a vector with all its elements as multiplication results to the host code. Then the CPU can do all the adding up process.

Vector Dot Product source file: `Dot-GM.cu`

3.2.1 The Device Code

```
__global__ void dot( float *a, float *b, float *c ) {

    // keep track of the index
    int tid = threadIdx.x + blockIdx.x * numThread;

    while (tid < N) {
         c[tid] = a[tid] * b[tid];
         tid += numThread * numBlock; // shift by the total number of thread in a grid
    }
}
```

The device code is pretty straight forward. Each thread multiplies a pair of corresponding elements in two vectors. After each thread done their job for the first time, if there are still elements left unprocessed, they runtime will instruct the threads to do another round of computation until all the elements are processed.

3.2.2 The Host Code

```
int main( void ) {

    float   *a, *b, sum, *c;
    float   *dev_a, *dev_b, *dev_c;

    // allocate memory on the cpu side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    c = (float*)malloc( N*sizeof(float) );

    // fill in the host memory with data
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }

    // start the timer
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N*sizeof(float) ) );

    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float), cudaMemcpyHostToDevice ) );

    dot<<<numBlock,numThread>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    HANDLE_ERROR( cudaMemcpy( c, dev_c, N*sizeof(float), cudaMemcpyDeviceToHost ) );

    // get stop time, and display the timing results
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    float   elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                        start, stop ) );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );

    // finish up on the CPU side
    sum = 0;
    for (int i=0; i<N; i++) {
        sum += c[i];
    }

    #define sum_squares(x)  (x*(x+1)*(2*x+1)/6)
    printf( "Does GPU value %.6g = %.6g?\n", sum,
             2 * sum_squares( (float)(N - 1) ) );

    // free memory on the gpu side
    HANDLE_ERROR( cudaFree( dev_a ) );
```

```
    HANDLE_ERROR( cudaFree( dev_b ) );
    HANDLE_ERROR( cudaFree( dev_c ) );

    // free memory on the cpu side
    free( a );
    free( b );
    free( c );
}
```

The host code the much like the vector addition example. We first allocate the memory on host memory and device memory. Then we initialize the matrices and fill them with data. After that we copy the data from host to device and execute the kernel code. Finally we transfer the data back from device memory to host memory. Do not forget to use Event API to measure the performance.

However, we need to point out two differences.

```
    float   *a, *b, sum, *c;
    float   *dev_a, *dev_b, *dev_c;
```

First is that we need to declare one more pointer for the host code. In the vector addition example, two sets of array pointers are enough. However, in this example, we are returning a number. Therefore, a pointer which pointing to that number is essential.

```
    // finish up on the CPU side
    sum = 0;
    for (int i=0; i<N; i++) {
        sum += c[i];
    }
```

Another difference is that we need to add up all the elements in the returned vector. This can simply be done by adding a for loop in the host code. Notice that we put this loop outside of the Event API so that it will not interfere our timing result.

You can verify the result by adding the following code into the host code.

```
    #define sum_squares(x)  (x*(x+1)*(2*x+1)/6)
    printf( "Does GPU value %.6g = %.6g?\n", sum,
             2 * sum_squares( (float)(N - 1) ) );
```

For people who are familiar with discrete math, the code above should be simple to apprehend. This function will give the result through a more *clever* way.

3.3 Vector Dot Product with Reduction

In the previous example, you might have the question: why do we need to return the whole array back to the CPU? Is is possible for use to first reduce them a little bit and then return it to the CPU?

Well, we can do reduction in CUDA. In this chapter, we will see how to use reduction in CUDA. But before we proceed, we first need to know something about shared memory in CUDA. In each of the blocks we create, CUDA runtime will assign a region memory to this block. This type of memory is called shared memory. When you are declaring your variables, you can add the CUDA C keyword **__shared__** to make this variable reside in shared memory. Why do we need shared memory?

When we are learning Block and Thread, we had the question why would we need two hierarchy to organize threads question in mind. Well, part of the reason is that we can benefit from shared memory by organize threads in blocks. When we declare a variable and make it reside in shared memory, CUDA runtime creates a copy of this variable in

each block you launched in host code. Every threads in one block shares this memory, which means they can see and modify their shared memory. However, they cannot see or modify shared memory assigned in other blocks. What does this mean to us? Well, if you can have one region of memory that is private to threads in one block, you can explore ways to facilitate communication and collaboration between threads within this block.

Another reason we like to use shared memory is that it is faster than global memory we used to use. As the latency of shared memory tends to be far lower than global memory, it is the ideal choice to serve as cache in each block.

In this example, we will see how we use shared memory to serve as a cache-per-block and how we can perform reduction on it.

Vector Dot Product with Reduction source file: `Dot-SM.cu`

3.3.1 The Device Code

```
__global__ void dot( float *a, float *b, float *c ) {

    // declare cache in the shared memory
    __shared__ float cache[numThread];

    // keep track of thread index
    int tid = threadIdx.x + blockIdx.x * numThread;
    // connect thread index and cache index
    int cacheIndex = threadIdx.x;

    float   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += numThread * numBlock;// increase by the total number of thread in a grid
    }

    // set the cache values
    cache[cacheIndex] = temp;

    // synchronize threads in this block
    __syncthreads();

    // for reductions, numThread must be a power of 2 because of the following code
    int i = numThread/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    // write back to the global memory
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}

    // declare cache in the shared memory
    __shared__ float cache[numThread];
```

In the two lines of code above, we declared a cache in shared memory for this block. We will then use this cache to store each thread's running sum. You can also see that we set the size of the cache same as numThread so that each thread in the block can have its own place to store its running sum. Notice we only create one copy of cache instead

of creating numBlock copies. The compiler will automatically create a copy of cache in each block's shared memory, meaning we only have to declare one copy.

```
float   temp = 0;
while (tid < N) {
    temp += a[tid] * b[tid];
    tid += numThread * numBlock;// increase by the total number of thread in a grid
```

The actual vector dot product computation is the similar to what we seen in the global memory version. However, there is little difference. You can see from the code above, instead of using

```
c[tid] = a[tid] * b[tid];
```

we use

```
temp += a[tid] * b[tid];
```

The reason causes this difference is that we are finding a running sum in this example. In the previous example, as we are returning a vector exactly the same size as the input, we have enough space to store each multiplication results. Even we have less threads than vector dimension so that each thread will compute more than one value, they can store each value in different places. However, in this example, we have exactly the same amount of place for storage as number of threads in each block. If any thread compute more than once, they still have only one place to store values. This bring us to why we need to use running sum of each thread.

