# IMAGE PROCESSING WITH CUDA

by

Jia Tse

Bachelor of Science,
University of Nevada, Las Vegas
2006

A thesis submitted in partial fulfillment of
the requirements for the

**Master of Science Degree in Computer Science**

**School of Computer Science**
**Howard R. Hughes College of Engineering**
**The Graduate College**

**University of Nevada, Las Vegas**
**August 2012**


THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

**Jia Tse**

entitled

## Image Processing with Cuda

be accepted in partial fulfillment of the requirements for the degree of

**Master of Science in Computer Science**
School of Computer Science

Ajoy K. Datta, Committee Chair

Lawrence L. Larmore, Committee Member

Yoohwan Kim, Committee Member

Venkatesan Muthukumar, Graduate College Representative

Thomas Piechota, Ph. D., Interim Vice President for Research and Graduate Studies and Dean of the Graduate College

**August 2012**

# Abstract

This thesis puts to the test the power of parallel computing on the GPU against the massive computations needed in image processing of large images. The GPU has long been used to accelerate 3D applications. With the advent of high level programmable interfaces, programming to the GPU is simplied and is being used to accelerate a wider class of applications. More specifically, this thesis focuses on CUDA as its parallel programming platform.

This thesis explores on the possible performance gains that can be achieved by using CUDA on image processing. Two well known algorithms for image blurring and edge detection is used in the experiment. Benchmarks are done between the parallel implementation and the sequential implementation.

# Acknowledgements

I would like to express my deepest sincere gratitude to my adviser Dr. Ajoy K. Datta for sticking with me through this entire time. He is one of the best cs professors at UNLV, and I consider myself fortunate to be one of his students. His patience and guidance is what made this thesis possible.

I would also like to thank Dr. Larmore, Dr. Kim and Dr. Muthukumar for their time in reviewing my report and their willingness to serve on my committee.

I thank my family and friends for their unconditional support in finishing this thesis.

Jia Tse

*University of Nevada, Las Vegas*

*August 2012*

# Contents

# List of Tables

# List of Figures

# Listing

# Chapter 1

# Introduction

Graphics cards are widely used to accelerate gaming and 3D graphics applications. The GPU (Graphics Processing Unit) of a graphics card is built for compute-intensive and highly parallel computations. With the prevalence of high level APIs (CUDA - Compute Unified Device Architecture), the power of the GPU is being leveraged to accelerate more general purpose and high performance applications. It has been used in accelerating database operations[1], solving differential equations[2], and geometric computations[3].

Image processing is a well known and established research field. It is a form of signals processing in which the input is an image, and the output can be an image or anything else that undergoes some meaningful processing. Altering an image to be brighter, or darker is an example of a common image processing tool that is available in basic image editors.

Often, processing happens on the entire image, and the same steps are applied to every pixel of the image. This means a lot of repetition of the same work. Newer technology allows better quality images to be taken. This equates to bigger files and longer processing time. With the advancement of CUDA, programming to the GPU is simplified. The technology is ready to be used as a problem solving tool in the field of image processing.

This thesis shows the vast performance gain of using CUDA for image processing. Chapter

two gives an overview of the GPU, and gets into the depths of CUDA, its architecture and its programming model. Chapter three consists of the experimental section of this thesis. It provides both the sequential and parallel implementations of two common image processing techniques: image blurring and edge detection. Chapter four shows the experiment results and the thesis is concluded in chapter five.

# Chapter 2

# CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVidia for massively parallel high-performance computing. It is the compute engine in the GPU and is accessible by developers through standard programming languages. CUDA technology is proprietary to NVidia video cards.

NVidia provides APIs in their CUDA SDK to give a level of hardware extraction that hides the GPU hardware from developers. Developers no longer have to understand the complexities behind the GPU. All the intricacies of instruction optimizations, thread and memory management are handled by the API. One benefit of the hardware abstraction is that this allows NVidia to change the GPU architecture in the future without requiring the developers to learn a new set of instructions.

## 2.1 GPU Computing and GPGPU

The Graphics Processing Unit (GPU) is a processor on a graphics card specialized for compute-intensive, highly parallel computation. it is primarily designed for transforming, rendering and accelerating graphics. It has millions of transistors, much more than the Central Processing Unit

(CPU), specializing in floating point arithemetic. Floating point arithemetic is what graphics rendering is all about. The GPU has evolved into a highly parallel, multithreaded processor with exceptional computational power. The GPU, since its inception in 1999, has been a dominant technology in accelerated gaming and 3D graphics application.

The main difference between a CPU and a GPU is that a CPU is a serial processor while the GPU is a stream processor. A serial processor, based on the Von Neumann architecture executes instructions sequentially. Each instruction is fetched and executed by the CPU one at a time. A stream processor on the other hand executes a function *(kernel)* on a set of input data *(stream)* simultaneously. The input elements are passed into the kernel and processed independently without dependencies among other elements. This allows the program to be executed in a parallel fashion.

Due to their highly parallel nature, GPUs are outperforming CPUs by an astonishing rate on floating point calculations (Figure 2.1)[4]. The main reason for the performance difference lies in the design philosophies between the two types of processors (Figure 2.2)[4]. The CPU is optimized for high performance on sequential operations. It makes use of sophisticated control logic to manage the execution of many threads while maintaining the appearance of a sequential execution. The large cache memories used to reduce access latency and slow memory bandwidth also contribute to the performance gap.

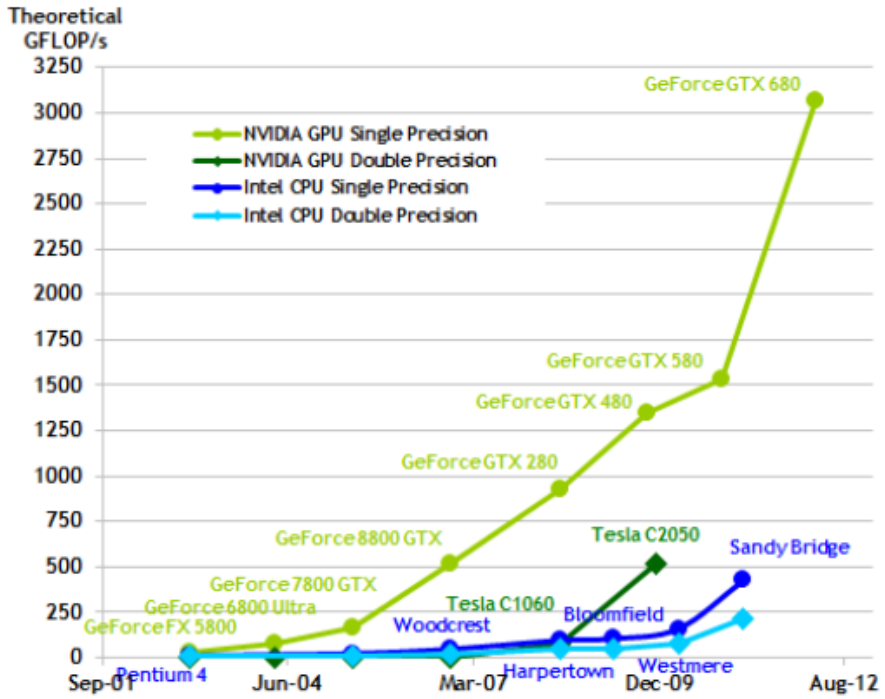


Figure 2.1: GPU vs CPU on floating point calculations


Figure 2.2: CPU and GPU chip design

The design philosophy for GPUs on the other hand is driven by the fast growing video game industry that demands the ability to perform massive floating-point calculations in advanced video games. The motivation is to optimize the execution of massive number of threads, minimize control logic, and have small memory caches so that more chip area can be dedicated to floating-point

calculations. This trade-off makes the GPU less efficient at sequential tasks designed for the CPU.

Recognizing the huge potential performance gains, developers hungry for high performance began using the GPU for non graphics purposes. Improvements in the programmability of graphics hardware further drove GPU programming. Using high-level shading languages such as DirectX, OpenGL and Cg, various data parallel algorithms can be mapped onto the graphics API. A traditional graphics shader is hardwired to only do graphical operations, but now it is used in everyday general-purpose computing. Researchers have discovered that the GPU can accelerate certain problems by over an order of magnitude over the CPU. Using the GPU for general purpose computing creates a phenomenon known as GPGPU.

GPGPU is already being used to accelerate applications over a wide range of cross-disciplinary fields. Many applications that process large data sets take advantage of the parallel programming model by mapping its data elements to parallel processing threads. Purcell and Carr illustrates how this mapping is done for ray-tracing[5][6]. Similarly, this concept can be applied to other fields. The GPU is also being adopted in accelerating database operations[1][7][8][9][10][11]. Work has been done using the GPU for geometric computations[3][12][13][14], linear algebra[15], solving partial differential equations[2][16] and solving matrices[17][18]. As the GPU's floating-point processing performance continues to outpace the CPU, more data parallel applications are expected to be done on the GPU.

While the GPGPU model has its advantages, programmers initially faced many challenges in porting algorithms from the CPU over to the GPU. Because the GPU was originally driven and designed for graphics processing and video games, the programming environment was tightly constrained. The programmer requires a deep understanding of the graphics API and GPU architecture. These APIs severely limit the kind of applications that can be written on this platform. Expressing algorithms in terms of vertex coordinates and shader programs increased programming complexity. As a result, the learning curve is steep and GPGPU programming is not widespread.

Higher-level language constructs are built to abstract the details of shader languages. The *Brook*

*Specifications* is created in 2003 by Stanford as an extension of the C language to efficiently incorporate ideas of parallel computing into a familiar language[19]. In 2006 a plethora of platforms including Microsoft's *Accelerator*[20], the *RapidMind Multi-Core Development Platform*[21] and the *PeakStream Platform*[22] emerge. *RapidMind* is later acquired by Intel in 2009 and *Peakstream* is acquired by Google in 2007. By 2008 Apple released OpenCL[23], and AMD released its Stream Computing software deveopment kit (SDK) that is built on top of the *Brook Specifications*. Microsoft released *DirectCompute* as part of its DirectX 11 package. NVidia released its *Compute Unified Device Architecture (CUDA)* as part of its parallel computing architecture. Popular commercial vendors such as Adobe and Wolfram are releasing cuda-enabled versions of their products (Figure 2.3)[24].

It is important to note that GPU processing is not meant to *replace* CPU processing. There are simply algorithms that run more effiicently on the CPU than on the GPU. Not everything can be executed in a parallel manner. GPUs, however, offer an efficient alternative for certain types of problems. The prime candidates for GPU parallel processing are algorithms that have components that require a repeated execution of the same calculations, and those components must be able to be executed independently of each other. Chapter 3 explores image processing algorithms that fit this paradigm well.

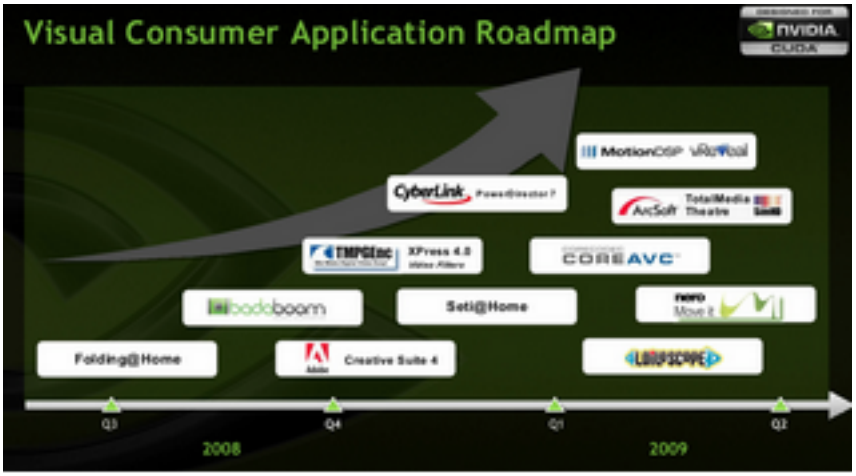


Figure 2.3: Products supporting CUDA

## 2.2 CUDA architecture

A typical CUDA architecture consists of the components as illustrated in Figure 2.4[25]. The Host CPU, Bridge and System memory are external to the graphics card, and are collectively referred to as the *host*. All remaining components forms the GPU and the CUDA architecture, and are collectively referred to as the *device*. The host interface unit is responsible for communication such as responding to commands, and facilitating data transfer between the host and the device.

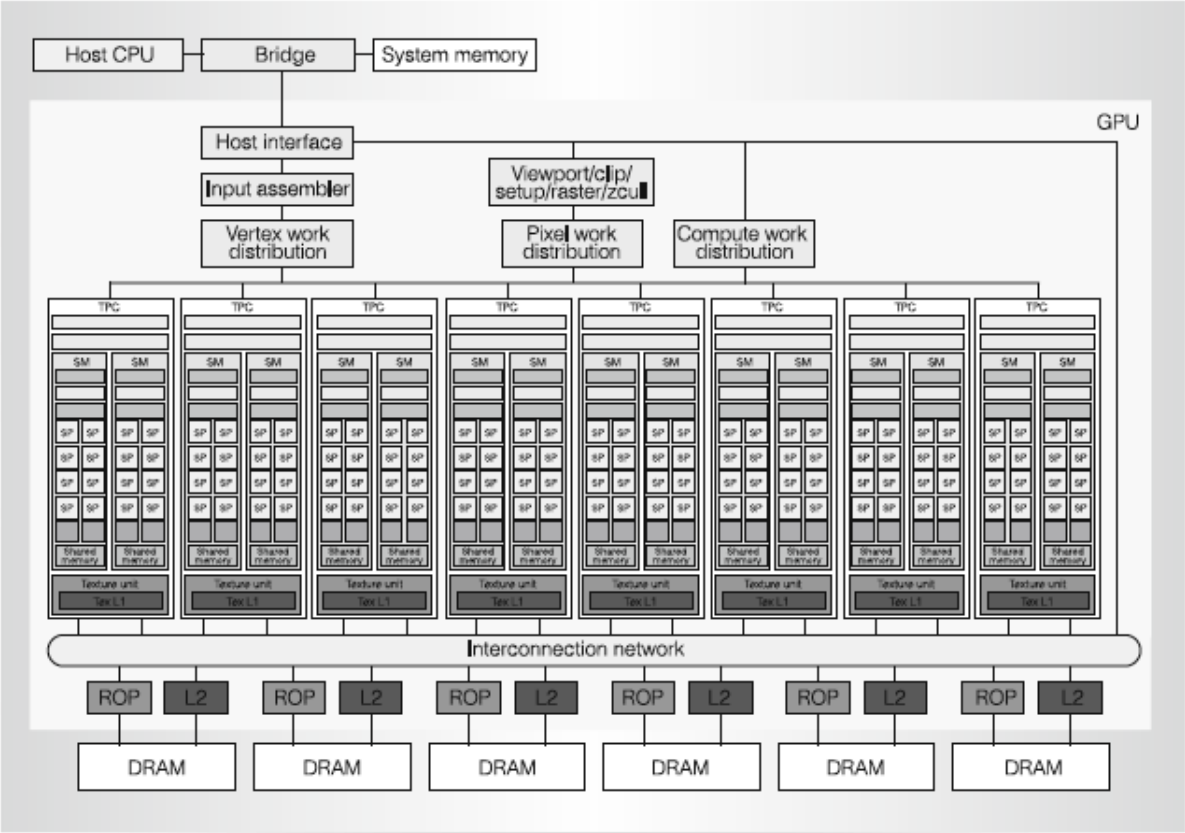


Figure 2.4: GPU Architecture. TPC: Texture/processor cluster; SM: Streaming Multiprocessor; SP: Streaming Processor

The input assembler collects geometric primitives and outputs a stream to the work distributors[25]. The work distributors forward the stream in a round robin fashion to the Streaming Processor Array (SPA). The SPA is where all the computation takes place. The SPA is an array of Texture/Processor Clusters (TPC) as shown in Figure 2.4[25]. Each TPC contains a geometry controller, a Streaming Multiprocessor (SM) controller (SMC), a texture unit and 2 SMs. The texture unit is used by the SM as a third execution unit and the SMC is used by the SM to implement external memory load, store and atomic access. A SM is a multiprocessor that executes vertex, geometry and other shader programs as well as parallel computing programs. Each SM contains 8 Streaming Processors (SP), and 2 Special Function Units (SFU) specializing in floating point functions such as square root and

transcendental functions and for attribute interpolations. It also contains an instruction cache, a constant cache, a multithreaded instruction fetch and issue unit (MT) and shared memory (Figure 2.5)[25]. Shared memory holds shared data between the SPs for parallel thread communication and cooperation. Each SP contains its own MAD and MUL units while sharing the 2 SFU with the other SPs.

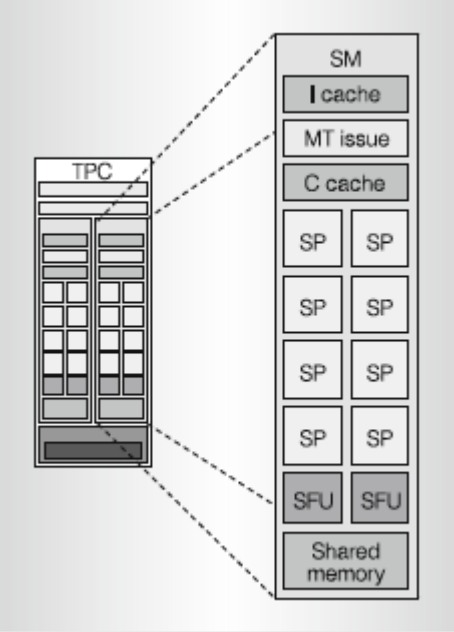


Figure 2.5: Streaming Multiprocessor

A SM can execute up to 8 thread blocks, one for each SP. It is capable of efficiently executing hundreds of threads in parallel with zero scheduling overhead. The SMs employ the Single-Instruction, Multiple-Thread (SIMT) architecture to manage hundreds of concurrent threads[26]. GTX-200 series is equipped with 16 KB of shared memory per SM. In the GeForce 8-series GPU, each SP can handle up to 96 concurrent threads for a total of 768 threads per SM[27]. On a GPU with 16 SMs, up to 12,288 concurrent threads can be supported.

## 2.3 CUDA Programming Model

CUDA programming is a type of heterogeneous programming that involves running code on two different platforms: a *host* and a *device*. The *host* system consists primarily of the CPU, main memory and its supporting architecture. The *device* is generally the video card consisting of a CUDA-enabled GPU and its supporting architecture.