You may notice we add a line of code you have never seen before.

```
// synchronize threads in this block
__syncthreads();
```

We have seen code similar to this before. When we are learning how to use CUDA Event API to measure performance, we used command **cudaEventSynchronize()** to synchronize the stop event. The purpose of **__syncthreads()** is somehow similar to the command **cudaEventSynchronize()**. When we are using shared memory to facilitate communication and collaboration between threads, we need to create a way to synchronize them as well. For example, if one thread is writing a number to the shared memory and another thread need to use that number for further computation, we want the first thread to finish its work before the second thread executes its command. What the command __syncthread() will do, is essentially create a barrier for all the threads and block them from executing further command. After all threads have finished executing commands before __syncthread(), then they can all proceed to the next command.

After we make sure all the elements in cache is filled, we can proceed to the reduction process.

```
// for reductions, numThread must be a power of 2 because of the following code
int i = numThread/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
```

Suppose we have a cache with 256 entries. What the code above would do is that it first take 128 thread in the block and each thread will add two of the values in cache[a] and cache[a+128] and store the value back to cache[a]. Then in the second iteration it will take 64 thread in the block and each thread will add values in cache[a] and cache[a+64] and store the value back to cache[a]. After log2(numThread) times of operation, we would have the sum of all 256 values stored in the first element of the cache. Be really careful that we need to synchronize threads every time after we perform one reduction.

Finally, we choose the thread with index 0 to write the result back to the global memory.

```
    // write back to the global memory
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```

3.3.2 The Host Code

In general, the host code of the shared memory version is very similar to that of the global memory version. However, there are several difference we need to point out.

```
    partial_c = (float*)malloc( numBlock*sizeof(float) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                              numBlock*sizeof(float) ) );
```

In the above two lines of code, in the previous example, we declared two sets of pointers with each pointer pointing to array having the same size. This time, however, we need the output array to be smaller than the vector size. Since every block's will write only one number back to the global memory, the size of output array should be numBlock.

Another point worth mentioning is that we define the numBlock in the following way instead of assign a number to it.

```
#define imin(a,b) (a<b?a:b)
const int numBlock = imin( 32, (N+numThread-1) / numThread );
```

When we are choosing the number of blocks to launch in this problem, we faces to requirements. First is that we should not create too many blocks. In the final step where all the results returned by all the blocks are summed up, we are using CPU to compute. This means if we create too many blocks, we would leave CPU too much workload. Another requirements is that we cannot assign too less blocks either. As we can only fit 256 threads in each block, if we assign not enough blocks, we would end up having each thread doing many times of computation. Facing this two requirements, we came up with the solution above. We use the smaller number between 32 and (N+numThread-1) / numThread. The function (N+numThread-1) / numThread gives the smallest multiple of numThread that is equal or larger than the vector size. Calculating this number will ensure we have just enough blocks so that each element in a small vector has its own thread. If we are facing a small vector, we can us the later to assign not too many blocks. If we are facing a gigantic vector, 32 blocks is somehow enough to keep the GPU busy.

Be aware the number 32 was given by a CUDA programming book that is several years old. We decide to use it because we think its safe and yet sufficient for our problem size. If you are dealing with much larger problem size and have much more powerful GPU cards in hand, feel free to stretch this number to thousands even hundreds of thousand.

CHAPTER

FOUR

CUDA IN TWO-DIMENSION

4.1 Acknowledgement

The examples used in this chapter are based on examples in Programming Massively Parallel Processors: A Hands-on Approach, written by David B. Kirk and Wen-mei W. Hwu, and published by Morgan Kaufmann publishers.

This copy of code is a derivative based on the original code and designed for educational purposes only. It contains source code provided by the book above.

4.2 An Example of Matrix Multiplication

In this chapter, we will learn more about GPU computing on multi-dimensional problems and really experience the advantage of GPU computing over CPU computing. Before we proceed to the next section, this section will introduce a Matrix Multiplication program that is written in standard C and use only CPU for the computation. We hope the result we obtain in this chapter can serve as a baseline for following sections and provide a clearer view on how fast GPU computing can be.

CPU Matrix Multiplication Program source file: `MM-CPU.c`

4.2.1 Performance

We conducted 5 tests and the results are below.

- 1. 41045.35 ms
- 2. 40277.44 ms
- 3. 40369.30 ms
- 4. 40385.85 ms
- 5. 40446.14 ms
- Average: 40504.82 ms

4.3 Global Memory Version

Starting from this example, we will look at the how to solve problem in two-dimensional domain using two-dimensional grid and block. As we know, threads can be organized into multi-dimensional block and blocks can also be organized into multi-dimensional grid. This feature in CUDA architecture enable us to create two-dimensional or even three-dimensional thread hierarchy so that solving two or three-dimensional problems becomes easier and more efficient.

In this example, we will do the Square Matrix Multiplication. Two input matrices of size *Width x Width* are M and N. The output matrix is P with the same size. If you have learned linear algebra before, you will know that the output of two square matrices multiplied together is a square matrix of the same size. For example, to calculate entry (A,B) in the output matrix, we need to use row A in one input matrix and column B in another input matrix. We first take the left most element in row A and multiply it by top element in column B. Later, we take the second left element in row A and multiply it by second top element in column B. We do this for all the elements in row A and column B, and then we get the sum of products. The result will be the value at entry (A,B) in the output matrix. As you can see, this kind of operation is highly paralleled, make it perfect for us to use CUDA. We do this by assigning each entry in output matrix a thread of its own. This thread will fetch the data and do all the calculations. It will later on write back the result to the out put matrix.

Matrix Multiplication with Global Memory source file: `MM-GPU-GM.cu`

4.3.1 The Device Code

```
__global__ void Kernel(float *Md, float *Nd, float *Pd, int Width) {

  // Calculate the column index of the Pd element, denote by x
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  // Calculate the row index of the Pd element, denote by y
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  float Pvalue = 0;
  // each thread computes one element of the output matrix Pd.
  for (int k = 0; k < Width; ++k) {
    Pvalue += Md[y*Width + k] * Nd[k*Width + x];
  }

  // write back to the global memory
  Pd[y*Width + x] = Pvalue;
}
```

This is the complete device code.

```
  // Calculate the column index of the Pd element, denote by x
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  // Calculate the row index of the Pd element, denote by y
  int y = threadIdx.y + blockIdx.y * blockDim.y;
```

This 4 lines of code will assign index to the thread so that they can match up with entries in output matrix. As you may notice, we introduced a new CUDA built-in variable **blockDim** into this code. **blockDim** has the variable type of dim3, which is an 3-component integer vector type that is used to specify dimensions. This variable contains the dimensions of the block, and we can access its component by calling blockDim.x, blockDim.y, blockdIM.z.

Each thread in one specific block is identified by threadIdx.x and threadIdx.y. Each block is one specific grid is identified by blockIdx.x and blockIdx.y. Therefore, if we have threadIdx.x, threadIdx.y, blockIdx.x and blockIdx.y, we can locate one specific thread.

4.3.2 The Host Code

```
    dim3 dimBlock(32, 32);
    dim3 dimGrid(Width/32, Width/32);
    Kernel<<<dimGrid, dimBlock>>>( Md, Nd, Pd, Width);
```

This 3 lines of code above is declaring and initializing dim3 variables which give the grid dimensions and block dimensions. In each of the initializations, we only passed two parameters as components. The CUDA runtime will initialize any component left unspecified to 1. So technically, we are initializing dimBlock as (32, 32, 1) and dimGrid as (Width/32, Width/32, 1).

The rest of the host code is similar to examples we have seen before. Here is the complete version of the host code.

```
main(void){

    void MatrixMultiplication(float *, float *, float *, int);

    const int Width = 1024;

    int size = Width * Width * sizeof(float);
    float *M, *N, *P;

    // allocate memory on the CPU
    M = (float*)malloc(size);
    N = (float*)malloc(size);
    P = (float*)malloc(size);

    // initialize the matrices
    for (int y=0; y<Width; y++) {
        for (int x=0; x<Width; x++){
           M[y*Width + x] = x + y*Width;
           N[y*Width + x] = x + y*Width;
        }
    }

    MatrixMultiplication(M, N, P, Width);

    // free the memory allocated on the CPU
    free( M );
    free( N );
    free( P );

    return 0;
}
void MatrixMultiplication(float *M, float *N, float *P, int Width) {

    int size = Width * Width * sizeof(float);
    float *Md, *Nd, *Pd;

    // capture start time
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    // allocate memory on the GPU
    HANDLE_ERROR( cudaMalloc((void**)&Md, size) );
    HANDLE_ERROR( cudaMalloc((void**)&Nd, size) );
    HANDLE_ERROR( cudaMalloc((void**)&Pd, size) );
```

```
    // transfer M and N to device memory
    HANDLE_ERROR( cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice) );
    HANDLE_ERROR( cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice) );

    // kernel invocation code
    dim3 dimBlock(32, 32);
    dim3 dimGrid(Width/32, Width/32);
    Kernel<<<dimGrid, dimBlock>>>( Md, Nd, Pd, Width);

    // transfer P from device
    HANDLE_ERROR( cudaMemcpy(P,Pd,size,cudaMemcpyDeviceToHost) );

    // get stop time, and display the timing results
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    float   elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                        start, stop ) );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );

    // free the memory allocated on the GPU
    HANDLE_ERROR( cudaFree(Md) );
    HANDLE_ERROR( cudaFree(Nd) );
    HANDLE_ERROR( cudaFree(Pd) );