The source code for a CUDA program consists of both the host and device code mixed in the same file. Because the source code targets two different processing architectures, additional steps are required in the compilation process. The NVidia C Compiler (NVCC) first parses the source code and creates two separate files: one to be executed by the host and one for the device. The host file is compiled with a standard C/C++ compiler which produces standard CPU object files. The device file is compiled with the CUDA C Compiler (CUDACC) which produces CUDA object files. These object files are in an assembly language known as Parallel Thread eXecution or PTX files. PTX files are recognized by device drivers that are installed with NVidia graphics cards. The two resulting file set is linked and a CPU-GPU executable is created (Figure 2.6)[28]. As shown in Figure 2.7[28] & 2.8[29], this type of architecture allows the flexibility for developers who are familiar with other languages to leverage the power of CUDA without having to learn a brand new language.

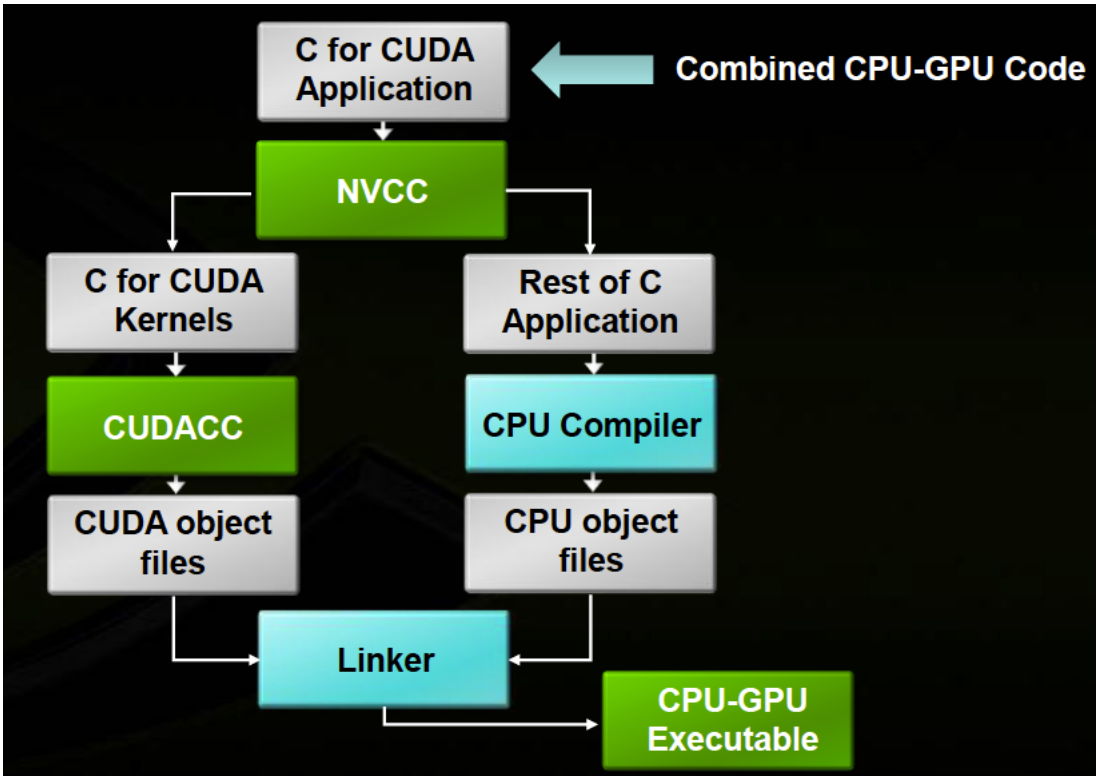


Figure 2.6: The compilation process for source file with host & device code

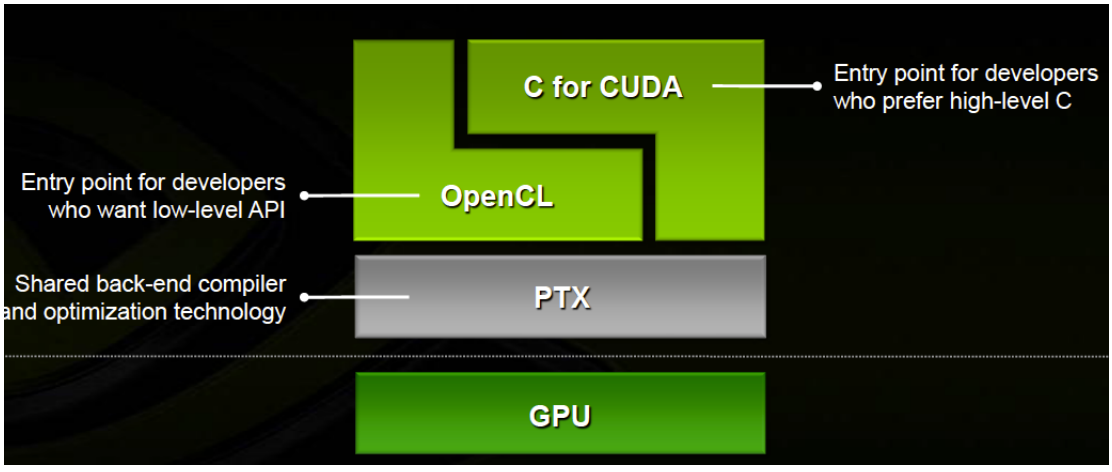


Figure 2.7: CUDA architecture

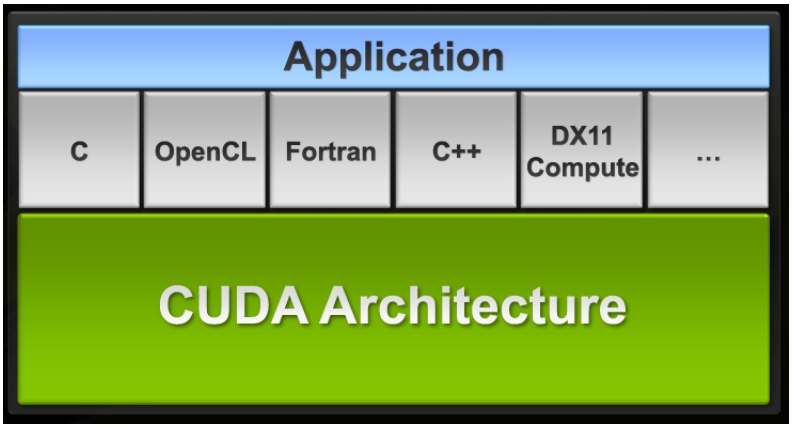


Figure 2.8: CUDA architecture

NVCC separates host from device code by identifying specific keywords that represents instructions for the device. Methods/Functions that are designed to execute on the device are called *kernels*. Kernels are typically executed by thousands to millions of threads to take advantage of data parallelism. Since all threads are executing the same code, this falls into the well known paradigm of Single Program Multiple Data (SPMD) widely used in parallel computing systems[30]. SPMD is an asynchronous version of another technique known as Single-Instruction Multiple-Data (SIMD). In SIMD, multiple processors execute the same program instructions (a function) on different data. The key difference between SIMD and SPMD is that SIMD executes the program instructions in locksteps. Every processor executes the identical instruction at any given time. SPMD however removes that restriction. This allows the possibility of having branching in the program instruction where the instructions executed by each processor is not always the same.

Listing 2.1 shows an example of a typical C program involving CUDA. __global__ is a C extension that defines a kernel. The kernel is invoked inside the main function by using the <<< ... >>> syntax. *dimblock* and *dimGrid* defines the number of threads and its configuration when it executes in the kernel. Each thread that executes the kernel is assigned a unique thread id. A particular thread within the kernel can be identified by the combination of its *blockIdx*, *blockDim* and *threadIdx*.

This allows for the control of having different threads do different work.

Listing 2.1: Sample source code with Host & Device code

```
1 //Kernel Definition
2 __global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
      {
3     int i = blockIdx.x * blockDim.x + threadIdx.x;
4     int j = blockIdx.y * blockDim.y + threadIdx.y;
5
6     if (i < N && j < N) {
7         C[i][j] = A[i][j] + B[i][j];
8     }
9 }
10
11 int main() {
12     //Kernel Invocation
13     dim3 dimBlock(16, 16);
14     dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x, (N + dimBlock.y -
           1) / dimBlock.y);
15
16     MatAdd <<< dimGrid, dimBlock >>> (A, B, C);
17 }
```

A CUDA program starts execution on the the host (Figure 2.9)[31]. When it encounters the kernel, it will launch the kernel and continues execution on the CPU without waiting for the completion of the kernel. The groups of threads created as a result of the kernel invocation is collectively referred to as a grid. The grid terminates when the kernel terminates. Currently in CUDA, only one kernel can be executed at a time. If the host encounters another kernel while a previous kernel is not yet complete, the CPU will stall until the kernel is complete. The next-generation architecture *FERMI* allows for the concurrent execution of multiple kernels.