    // destroy events to free memory
    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );
}
```

4.3.3 Performance

In the very top of the source file you can define the size of the matrix. Just change the Width definition to some number you like. While testing the performance, we used 1024 as Width same as the number used in the CPU baseline program. We conducted 5 tests and the results are below.

- 1. 52.5 ms
- 2. 52.4 ms
- 3. 52.4 ms
- 4. 52.4 ms
- 5. 52.6 ms
- average: 52.46 ms

Compared the CPU program, our GPU program is **772** times faster.

4.4 CUDA Memory Types

The reason CUDA architecture has many memory types is to increase the memory accessing speed so that data transfer speed can match data processing speed. The following example will show you why matching these two speeds is so important to GPU computation.

One of the most important standards of a processor's computation ability is its flops computation. We assume that in order to perform one floating point operation, the runtime need to transfer one single-precision floating-point from global memory datum to the computational kernel. With this in mind, we can proceed to our example.

The nVidia Tesla C2075 companion processor supports 144 gigabytes per second (GB/s) of global memory access bandwidth. With 4 bytes in each single precision floating-point datum, we can load no more than 36 (144/4) giga single precision data per second. Since the computational kernel cannot compute more floating-point than the amount global memory has loaded, it will execute no more than 36 gigaflops per second. The actual kernel computational capability of our tesla card is 1 teraflops (1000 gigaflops) per second, but due to limited memory accessing speed, we can only achieve less than 4 percent of the actual speed. In other words, the highest achievable floating-point calculation throughout is limited by the rate at which the input data can be transfered from global memory to computational kernel.

To address this problem, CUDA architecture designed several types of memory that could potentially speed up the data loading process. We will see how use them in later examples. For now, we first need to know specifications of different memory types.

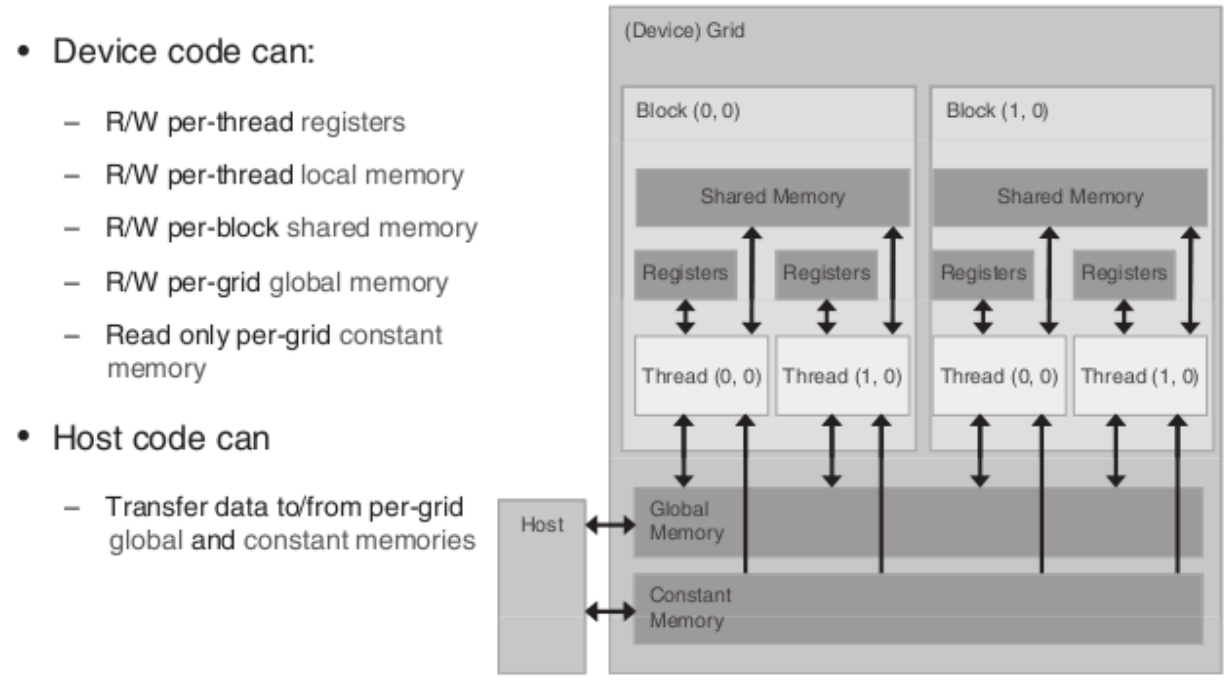

Figure 4.1: This figure is from the website http://www.elsevierdirect.com/v2/companion.jsp?ISBN=9780123814722, originally found in the book Programming Massively Parallel Processors: A Hands-on Approach.

There are in total 4 types of memory designed for GPU cards with CUDA architecture. Global memory, located in the gird, has large storage capacity but limited speed, and can be read and write from all the blocks within CUDA system. Shared memory, located in each block, has small storage capacity (16KB per block) but fast accessing speed, can be read and write by all the threads within the located block. Constant memory, also located in the grid, has very small storage capacity (64KB per GPU) but very fast accessing speed, and can read (can't write) from any threads. There is also local memory located in each thread.

Table 4.1: CUDA Memory Types

Memory	Scope of Access	Lifetime	R/W ability	Speed	Declaration
Register	Thread	Kernel	R/W	Fast	Automatic Variables
Local	Thread	Kernel	R/W	Fast	Automatic Arrays
Shared	Block	Kernel	R/W	Fast	__shared__
Global	Grid	Host	R/W	Slow	__device__
Constant	Grid	Host	Read only	Fast	__constant__

4.5 Shared Memory Version

Matrix Multiplication with Shared Memory source file: `MM-GPU-SM.cu`

Why we need a shared memory version? We already seen hundreds of times speed improvement.

Well, to answer that question, we need to first look at the relationship between global memory and our program. That is, in order to finish the matrix multiplication, how many times each element in matrix is accessed in the global memory?

First, we have in total *Width x Width* many of threads and each thread computes one element of the output matrix. Then, let's take a closer look at each thread. For example, thread with the threadIdx of (x,y) will computes the element in the x column and y row of the output matrix. In order to do this, thread (x,y) have to access elements in row x of matrix M and elements in column y of matrix N. How about thread (x,y+1)? This time kernel will have to access row x in matrix M again and a different column y+1 in matrix N. What about thread (x,y+2) or (x+1,y)? It is not hard for you to find out that we accessed each row in matrix M the *Width* times and each column in matrix N the *Width* times as well. If we can reduce the access time to once for every row in matrix M and once for every column in matrix N, we can not only save bandwidth, but also increase performance significantly.

Notice that although we say we want the kernel to access each row in matrix M and each column in matrix N once from global memory, we are not saying that the kernel access data once throughout the program. As we can see from previous sections, global memory has large capacity but low access speed. What we want is to transfer data from global memory to another type of memory which has fast access speed.

Note: The kernel still need to access every row and every column *Width* times in that memory location. However, as we are accessing them in a faster memory location, the time takes for those data to load will be significantly reduced. So technically we did not reduce the number of times each row or column was accessed, we simply made the speed of accessing them faster.

Upon this point, it may occur to you that shared memory is the ideal candidate for such task for it can access data faster than global memory. However, shared memory also has the drawback of small storage capacity. In the case of matrix multiplication, we can't just store the whole matrix into the shared memory. Remember that shared memory only has 48kB storage space per block, which is not large enough for some gigantic matrices. We solve this problem by managing shared memory in a dynamically way.

In the previous example, we assigned *Width x Width* many of threads for the computation where each thread will read one row of input matrix M and one column of input matrix N and computes the corresponding element in output matrix P. Although we use multiple blocks in a grid and multiple threads in a block, we don't see how threads are cooperating in the previous example. If we are allowed to assign infinite number of threads in one block, we can use just one block for the previous example. In this example, however, we will instruct all threads within one block to cooperate.

In order to make the problem easier, we use two 4x4 matrices for illustration. We set the size of block as 2x2, which in total has 4 threads. Therefore, the output matrix will have 4 blocks. As shown in the **graph** above, each element of

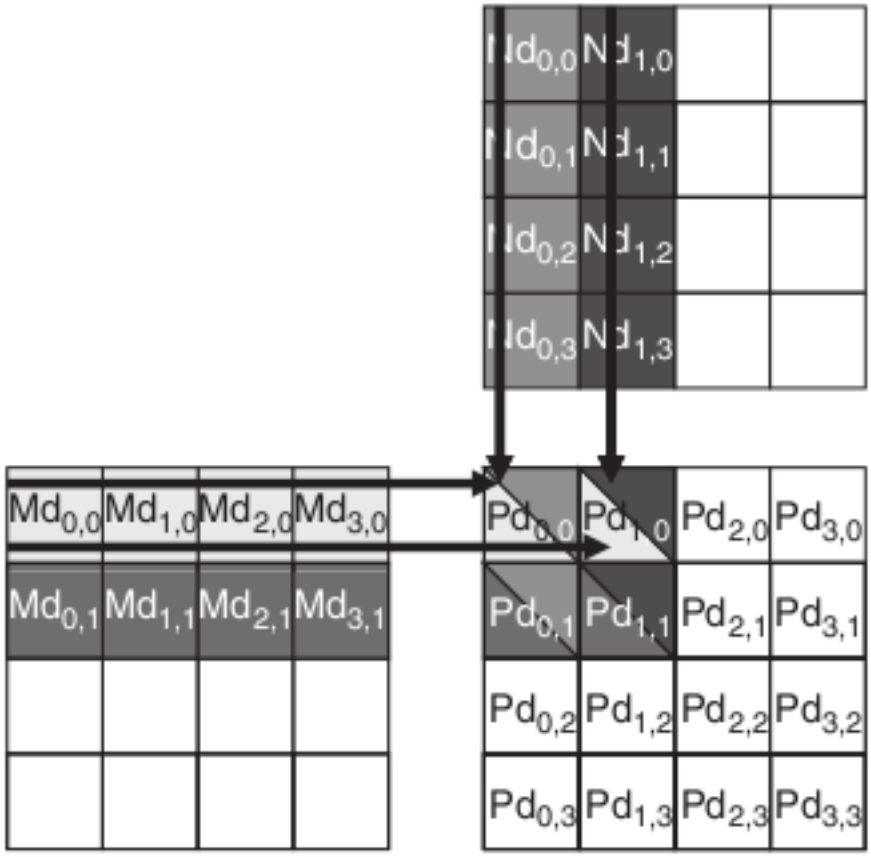

Figure 4.2: This figure is from the website http://www.elsevierdirect.com/v2/companion.jsp?ISBN=9780123814722, originally found in the book Programming Massively Parallel Processors: A Hands-on Approach.

the output matrix is marked by Pd(x,y) where x is the column number and y is the row number. Lets take a look at the block which has element Pd(0,0),Pd(1,0),Pd(0,1) and Pd(1,1).

As you can see from graph, to compute four elements in this block, we need not only to access the corresponding block in input matrix M and input matrix N, but also the block to right of the corresponding block in matrix M and the block below the corresponding block in matrix N. That is in total 4 blocks of data need to be loaded. What if the maximum capacity of shared memory per block can only hold 2 blocks of data?

The solution is simple. All threads within a block can first collaborate together to load some portion of data from global memory. This can be easily done by every thread in the block load one element from both input matrices into shared memory. In our example, thread(0,0) loads Md(0,0) and Nd(0,0); thread(1,0) loads Md(1,0) and Nd(1,0); thread(0,1) loads Md(0,1) and Nd(0,1); finally threads(1,1) loads Md(1,1) and Nd(1,1).Then we use these data to do some computations in each thread even though this is enough to give the final results. We can always let each threads to remember the running sum. After the computation, We can delete the data in shared memory because we do not need them any more. Actually, you don't even need to *delete* them, you can just load new data into it and old data will be erased automatically.

Then we can load more data from global memory to shared memory. This time, however, we cannot have each thread in the block load corresponding elements in input matrices. In our example, thread(0,0) loads Md(2,0) and Nd(0,2); thread(1,0) loads Md(3,0) and Nd(1,2); thread(0,1) loads Md(2,1) and Nd(0,3); finally threads(1,1) loads Md(3,1) and Nd(1,3). We can use this data for further computations. By the time we finished loading all the data to the shared memory from global memory, all the threads would have final results in the running sums. This way, we can use shared memory to increase the speed but not suffer from the limited storage capacity.

	Phase 1			Phase 2		
$T_{0,0}$	$\mathbf{Md_{0,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,0}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	$\mathbf{Md_{2,0}}$ ↓ $Mds_{0,0}$	$\mathbf{Nd_{0,2}}$ ↓ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md_{1,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,0}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	$\mathbf{Md_{3,0}}$ ↓ $Mds_{1,0}$	$\mathbf{Nd_{1,2}}$ ↓ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md_{0,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,1}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + Mds_{11}*$Nds_{0,1}$	$\mathbf{Md_{2,1}}$ ↓ $Mds_{0,1}$	$\mathbf{Nd_{0,3}}$ ↓ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md_{1,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,1}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{Md_{3,1}}$ ↓ $Mds_{1,1}$	$\mathbf{Nd_{1,3}}$ ↓ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time →

Figure 4.3: This figure is from the website http://www.elsevierdirect.com/v2/companion.jsp?ISBN=9780123814722, originally found in the book Programming Massively Parallel Processors: A Hands-on Approach.

We call each data loading and computing process a phase. Therefore, in the previous example, we went through 2 phases before we have our final results. It is not hard to find out that by using shared memory, we can reduce the number if times of accessing global memory from *Width* times for every column or row to *Width/blockDim* times.

1756.50 Back to our problem, we are dealing with input matrices with the size of *1024 x 1024* and we are using blocks with the size of *32 x 32*. We can potentially reduce the global memory accessing time to 1/32 of the original.

4.5.1 The Device Code

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // declare cache in the shared memory
  __shared__ float Mds[blockD][blockD];
  __shared__ float Nds[blockD][blockD];

  // keep track of column index of the Pd element using thread index
  int x = threadIdx.x + blockIdx.x * blockDim.x; // x is column
  // keep track of row index of the Pd element using thread index
  int y = threadIdx.y + blockIdx.y * blockDim.y; // y is row

  float Pvalue = 0;
  // Loop over the Md and Nd block dimension required to compute the Pd element
  for (int m = 0; m < Width/blockD; m++){

    // collaboratively loading of Md and Nd blocks into shared memory
    Mds[threadIdx.y][threadIdx.x] = Md[y * Width + (m * blockD + threadIdx.x)];
    Nds[threadIdx.y][threadIdx.x] = Md[(m * blockD + threadIdx.y) * Width + x];
    __syncthreads();

    // keep track of the running sum
    for (int k = 0; k < blockD; k++)
      Pvalue += Mds[threadIdx.y][k] * Nds[k][threadIdx.x];
    __syncthreads();
  }

  // write back to the global memory
  Pd[y * Width + x] = Pvalue;
}
```

With all the explanations before, you should understand this device code easily.

If you are careful enough, you may see that I used a variable called **blockD**. This variable was defined at the very beginning of the source code.

```
#define blockD 32
```

There are two thing you need to pay attention when defining this variable. First is that you should not assign this variable with a number that is too big. This variable is used to define the dimension of each block. The reason we are using block is to reduce the size of data transfer between global memory and shared memory every time. If you assign too big a number to it, you are risking running out of share memory.

Another thing is that some of you might wonder why we are using *blockD* to represent block dimension instead of using blockDim. Well, blockDim is a built-in function used by CUDA C, you can define blockDim as a constant in here, but the built-in function will fail if you call it since you define a function equals a constant. The point I am trying to make here is that be very careful when you are choosing your variable names. CUDA C, different from standard C, has more built-in functions and you might bump into one or two while you are naming variables.

4.5.2 About blockDim and matrix dimension

Another thing needs mentioning is that while choosing the value of *blockD*, it is crucial for you to reference the matrix dimension before you decide which number to assign to *blockD*. Different from the global memory version and CPU

version, shared memory version requires threads within a block to work collaboratively to load part of the data to shared memory each time. This means matrix's dimension should be multiples of *blockD* so that threads in a block can load same amount of data each time.

Further, recall that in the device code, we have expression like

```
  for (int m = 0; m < Width/blockD; m++){
```

where we have *Width* divided by *blockD*. If you pick *Width* that is not dividable by *blockD*, program will return weird thing because it expects a integer coming out of this line of code, instead of some floats.

As this program is using 1024 as *Width*, we picked 32 as the *blockD* value. If you use 1000 instead if 1024 for *Width* and print out the result, you will see weird results. However, if you happen to have matrices with dimension of 1000, you should use 25 instead of 32 as the *blockD* value.