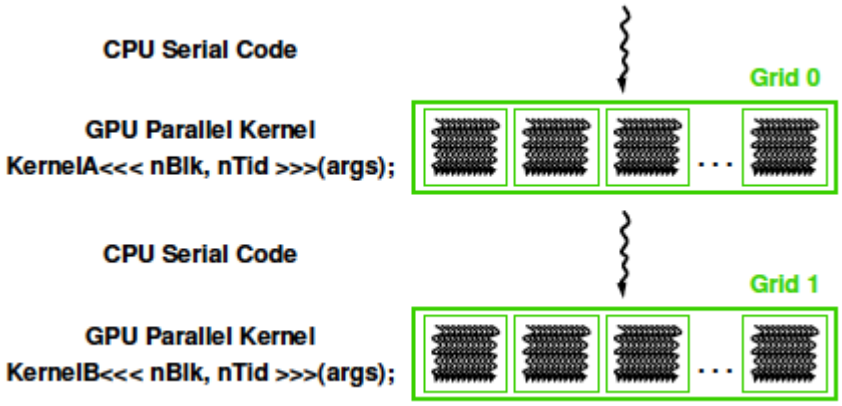


Figure 2.9: Execution of a CUDA program

In CUDA, the host and devices have separate memory spaces. Variables and data in the host memory is not directly accessible by the GPU. The data allocated on the host must first be transfered to the device memory using the CUDA API. Similarly, the results from the device must be transferred back to the host. Memory management techniques must be applied on both platforms. Listing 2.2[31] shows a snippet of operations dealing with memory on the host and device. *cudaMalloc*, *cudaMemcpy*, *cudaFree* are all CUDA APIs that allocates memory, copies memory, and frees memory respectively on the device.

Listing 2.2: Memory operations in a CUDA program

```
1  void MatrixMulOnDevice(float* M, float* N, float* P, int Width) {
2     int size = Width * Width * sizeof(float);
3
4     //1.  Load M and N to device memory
5     cudaMalloc(Md, size);
6     cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
7     cudaMalloc(Nd, size);
8     cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);
9
10    //Allocate P on the device
11    cudaMalloc(Pd, size);
12
```

```
13    //2.  Kernel invocation code here
14    //...
15
16    //3.  Read P from the device
17    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);
18    //Free device matrices
19    cudaFree(Md);
20    cudaFree(Nd);
21    cudaFree(Pd);
22 }
```

## 2.4 CUDA Thread Hierarchy

Threads on the device are automatically invoked when a kernel is being executed. The programmer determines the number of threads that best suits the given problem. The thread count along with the thread configurations are passed into the kernel. The entire collection of threads responsible for an execution of the kernel is called a grid (Figure 2.10)[4].

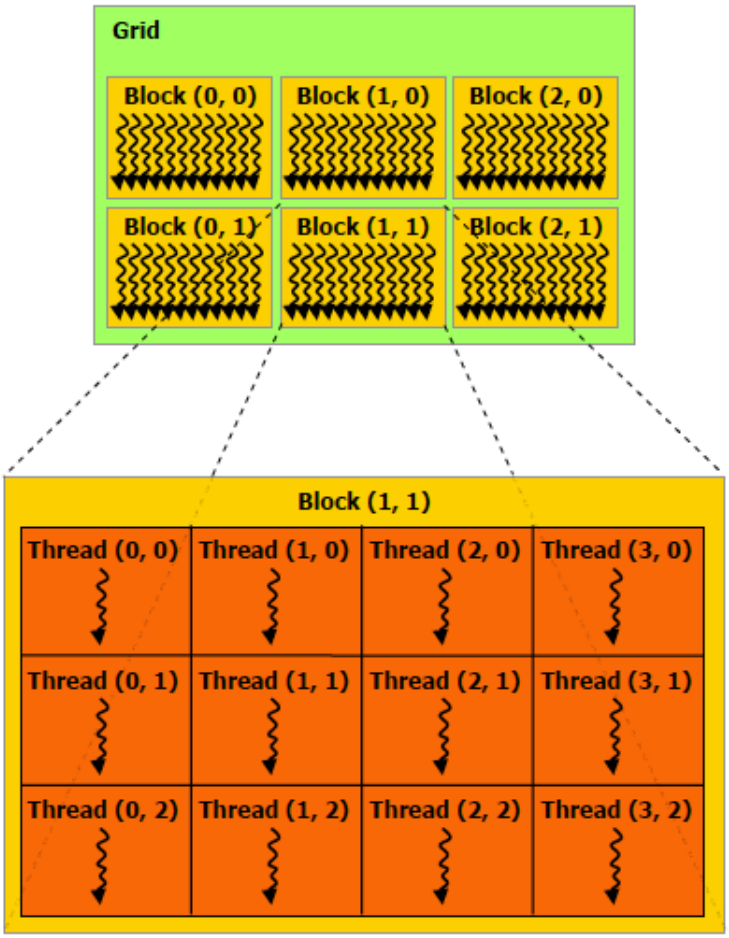


Figure 2.10: Grid of thread blocks

A grid is further partitioned and can consist of one or more thread blocks. A block is an array of concurrent threads that execute the same thread program and can cooperate in achieving a result. In Figure 2.10[4], the blocks are organized into a 2 x 3 array. A thread block can be partitioned into one, two or three dimensions, facilitating calculations dealing with vectors, matrices or fields. Each block has its own unique block identifier. All threads within a block can cooperate with each other. They can share data by reading and writing to shared memory, and they can synchronize their execution by using *__syncthreads()*. __syncthreads acts as a barrier so that all threads of the same block must wait for all threads to execute before moving forward. This ensures that all threads

have finished executing a phase of their execution in the kernel before moving on to the next phase. __synchthread is commonly used inside the kernel to coordinate read and write phases to shared memory. Since the data in the memory is shared, all threads must write first and read second.

Threads of different blocks cannot communicate with each other. In fact, thread blocks are required that they can be executed independently of other blocks, whether in series or in parallel. Like blocks, threads within a block can be strategically structured as well. Figure 2.10 shows a 3 x 4 array of threads within block (1,1). All blocks must contain the same number of threads and thread structure. Each block can have a maximum of up to 512 threads. The programmer has the freedom to structure the threads in any different combinations of up to three dimensions (512 x 1, 16 x 8 x 2, etc) as long as the total number of threads do not exceed 512. The organization of blocks and threads can be established and passed to the kernel when it is invoked by the host. this configuration is maintained throughout the entire execution of the kernel.

Block and grid dimensions can be initialized by the to type *dim3*, which is a essentially a struct with x, y, z fields. Listing 2.3 creates a 2 x 2 x 1 grid and each block has a dimension of 4 x 2 x 2. The threading configuration is then passed to the kernel. The resulting hierarchy can be graphically represented as shown in Figure 2.11[31]. Within the kernel, these information are stored as built-in variables. *blockDim* holds the dimension information of the current block. *blockIdx* and *threadIdx* provides the current block and thread index information. All *blockIdx*, *threadIdx*, *gridDim*, and *blockDim* have 3 dimensions: x, y, z. For example, block (1,1) has $blockIdx.x = 1$ and $blockIdx.y = 1$.

Listing 2.3: Invoking a kernel with a 2 x 2 x 1 grid and a 4 x 2 x 2 block

```
1    dim3 dimBlock(4,2,2);
2    dim3 dimGrid(2,2,1);
3    KernelFunction <<< dimGrid, dimBlock >>>
```

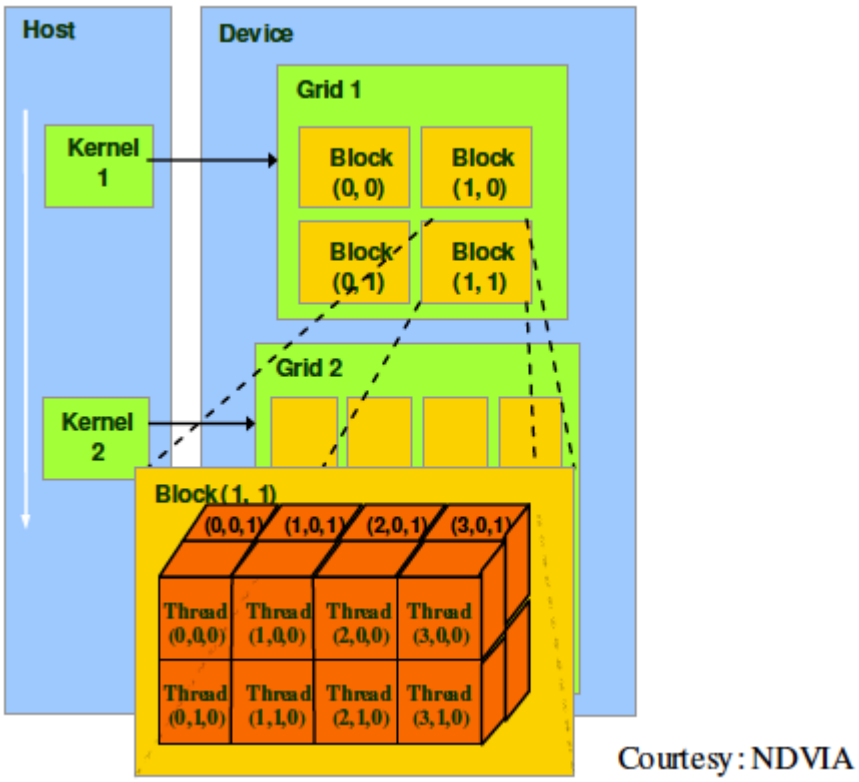


Figure 2.11: A grid with dimension (2,2,1) and a block with dimension (4,2,2)

One of the main functionality of *blockId* and *threadId* is to distinguish themselves from other threads. One common usage is to determine which set of data a thread is responsible for. Listing 2.4 is a simple example of squaring all elements of a 1-dimensional array of size 10. To do that we create a 1-dimensional grid, containing a 1-dimensional 10 x 1 block. When the *square_array* kernel is called, it generates a threading configuration resembling Figure 2.12; a 1-dimensional array of 10 threads.