4.5.3 The Host Code

```
main(void){

    void MatrixMultiplication(float *, float *, float *, int);

    const int Width = 1024;

    int size = Width * Width * sizeof(float);
    float *M, *N, *P;

    // allocate memory on the CPU
    M = (float*)malloc(size);
    N = (float*)malloc(size);
    P = (float*)malloc(size);

    // initialize the matrices
    for (int y=0; y<Width; y++) {
            for (int x=0; x<Width; x++){
                        M[y*Width + x] = x + y*Width;
                    N[y*Width + x] = x + y*Width;
           }
    }

    MatrixMultiplication(M, N, P, Width);

    // free the memory allocated on the CPU
    free( M );
    free( N );
    free( P );

    return 0;
}
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
  // declare cache in the shared memory
  __shared__ float Mds[blockD][blockD];
  __shared__ float Nds[blockD][blockD];

  // keep track of column index of the Pd element using thread index
  int x = threadIdx.x + blockIdx.x * blockDim.x; // x is column
  // keep track of row index of the Pd element using thread index
  int y = threadIdx.y + blockIdx.y * blockDim.y; // y is row
```

```
  float Pvalue = 0;
  // Loop over the Md and Nd block dimension required to compute the Pd element
  for (int m = 0; m < Width/blockD; m++){

    // collaboratively loading of Md and Nd blocks into shared memory
    Mds[threadIdx.y][threadIdx.x] = Md[y * Width + (m * blockD + threadIdx.x)];
    Nds[threadIdx.y][threadIdx.x] = Md[(m * blockD + threadIdx.y) * Width + x];
    __syncthreads();

    // keep track of the running sum
    for (int k = 0; k < blockD; k++)
      Pvalue += Mds[threadIdx.y][k] * Nds[k][threadIdx.x];
    __syncthreads();
  }

  // write back to the global memory
  Pd[y * Width + x] = Pvalue;
}

void MatrixMultiplication(float *M, float *N, float *P, int Width) {

    int size = Width * Width * sizeof(float);
    float *Md, *Nd, *Pd;

    // capture start time
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    // allocate memory on the GPU
    HANDLE_ERROR( cudaMalloc((void**)&Md, size) );
    HANDLE_ERROR( cudaMalloc((void**)&Nd, size) );
    HANDLE_ERROR( cudaMalloc((void**)&Pd, size) );

    // transfer M and N to device memory
    HANDLE_ERROR( cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice) );
    HANDLE_ERROR( cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice) );

    // kernel invocation code
    dim3 dimBlock(blockD, blockD);
    dim3 dimGrid(Width/blockD, Width/blockD);
    MatrixMulKernel<<<dimGrid, dimBlock>>>( Md, Nd, Pd, Width);

    // transfer P from device
    HANDLE_ERROR( cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost) );

    // get stop time, and display the timing results
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    float   elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                        start, stop ) );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );

    // free the memory allocated on the GPU
    HANDLE_ERROR( cudaFree(Md) );
    HANDLE_ERROR( cudaFree(Nd) );
```

```
    HANDLE_ERROR( cudaFree(Pd) );

    // destroy events to free memory
    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );
}
```

There is nothing worth mentioning in the host code because it is almost identical to what we had in the previous example.

4.5.4 Performance

While testing the performance, we used 1024 as Width same as the number used in the CPU baseline program. We conducted 5 tests and the results are below.

- 1. 24.4 ms
- 2. 24.2 ms
- 3. 24.3 ms
- 4. 24.4 ms
- 5. 24.4 ms
- Average: 24.34 ms

Compared the CPU program, our GPU program is **1664** times faster.

CHAPTER

FIVE

RAY TRACING AND CONSTANT MEMORY

5.1 Acknowledgement

The examples used in this chapter are based on examples in CUDA BY EXAMPLE: An Introduction to General-Purpose GPU Programming, written by Jason Sanders and Edward Kandrot, and published by Addison Wesley.

5.2 Basics of Ray Tracing

First of all, what is ray tracing. Well, ray tracing is how you reflect a scene consisting three-dimensional objects on a two dimensional image. This is similar to the games you play on your computer, except your games might use a different method. However, the basic idea behind is the same.

How does ray tracing work? It is actually pretty simple. In the two-dimensional image, you place a imaginary camera in there. Just like most real cameras, this imaginary camera contains light sensor as well. To produce a image, all we have to do is determine what light would hit our camera. The camera, on the other hand, would automatically record the color and light intensity of the ray hit it and produce exact same color and light intensity on the corresponding pixel.

Furthermore, deciding which ray would hit the camera is painstaking. So our clever computer scientist came up with an idea. Rather than deciding which ray would hit our camera, we can imagine shooting out a ray from our camera into the scene consisting three-dimensional objects. In other words, our imaginary camera is acting as an eye and we are now trying to find out what the eye is looking at. To seen what the eye is seeing, all we need to do is trace the ray shot out from the camera until it hits an object in our three-dimensional scene. We then record the color of the object and assign the color to the pixel. As you can see, most of the work in ray tracing is just deciding how the rays shot out and the objects in the scene would interact.

5.3 Notes for Compile

Before this chapter, we use the following code to compile CUDA code.

> nvcc -o example_name example_name.cu

However, since we are using CUDA to produce images in this chapter, we need to use different code for compiling. Shown as follow

> nvcc -lglut -o example_name example_name.c

5.4 Ray Tracing Without Constant Memory

In our example, we will create a scene with 20 random spheres. They are placed in a cube with dimension 1000 x 1000 x 1000. The center of the cube is at the origin. All the spheres are random in size, position as well as color. We then place the camera on a random place on z-axis and fix it facing origin. Later on, all we need to do is to fire a ray from each pixel and keep tracing it until it hits one of the objects. We also need to keep track of the depth of the ray. Since one ray can hit more than one objects, we only need to record the nearest object and its color.

Ray Tracing Without Constant Memory source file: `ray_noconst.cu`

5.4.1 Structure Code

We first create a data structure Sphere. Just like standard C, you can also create data structures in CUDA C.

```
struct Sphere {

    float   r,b,g;// color of the sphere
    float   radius;
    float   x,y,z;// coordinate of the center

    // will return the distance between imaginary camera and hit
    __device__ float hit( float ox, float oy, float *n ) {
        float dx = ox - x; // distance on x-axis
        float dy = oy - y; // distance on y-axis
        // if (dx*dx + dy*dy > radius*radius), ray will not hit sphere
        if (dx*dx + dy*dy < radius*radius) {
            float dz = sqrtf( radius*radius - dx*dx - dy*dy );
            // n is used in visual effect
            *n = dz / sqrtf( radius * radius );
            return dz + z;
        }
        return -INF;
    }
};
```

Inside the data structure, we stores the coordinate of the center of the Sphere as *(x, y, z)* and its color as *(r, g, b)*. You can see we also defined a method called hit. This method will decide whether the ray shot out from point *(ox, oy)* can hit the Sphere defined in the structure or not. The basic idea is simple, you can think of we project the sphere on our two-dimensional image. We first find out the distance between center of the Sphere and point *(ox, oy)* on the x-axis. We then do the same thing on the y-axis. Using Pythagorean theorem, we can find out the distance between center of the sphere and point *(ox, oy)*. If this distance is less than radius, then we are sure about the ray hitting the sphere. We then use this distance and the sphere's coordinate on z-axis to find out the distance between point *(ox, oy)* and sphere. On the other hand, it they don't intersect, we will assign negative infinity as the distance.

You may also noticed two other things left unexplained here. First, you can see that we add a qualifier **__device__** before the method definition.

```
    __device__ float hit( float ox, float oy, float *n ) {
```

Well, the purpose of this qualifier is to tell the kernel that this method should executes on the device (our GPU) instead of on the host (our CPU).

Second, you may also find the following line intrigued.

```
            *n = dz / sqrtf( radius * radius );
```

The value *n* is used to provide a better visual effect. You can see that we defined it as the percentage of distance between point *(ox, oy)* and center of sphere out of the radius. We will add this value to later code so that you can see center of the circle clearer while the edge of the sphere dimmer.

5.4.2 Device Code

```
__global__ void kernel( Sphere *s, unsigned char *ptr ) {

    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    // this is a linear offset into output buffer
    int offset = x + y * blockDim.x * gridDim.x;

    // shift the (x,y) image coordinate so that z-axis go through center
    float   ox = (x - DIM/2);
    float   oy = (y - DIM/2);

    float   r=0, g=0, b=0;// set the background to black
    float   maxz = -INF;
    for(int i=0; i<SPHERES; i++) {
        float   n;
        float   t = s[i].hit( ox, oy, &n ); // return the distance
        if (t > maxz) {
            float fscale = n;// improve visual effect
            r = s[i].r * fscale;
            g = s[i].g * fscale;
            b = s[i].b * fscale;
            maxz = t; // update maxz everytime a smaller distance is found
        }
    }

    // color the bitmap according to what the ray has 'seen'
    ptr[offset*4 + 0] = (int)(r * 255);
    ptr[offset*4 + 1] = (int)(g * 255);
    ptr[offset*4 + 2] = (int)(b * 255);
    ptr[offset*4 + 3] = 255;
}
```

On the GPU, we will assign each pixel a thread which is used for ray tracing computation. Therefore, in the first several lines of code,

```
    // map from threadIdx/BlockIdx to pixel position
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
```

we first need to map each thread's threadIdx and blockIdx to the pixel position on the bitmap, which is represented by *(x, y)*. Then, we need to create a linear offset so that when the kernel is coloring the pixel, the kernel need to know exactly which pixel it will color.

Then we shift image coordinate by *DIM/2* on the x-axis and *DIM/2* on the y-axis as well. We need to do this because the center of the bitmap is not the origin. We need the center of the bitmap to match origin's position so that the z-axis can go through the center of image.

```
    // shift the (x,y) image coordinate so that z-axis go through center
    float   ox = (x - DIM/2);
    float   oy = (y - DIM/2);
```

After the preparations, we can start our ray tracing program. We first set the *(r, g, b)* values for each pixel to be 0. We would have black background if the ray does not hit any object. Then we declare and initialize the variable *maxz*, which would hold the nearest distance between the pixel and one of the objects. Later on, each thread will call the method defined in the Sphere data structure. The method would use the *(ox, oy)* parameter passed by the thread to first decide whether one object will intersect the ray or not and second decide the distance if they intersect. The method will loop over all 20 spheres.

```
            float fscale = n;// improve visual effect
            r = s[i].r * fscale;
            g = s[i].g * fscale;
            b = s[i].b * fscale;
```

In the several lines of code above, you can see that we assign the actual *(r, g, b)* value according to the *(r, g, b)* value in the structure. We also multiplied a constant *fscale* to it. When we see a sphere from above, the nearest point aligned with your eye and the sphere center will be closer to you. On the other hand, the edge of the sphere will appear to be a little bit far away. When we multiply *fscale* to the *(r, g, b)* values, what we are trying to do is to create this effect.

```
    // color the bitmap according to what the ray has 'seen'
    ptr[offset*4 + 0] = (int)(r * 255);
    ptr[offset*4 + 1] = (int)(g * 255);
    ptr[offset*4 + 2] = (int)(b * 255);
    ptr[offset*4 + 3] = 255;
```

The last few line would be just color the the bitmap. Nothing needs to be clarified in these lines of code.

5.4.3 Host Code

```
int main( void ) {

    // declare the data block and other needed variables
    DataBlock   data;
    CPUBitmap bitmap( DIM, DIM, &data );
    unsigned char    *dev_bitmap;
    Sphere           *s;

    // allocate temp memory for the Sphere dataset on CPU
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );

    // initialize the Sphere dataset
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }

    // capture the start time
    cudaEvent_t     start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
```

```
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    // allocate memory on the GPU for the output bitmap
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap, bitmap.image_size() ) );

    // allocate memory for the Sphere dataset on GPU
    HANDLE_ERROR( cudaMalloc( (void**)&s, sizeof(Sphere) * SPHERES ) );

    // transfer the initialized Sphere dataset from CPU memory to GPU memory
    HANDLE_ERROR( cudaMemcpy( s, temp_s, sizeof(Sphere) * SPHERES,
                              cudaMemcpyHostToDevice ) );

    // generate a bitmap from our sphere data
    dim3    grids(DIM/32,DIM/32);
    dim3    threads(32,32);
    kernel<<<grids,threads>>>( s, dev_bitmap );

    // copy our bitmap back from the GPU for display
    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                              bitmap.image_size(),
                              cudaMemcpyDeviceToHost ) );

    // get stop time, and display the timing results
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    float   elapsedTime;
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                        start, stop ) );
    printf( "Time to generate:  %3.1f ms\n", elapsedTime );

    // free CPU memory
    free( temp_s );

    // free GPU memory
    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );

    HANDLE_ERROR( cudaFree( dev_bitmap ) );
    HANDLE_ERROR( cudaFree( s ) );

    // display
    bitmap.display_and_exit();
}
```

There is nothing worth mentioning about the host code. You first declare the data block and the variables. Then you allocate memory on both CPU and GPU for those variables. Then you can initialize some variables, the 20 spheres in this case on the CPU and then transfer them to the GPU memory. Later on you can call the kernel invocation code and let GPU finish the hard work. Finally, you transfer the bitmap back to CPU and display the bitmap.