Listing 2.4: A program that squares an array of numbers

```
1 __global__ void square_array (float *a, int N) {
2    int idx = blockIdx.x * blockDim.x + threadIdx.x;
3    if (idx < N) {
4       a[idx] = a[idx] * a[idx];
5    }
```

```
6 }
7
8 int main (void) {
9    ...
10   dim3 Block(10, 1);
11   dim3 Grid(1);
12   square_array <<< Grid, Block >>> (arr, 10);
13   ...
14 }
```

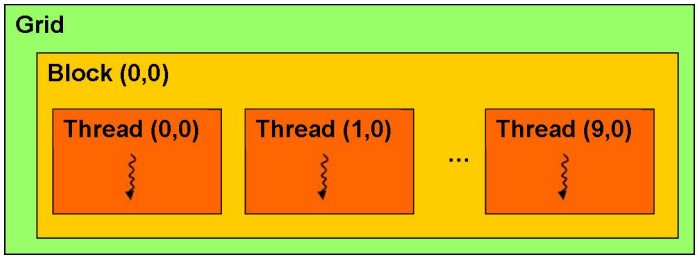


Figure 2.12: A 1-dimensional 10 x 1 block

The code in the kernel identifies a thread by using the *blockIdx.x*, *blockDim.x* and *threadIdx.x*. In this case, $blockIdx.x = 0$, $blockDim.x = 10$ and *threadIdx.x* ranges from 0,9 inclusive depending on which thread executes the kernel. Figure 2.13 is the result of executing kernel *square_array*. Each thread is responsible for computing the square of the value stored in the array at index equal to the thread id. It is easily seen that each thread can operate independently of each other. Mapping thread Ids to array indices is a common practice in parallel processing. A similar technique is used in mapping to matrices and fields.

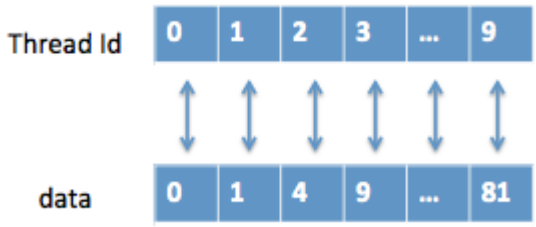


Figure 2.13: Each thread computing the square of its own value

One limitation on blocks is that each block can hold up to 512 threads. In trivial cases where each thread is independent of other threads (such as *square_array* in the example above) the grid can simply be augmented to contain more blocks. Grid dimensions are limited to 65535 x 65535 x 1 blocks. For situations where each thread is dependent of other threads such as the computation of a dot product that exceeds 512 in length, A more sophisticated technique is required. The programmer needs to be creative and craft a design that allow threads to be mapped to larger regions, and at the same time not overlap the work of other threads. Taking the *square_array* example, if the problem deals with 1024 elements, each thread can be responsible for data at indices *threadIdx* and *threadIdx* + *blockDim.x*, where $blockDim.x = 512$.

Once a kernel is launched, the corresponding grid and block structure is created. The blocks are then assigned to a SM by the SMC (see CUDA architecture). Each SM executes up to 8 blocks concurrently. Remaining blocks are queued up until a SM is free. The SMCs are smart enough to monitor resource usage and not assign blocks to SMs that are deficient of resources. This ensures that all SMs are functioning to its maximum capacity. As shown in Figure 2.14[4], the more SM a graphics card has, the more concurrent blocks can be executed. Although each block can contain up to 512 threads, and each SM can execute up to a maximum of 8 concurrent blocks, it is not true that at any given time a SM can execute 4096 concurrent threads. Resources are required to maintain the thread and block IDs and its execution state. Due to hardware limitations the SM can

only manage up to 768[4] concurrent threads. However, those threads can be provided to the SM in any configuration of blocks. If a graphics card have 16 SM, then the GPU can be executing up to 12,288 threads concurrently.

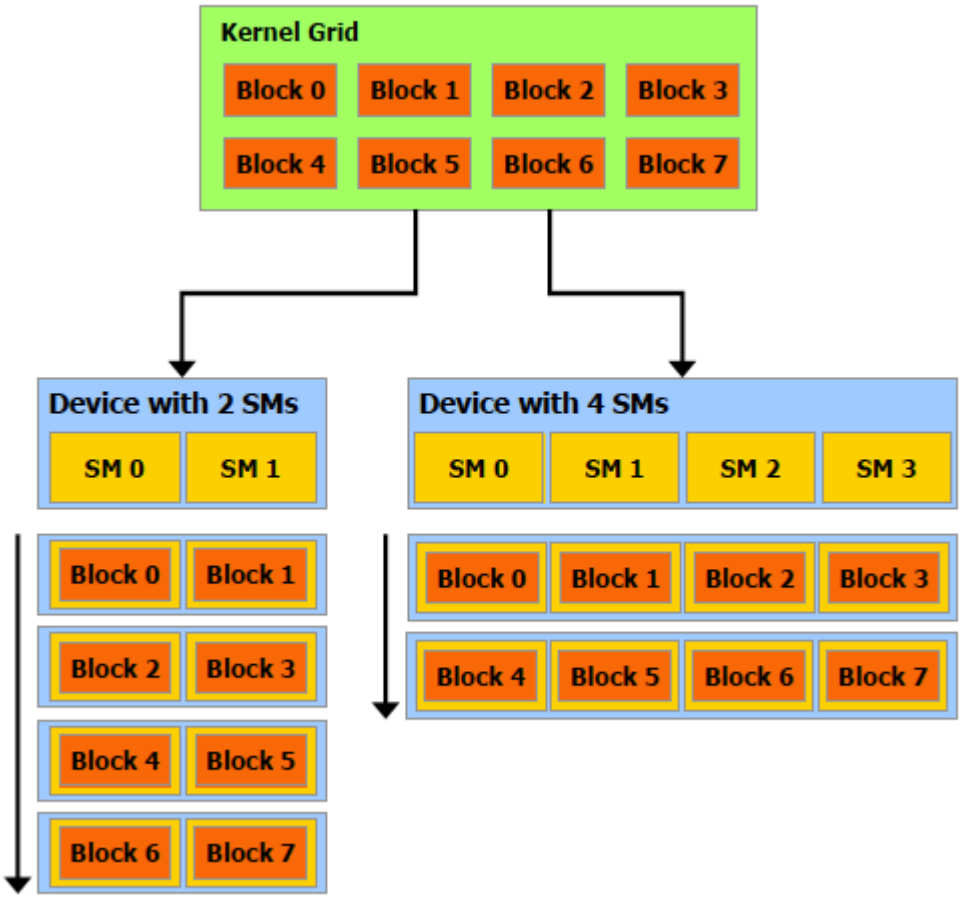


Figure 2.14: A device with more multiprocessors will automatically execute a kernel grid in less time than a device with fewer multiprocessors

To manage and execute hundreds of concurrent threads efficiently, the SM uses a processor architecture known as Single-Instruction, multiple-thread (SIMT). The SIMT instruction unit subdivides threads within a block into groups of 32 parallel thread units call warps. Since a SM can handle up to 768 concurrent threads, it can support up to 24 warps. However, the SM's hardware is designed

to execute only one warp at a time. The reason it is assigned up to 8 warps is to mask long latency operations such as memory access. When an instruction executed by a thread in a warp requires it to wait, the warp is placed in a queue while the SM continues to execute other warps that are available. The SMC employs a priority scheme in assigning warps to the SM. A warp is a construct developed for thread scheduling within the SM. Although warps are not part of the CUDA language specification, it is beneficial to understand what warps are and how it is used. This knowledge provides an edge in optimizing performance of CUDA applications.

All threads of a warp are designed to execute the same block of code in lock steps. When an instruction is issue, the SIMT unit selects a warp that is ready to execute. Full efficiency is achieved when all 32 threads can execute that instruction simultaenously. However, threads are free to branch and execute independently. If a particular thread of the warp diverges from the group based on a conditional branch, the warp will execute each branch serially. While a group of threads are executing a branch, all threads not part of that branch will be disabled. When all threads finish executing their respective branches, the warp will converge back to its original execution path. The SM manages branching threads by using a branch synchronization stack. The branching of threads in a warp is known as thread divergence, and should be avoided since it serializes execution. Divergence only occurs within warps. Different warps are executed independent of each other regardless of the path it takes.

A warp always contains consecutive threads of increasing thread Ids, and is always created the same way. The programmer can take advantage of this fact and use designs that minimizes thread divergence.

SIMT is very similar to the SIMD and SPMD models described earlier. Like SIMD, SIMT allows all threads execute the same instruction. However, similar to SPMD, the SIMT architecture is flexible enough to allow threads to follow different execution paths based on conditional branches. SIMT differs with SPMD in that SIMT refers to the management of threads within a warp where as SPMD focuses on the larger scale of a kernel. The SIMT model greatly increases the set of

algorithms that can be run on this parallel architecture. The SIMT architecture is user friendly in that the programmer can ignore the entire SIMT behavior and the idea of warps. However, substantial performance gain can be achieved if thread divergence is avoided.

## 2.5 CUDA Memory

The typical flow of a CUDA program starts by loading data into host memory and from there transfer to device memory. When an instruction is executed, the threads can retrieve the data needed from device memory. Memory access however can be slow and have limited bandwidth. With thousands of threads making memory calls, this potentially can be a bottle neck and thus, rendering the SMs idled. To ease traffic congestion, CUDA provides several types of memory constructs that improve execution efficiency.

There are 4 major types of device memories: global, constant, shared and register memory (Figure 2.15)[31]. Global memory has the highest access latency among the three. A global variable is declared by using the keyword *__device__*. It is the easiest to use and requires very little strategy. It can easily be read and written to by the host using CUDA APIs and it can be easily accessed by the device. As Listing 2.5 shows, the first step is to allocate global memory by using the *cudaMalloc* function. Then the data in the host is copied to the device by the *cudaMemcpy* function and the constant *cudaMemcpyHostToDevice* indicates that the transfer is from host to device. After the computation is done, the same step is applied to move the data back to the host. Finally the global memory allocated on the device is freed by the *cudaFree()* function. The only constraint on usage of global memory is that it is limited by memory size. Data in global memory lasts for the duration of the entire application and is accessible by any thread across any grid. Global memory is the only way for threads from different blocks to communicate with each other. However, during execution of a single grid, there is no way to synchronize threads from different blocks. Therefore, for practical purposes, global memory is more useful for information from one kernel invocation to be saved and used by a future kernel invocation.

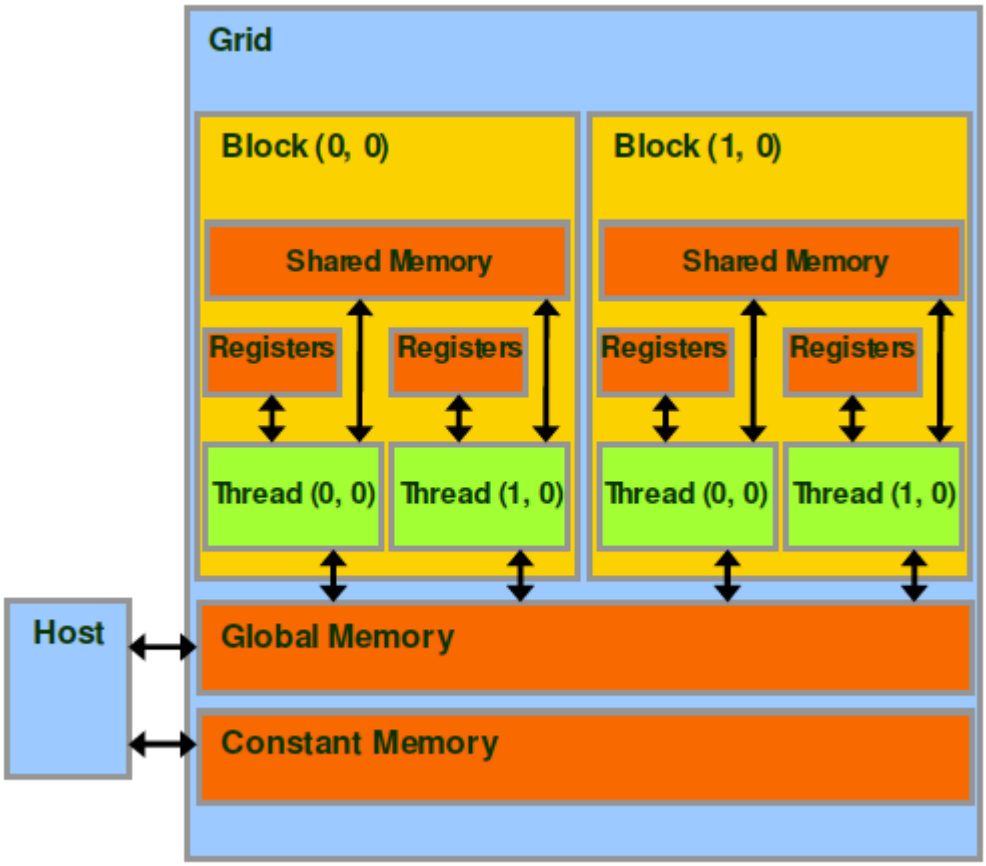


Figure 2.15: Different memory types: Constant, Global, Shared and Register memory

Listing 2.5: Copying data from host memory to device memory and vice versa

```
1 cudaMalloc((void **) &a_d, size);   //Allocate array on device
2 cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
3 ...
4 cudaMemcpy(a_h, a_d, size, cudaMemcpyDeviceToHost);
5 cudaFree(a_d);     //Frees memory on the device.
```

Constant memory is very similar to global memory. In fact, these are the only two memory that the host can read and write to. The main difference from global memory is that constant memory is read-only to the device because it is designed for faster parallel data access. Data is stored in global memory but are cached for efficient access. It allows for high-bandwidth, short-latency access when

all threads simultaneously read from the same location. A constant variable is declared by using the keyword *__constant__*. Like global memory, constant memory also lasts for the entire duration of the application.

Shared memory is an on-chip memory that the host cannot access. This type of memory is allocated on a block level and can only be accessed by threads of that block. Shared memory is the most efficient way for threads of the same block to cooperate, usually by synchronizing read and write phases. It is much faster than using global memory for information sharing within a block. Shared memory is declared by using the keyword *__shared__*. It is typically used inside the kernel. The contents of the memory last for the entire duration of the kernel invocation.

The last type of memory is register memory. Registers are allocated to each individual thread, and are private to each thread. If there are 1 million threads declaring a variable, 1 million versions will be created and stored in their registers. Once the kernel invocation is complete, that memory is released. Variables declared inside a kernel (that are not arrays, and without a keyword) are automatically stored in registers. Variables that are arrays are stored in global memory, but since the variables are declared inside a kernel, the scope is still at the kernel level. Arrays inside a kernel is seldomly needed.

## 2.6 Limitations of CUDA

One of the limitations of the early CUDA architecture is the lack of support for recursion. Mainly a hardware limitation, the the stack and overhead for recursion was too heavy to support. This limitation has been overcome in devices with CUDA compute capability greater than 2.0, which is a new architecture code name *FERMI*.

Another limitation is its compliance with the *IEEE-754* standard for binary floating point arithmetic[4]. For single-precision floating point numbers:

- Addition and Multiplication are combined into a single multiply-add operation (FMAD), which

truncates the intermediate result of the multiplication

- Division is implemented via the riciprocal
- For addition and multiplication, only round-to-nearest-even and round-towards-zero are supported via static rounding modes
- Underflowed results are flushed to zero

For double-precision floating point numbers:

- Round-to-nearest-even is the only supported *IEEE* rounding mode for reciprocal, division and square root.

Finally, CUDA is a proprietary architecture owned by NVidia and is available through NVidia video cards only.

## 2.7 Common CUDA APIs

*Function Qualifiers*

- *__device__*
  - declares a function that is executed on the device, and called by the device
  - do not support recursion
- *__global__*
  - declares a function that is executed on the device, and called by the host
  - must have void as return type
  - function call is asynchronous
  - do not support recursion
- *__host__* - declares a function that is executed on the host, and called by the host

*Variable Type Qualifiers*

- *__device__*
  - declares a variable on the device that resides in global memory
  - has the lifetime of an application
  - is accessible from all threads across all grids
  - can read/write by the host and device
- *__constant__*
  - declares a variable on the device that resides in constant memory
  - has the lifetime of an application
  - is accessible from all threads across all grids
  - can read/write by the host and read only by the device
- *__shared__*
  - declares a variable on the device that resides in shared memory
  - has the lifetime of a block
  - is accessible (read/write) from all threads within the same block

*Built-In Variables*

- *gridDim* - contains the dimension of the grid
- *blockDim* - contains the dimension of the block
- *blockIdx* - contains the index of the block
- *threadIdx* - contains the index of the thread

*Common Runtime Components*

- *dim3* Type - Used to declare a type with dimensions
- *__syncthreads()* - used to synchronize threads within a kernel
- *cudaThreadSynchronize()* - used to synchronize threads between kernels
- *cudaMalloc()* - allocates memory in the device
- *cudaFree()* - frees the allocated memory in the device
- *cudaMemcpy()* - copies memory content between the host and device

For a complete reference of the CUDA API, please visit NVidia's website.

# Chapter 3

# Image Processing and CUDA

Image processing is a type of signals processing in which the input is an image, and the output can be an image or anything else that undergoes some meaningful processing. Converting a colored image to its grayscale representation is an example of image processing. Enhancing a dull and worn off fingerprint image is another example of image processing. More often than not, image processing happens on the entire image, and the same steps are repeatedly applied to every pixel of the image. This programming paradigm is a perfect candidate to fully leverage CUDAs massive compute capabilities.

This section will compare the performance differences between software that are run on a sequential processor (CPU) and a parallel processor (GPU). The experiment will consist of performing various image processing algorithms on a set of images. Image processing is ideal for running on the GPU because each pixel can be directly mapped to a separate thread.

The experiment will involve a series of image convolution algorithms. Convolutions are commonly used in a wide array of engineering and mathematical applications. A simple highlevel explanation is basically taking one matrix (the image) and passing it through another matrix (the convolution matrix). The result is your convoluted image. The matrix can also be called the filter.

## 3.1 Gaussian Blur

Image smoothing is a type of convolution most commonly used to reduce image noise and detail. This is generally done by applying the image through a low pass filter. The filter will retain lower frequency values while reducing high frequency values. The image is smoothed by reducing the disparity between pixels by its nearby pixels.