5.4.4 Performance

We conducted 5 tests and the results are below.

- 1. 6.7 ms
- 2. 6.8 ms
- 3. 6.8 ms

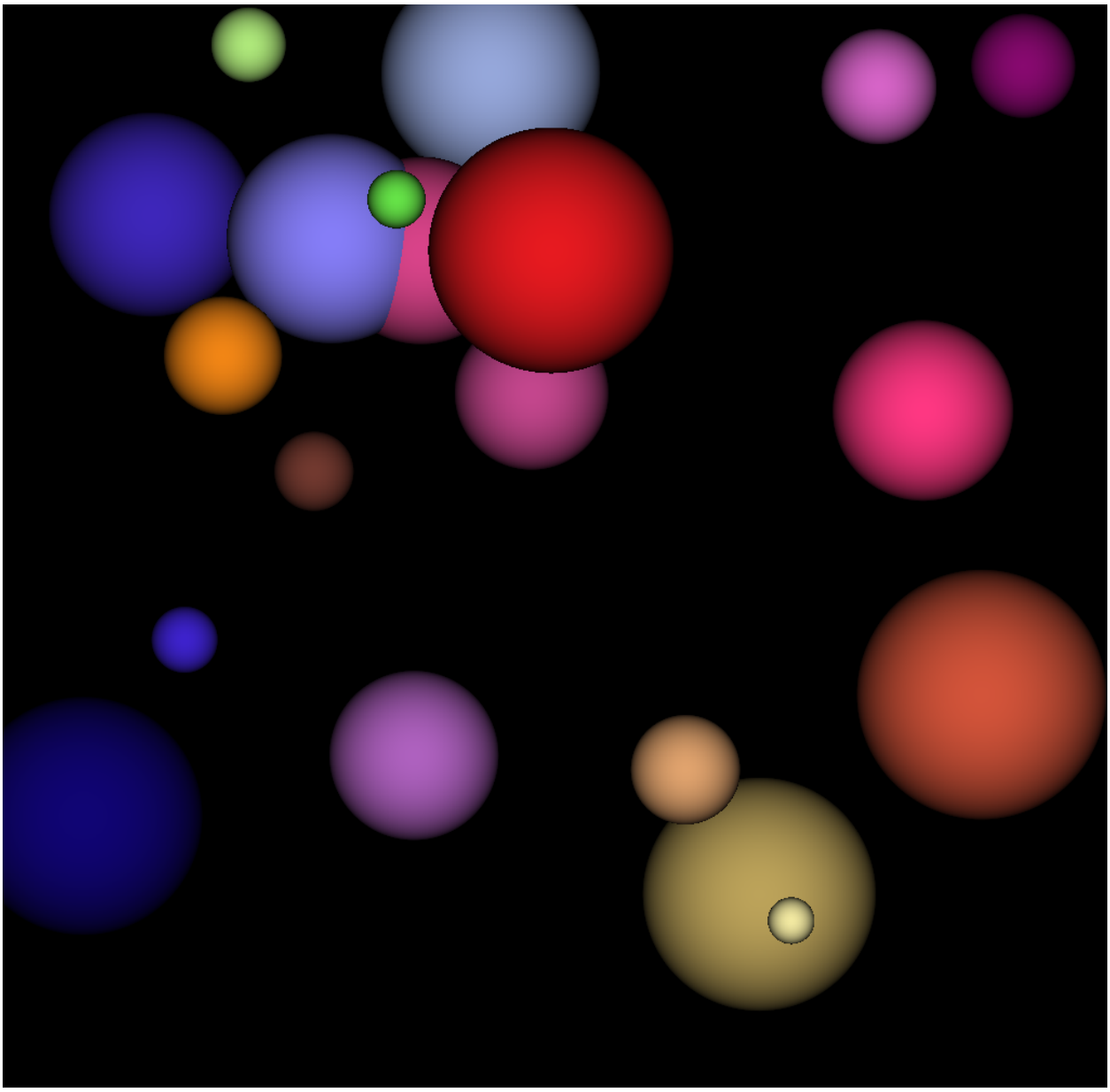

Figure 5.1: A screenshot from the ray tracing example

- 4. 6.7 ms
- 5. 6.8 ms
- Average: 6.76 ms

5.5 Constant Memory

We have mentioned that there are several types of memory in CUDA architecture. Till now, we have seen global memory and shared memory. This time, we will explore the characteristics of constant memory.

By its name, constant memory is designed to store variables that will not change when the kernel is executing commands. Constant memory is located in global memory, which means constant variables are stored in the global memory as well. However, constant variables are cached for higher access efficiency. Just like shared memory, there is always price come with faster access speed. The CUDA architecture provides only 64KB of space for global memory. Therefore, constant memory is not designed to store large dataset.

5.6 Ray Tracing With Constant Memory

In the example of ray tracing, we will see how to improve program efficiency by using constant memory. We do this by store 20 sphere object in the constant memory for faster access. In our example, every pixel of the image needs to access 20 sphere objects over the course of kernel execution. If we have a bitmap of the size *1024x1024*, we are looking at over one million times of access for each of the sphere.

Ray Tracing With Constant Memory source file: `ray.cu`

5.6.1 Constant Memory Declaration

```
__constant__ Sphere s[SPHERES]; // declare spheres in constant memory
```

This line of code shows you how to declare variables in constant memory. The only difference is that you have to add **__constant__** qualifier before the declaration.

5.6.2 Structure & Device Code

The device code and the code to create structure are exactly the same as the version not using constant memory.

5.6.3 Host Code

Most of the host code is the same as the version not using constant memory. There are only two different places. First, since we have already prepared spaces in constant memory for Sphere dataset, we do not use the command *cudaMalloc()* and *cudaMemcpy()* anymore to allocate it in global memory anymore. Second, we use the following code to copy initialized Sphere dataset to the constant memory.

```
// transfer the initialized Sphere dataset to constant memory
HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
                            sizeof(Sphere) * SPHERES) );
```

5.6.4 Performance

We conducted 5 tests and the results are below.

- 1. 6.2 ms
- 2. 6.1 ms
- 3. 6.3 ms
- 4. 6.4 ms
- 5. 6.4 ms
- Average: 6.28 ms

Due to the small bitmap size we are using, the improvement is not significant.