Image smoothing is sometimes used as a preprocessor for other image operations. Most commonly, an image is smoothed to reduce noise before an edge detection algorithm is applied. Smoothing can be applied to the same image over and over again until the desired effect is achieved.

A simple way to achieve smoothing is by using a mean filter. The idea is to replace each pixel with the average value of all neighboring pixels including itself. One of the advantages of this approach is its simplicity and speed. However, a main disadvantage is that outliers, especially ones that are farthest away from the pixel of interest can create a misrepresentation of the true mean of the neighborhood.

Another way to smooth an image is to use the Gaussian Blur[32]. The Gaussian Blur is a sophisticated image smoothing technique because it reduces the magnitude of high frequencies proportional to their frequencies. It gives less weight to pixels further from the center of the window. The Gaussian function is defined as:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where $\sigma$ is the standard deviation of the distribution. The discrete kernel at (0,0) and $\sigma = 1$ is shown in Figure 3.1[33].

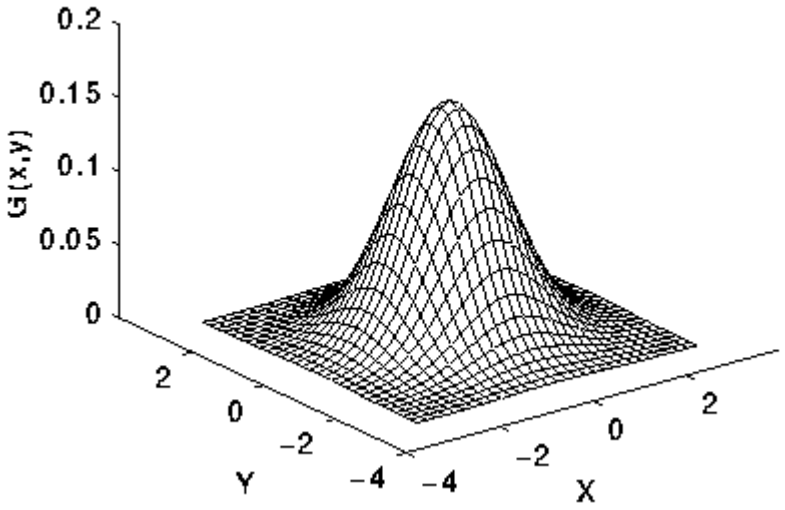


Figure 3.1: Discrete kernel at (0,0) and $\sigma = 1$

## 3.2 Sobel Edge Detection

Edge detection is a common image processing technique used in feature detection and extraction. Applying an edge detection on an image can significantly reduce the amount of data needed to be processed at a later phase while maintaining the important structure of the image. The idea is to remove everything from the image except the pixels that are part of an edge. These edges have special properties, such as corners, lines, curves, etc. A collection of these properties or features can be used to accomplish a bigger picture, such as image recognition.

An edge can be identified by significant local changes of intensity in an image[34]. An edge usually divides two different regions of an image. Most edge detection algorithms work best on an image that has the noise removal procedure already applied. The main ones existing today are techniques using differential operators and high pass filtration.

A simple edge detection algorithm is to apply the Sobel edge detection algorithm. It involves convolving the image using a integer value filter, which is both simple and computationally inex-

pensive.

The Sobel filter is defined as:

$$S_1 = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, S_2 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

To apply the sobel algorithm on an image, we first find the approximate derivatives with respect to the horizontal and vertical directions. Let $A$ be the original image, $G_x$ be the derivative approximation on the horizontal axis and $G_y$ be the derivative approximation on the vertical axis.

$$Gx = S_1 \cdot A$$

$$Gy = S_2 \cdot A$$

The resulting gradient image is the combination of $G_x$ and $G_y$. Each pixel $G(x, y)$ of the resulting image can be calculated by taking the magnitude of $G_x$ and $G_y$:

$$G(x, y) = \sqrt{{G_x}^2 + {G_y}^2}$$

The gradients direction is calculated by:

$$\theta = \arctan \frac{G_y}{G_x}$$

Finally, to determine whether a pixel of the original image $A$ is part of an edge, we apply:

$$\text{if } G(x, y) > \textit{threshold}, \text{ then } A(x, y) \text{ is part of an edge}$$

## 3.3 Gaussian Blur Implementation

To compare the speedup differences between processing on the CPU vs processing on the GPU, an experiment was done using the above algorithms in both the sequential and the parallel model. Both implementations are shown in the source code (Listing 3.1).

The programs are run on an Intel Core 2 Duo, 2GHz processor with a NVidia GeForce GTX 260. The graphics card contains 192 cores at 1.2 GHz each. Each algorithm is run against images that are 266kb, 791kb, and 7.7mb in size. The images had dimensions of 512 x 512, 1024 x 768, 3200 x 2400 respectively.

### 3.3.1 Implementation

Listing 3.1: Sequential and Parallel Implementation of the Gaussian Blur

```
1 #include <time.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <string.h>
5 #include <math.h>
6 #include <cuda.h>
7 #include <cutil.h>
8 #include <ctime>
9
10 unsigned int width, height;
11 int mask[3][3] = {1,2,1,
12                   2,3,2,
13                   1,2,1,
14                   };
15
16 int getPixel(unsigned char * arr, int col, int row){
17
18     int sum = 0;
19
20     for (int j=-1; j<=1; j++){
21        for (int i=-1; i<=1; i++){
22           int color = arr[(row + j) * width + (col + i)];
23           sum += color * mask[i+1][j+1];
24        }
25     }
26
27     return sum/15;
28 }
29
```

```
30 void h_blur(unsigned char * arr, unsigned char * result){
31     int offset = 2 * width;
32     for (int row=2; row< height-3; row++){
33        for (int col=2; col<width-3; col++){
34           result[offset + col] = getPixel(arr, col, row);
35        }
36        offset += width;
37     }
38 }
39
40
41 __global__ void d_blur(unsigned char * arr, unsigned char * result,
       int width, int height){
42     int col = blockIdx.x * blockDim.x + threadIdx.x;
43     int row = blockIdx.y * blockDim.y + threadIdx.y;
44
45     if (row < 2 || col < 2 || row >= height -3 || col >= width -3 )
46        return;
47
48     int mask[3][3] = {1,2,1, 2,3,2, 1,2,1};
49
50     int sum = 0;
51     for (int j=-1; j<=1; j++){
52        for (int i=-1; i<=1; i++){
53           int color = arr[(row + j) * width + (col + i)];
54           sum += color * mask[i+1][j+1];
55        }
56     }
57
58     result[row * width + col] = sum/15;
59
60 }
61
62
63 int main( int argc, char** argv)
64 {
65     /********************* setup work ***************************
          */
66     unsigned char * d_resultPixels;
```

```
67      unsigned char * h_resultPixels;
68      unsigned char * h_pixels = NULL;
69      unsigned char * d_pixels = NULL;
70
71
72      char * srcPath = "/Developer/GPU Computing/C/src/GaussianBlur/
            image/wallpaper2.pgm";
73      char * h_ResultPath = "/Developer/GPU Computing/C/src/
            GaussianBlur/output/h_wallpaper2.pgm";
74      char * d_ResultPath = "/Developer/GPU Computing/C/src/
            GaussianBlur/output/d_wallpaper2.pgm";
75
76
77      cutLoadPGMub(srcPath, &h_pixels, &width, &height);
78
79      int ImageSize = sizeof(unsigned char) * width * height;
80
81      h_resultPixels = (unsigned char *)malloc(ImageSize);
82      cudaMalloc((void**)&d_pixels, ImageSize);
83      cudaMalloc((void**)&d_resultPixels, ImageSize);
84      cudaMemcpy(d_pixels, h_pixels, ImageSize, cudaMemcpyHostToDevice
            );
85
86      /********************* END setup work
            ***************************  */
87
88      /************************* Host processing
            *************************/
89
90      clock_t starttime, endtime, difference;
91      starttime = clock();
92
93      //apply gaussian blur
94      h_blur(h_pixels, h_resultPixels);
95
96      endtime = clock();
97      difference = (endtime - starttime);
98      double interval = difference / (double)CLOCKS_PER_SEC;
99      printf("CPU execution time = %f ms\n", interval * 1000);
```

```
100     cutSavePGMub(h_ResultPath, h_resultPixels, width, height);
101
102     /************************* END Host processing
            *************************/
103
104
105     /************************* Device processing
            *************************/
106     dim3 block(16,16);
107     dim3 grid (width/16, height/16);
108     unsigned int timer = 0;
109     cutCreateTimer(&timer);
110     cutStartTimer(timer);
111
112     /* CUDA method */
113     d_blur <<< grid, block >>>(d_pixels, d_resultPixels, width,
            height);
114
115     cudaThreadSynchronize();
116     cutStopTimer(timer);
117     printf("CUDA execution time = %f ms\n",cutGetTimerValue(timer));
118
119     cudaMemcpy(h_resultPixels, d_resultPixels, ImageSize,
            cudaMemcpyDeviceToHost);
120     cutSavePGMub(d_ResultPath, h_resultPixels, width, height);
121
122     /************************* END Device processing
            *************************/
123
124     printf("Press enter to exit ...\n");
125     getchar();
126 }
```

### 3.3.2 Breaking Down CUDA

Listing 3.2: This calls a CUDA library to allocate memory on the device to *d_pixels*

```
cudaMalloc((void**)&d_pixels, ImageSize);
```

Listing 3.3: Copies the contents of the host memory to the device memory referenced by *d_pixels*

```
cudaMemcpy(d_pixels, h_pixels, ImageSize,cudaMemcpyHostToDevice);
```

Listing 3.4: CUDA calls to create/start/stop the timer

```
cutCreateTimer(&timer);
cutStartTimer(timer);
cutStopTimer(timer);
```

Listing 3.5: Declares block sizes of 16 x 16 for 256 threads per block.

```
dim3 block(16,16);
```

Listing 3.6: This tells us that we want to have a w/16 x h/16 size grid.

```
dim3 grid (width/16, height/16);
```

If the image we are dealing with is 256 x 256, then the grid will be 16 x 16 and will contain 256 blocks. Since each block contains 256 threads, this will amount to 65536, which is exactly the number of pixels in a 256 x 256 image.

Listing 3.7: Invokes the device method *d_blur* passing in the parameters.

```
d_blur <<< grid, block >>>(d_pixels, d_resultPixels, width,
   height);
```

Listing 3.8: Finding the current pixel location.

```
int col = blockIdx.x * blockDim.x + threadIdx.x;
int row = blockIdx.y * blockDim.y + threadIdx.y;
```

These two lines basically determine which thread process on which pixel of the image. As calculated above, there are 65536 threads performing on 65536 pixels. Each thread should perform on its own unique pixel and avoid processing the pixels owned by other threads. Since each thread is uniquely identified by its own thread id, block id and we know the dimensions of the block, we can use the techique above to assign a unique pixel coordinate for each thread to work on.

Listing 3.9: This forces the threads to synchronize before executing further instructions.

```
cudaThreadSynchronize();
```

Listing 3.10: This saves the image to a PGM file.

```
cutSavePGMub(d_ResultPath, h_resultPixels, width, height);
```

## 3.4 Sobel Edge Detection Implementation

The Sobel edge detection algorithm is also implemented in both the sequential and parallel version. It is run on the same hardware and uses the same images as the one used by the Gasssian Blur experiment.

### 3.4.1 Implementation

Listing 3.11: Sequential and Parallel Implementation of the Sobel Edge Detection

```
1
2 #include <time.h>
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <string.h>
6 #include <math.h>
7 #include <cuda.h>
8 #include <cutil.h>
9 #include <ctime>
```

```
10
11 unsigned int width, height;
12
13 int Gx[3][3] = {-1, 0, 1,
14               -2, 0, 2,
15               -1, 0, 1};
16
17 int Gy[3][3] = {1,2,1,
18               0,0,0,
19               -1,-2,-1};
20
21 int getPixel(unsigned char * org, int col, int row){
22
23     int sumX, sumY;
24     sumX = sumY = 0;
25
26     for (int i=-1; i<= 1; i++){
27        for (int j=-1; j<=1; j++){
28           int curPixel = org[(row + j) * width + (col + i)];
29           sumX += curPixel * Gx[i+1][j+1];
30           sumY += curPixel * Gy[i+1][j+1];
31        }
32     }
33     int sum = abs(sumY) + abs(sumX);
34     if (sum > 255)    sum = 255;
35     if (sum < 0) sum = 0;
36     return sum;
37 }
38
39 void h_EdgeDetect(unsigned char * org, unsigned char * result){
40     int offset = 1 * width;
41     for (int row=1; row< height-2; row++){
42        for (int col=1; col<width-2; col++){
43           result[offset + col] = getPixel(org, col, row);
44        }
45        offset += width;
46     }
47 }
48
```

```
49 __global__ void d_EdgeDetect(unsigned char *org, unsigned char *
     result, int width, int height){
50     int col = blockIdx.x * blockDim.x + threadIdx.x;
51     int row = blockIdx.y * blockDim.y + threadIdx.y;
52 
53     if (row < 2 || col < 2 || row >= height -3 || col >= width -3 )
54        return;
55 
56     int Gx[3][3] = {-1, 0, 1,
57                     -2, 0, 2,
58                     -1, 0, 1};
59 
60     int Gy[3][3] = {1,2,1,
61                     0,0,0,
62                     -1,-2,-1};
63 
64     int sumX, sumY;
65     sumX = sumY = 0;
66 
67     for (int i=-1; i<= 1; i++){
68        for (int j=-1; j<=1; j++){
69           int curPixel = org[(row + j) * width + (col + i)];
70           sumX += curPixel * Gx[i+1][j+1];
71           sumY += curPixel * Gy[i+1][j+1];
72        }
73     }
74 
75     int sum = abs(sumY) + abs(sumX);
76     if (sum > 255)    sum = 255;
77     if (sum < 0) sum = 0;
78 
79     result[row * width + col] = sum;
80 
81 }
82 
83 int main( int argc, char** argv)
84 {
85     printf("Starting program\n");
86 
```

```
87      /********************* setup work ***************************
            */
88
89      unsigned char * d_resultPixels;
90      unsigned char * h_resultPixels;
91      unsigned char * h_pixels = NULL;
92      unsigned char * d_pixels = NULL;
93
94      char * srcPath = "/Developer/GPU Computing/C/src/EdgeDetection/
            image/cartoon.pgm";
95      char * h_ResultPath = "/Developer/GPU Computing/C/src/
            EdgeDetection/output/h_cartoon.pgm";
96      char * d_ResultPath = "/Developer/GPU Computing/C/src/
            EdgeDetection/output/d_cartoon.pgm";
97
98      cutLoadPGMub(srcPath, &h_pixels, &width, &height);
99
100     int ImageSize = sizeof(unsigned char) * width * height;
101
102     h_resultPixels = (unsigned char *)malloc(ImageSize);
103     cudaMalloc((void**)&d_pixels, ImageSize);
104     cudaMalloc((void**)&d_resultPixels, ImageSize);
105     cudaMemcpy(d_pixels, h_pixels, ImageSize, cudaMemcpyHostToDevice
            );
106
107     /********************* END setup work
            ***************************  */
108
109     /************************ Host processing
            *************************/
110     clock_t starttime, endtime, difference;
111
112     printf("Starting host processing\n");
113     starttime = clock();
114     h_EdgeDetect(h_pixels, h_resultPixels);
115     endtime = clock();
116     printf("Completed host processing\n");
117
118     difference = (endtime - starttime);
```

```
119     double interval = difference / (double)CLOCKS_PER_SEC;
120     printf("CPU execution time = %f ms\n", interval * 1000);
121     cutSavePGMub(h_ResultPath, h_resultPixels, width, height);
122     /************************* END Host processing
            **************************/
123
124     /************************* Device processing
            **************************/
125     dim3 block(16,16);
126     dim3 grid (width/16, height/16);
127     unsigned int timer = 0;
128     cutCreateTimer(&timer);
129
130     printf("Invoking Kernel\n");
131     cutStartTimer(timer);
132     /* CUDA method */
133     d_EdgeDetect <<< grid, block >>>(d_pixels, d_resultPixels, width
            , height);
134     cudaThreadSynchronize();
135     cutStopTimer(timer);
136     printf("Completed Kernel\n");
137
138     printf("CUDA execution time = %f ms\n", cutGetTimerValue(timer))
            ;
139
140     cudaMemcpy(h_resultPixels, d_resultPixels, ImageSize,
            cudaMemcpyDeviceToHost);
141     cutSavePGMub(d_ResultPath, h_resultPixels, width, height);
142
143     /************************* END Device processing
            **************************/
144
145
146
147     printf("Press enter to exit ...\n");
148     getchar();
149 }
```

# Chapter 4

# Results

The results of both executions are shown in Table 4.1 & 4.2. As shown in the results, the GPU time has a significant increase over the CPU time in all the images that were processed. Irregardless of the type of algorithm ran, the results are affirmative. Processing on the GPU has a huge edge over processing on the CPU. The rate of percent increase increases as the image size increases. This aligns with the earlier claim that CUDA processing is most effective when lots of threads are being utilized simultaneously.

| | GPU Time(ms) | CPU Time(ms) | Percent Increase |
|---|---|---|---|
| 512 x 512 Lena | 0.67 | 16 | 2,288 |
| 1024 x 768 wallpaper2 | 0.84 | 62 | 7,280 |
| 3200 x 2400 cartoon | 2.92 | 688 | 23,461 |

Table 4.1: Results of the Gaussian Blur

| | GPU Time(ms) | CPU Time(ms) | Percent Increase |
|---|---|---|---|
| 512 x 512 Lena | 0.67 | 32 | 4,676 |
| 1024 x 768 wallpaper2 | 0.82 | 94 | 11,363 |
| 3200 x 2400 cartoon | 2.87 | 937 | 32,548 |

Table 4.2: Results of the Sobel Edge Detection

The results also show that the edge detection algorithm in general is slightly less computationally expensive than the Gaussian Blur. While that difference is shown with the need for more time in the sequential algorithm, the parallel algorithm is unaffected. This further confirms that the more computation power is required, the more CUDA is utilized to its full potential.

# Chapter 5

# Conclusion and Future Work

Graphics cards have widely been used to accelerate gaming and 3D graphical applications. High level programmable interfaces now allow this technology to be used for general purpose computing. CUDA is the first of its kind from the NVidia tech chain. It is fundamentally sound and easy to use. This thesis gives an introduction of the type of performance gains that can be achieved by switching over to the parallel programming model.

Image processing algorithms is a category of algorithms that work well in achieving the best benefits out of CUDA. Most algorithms are such that a type of calculation is repeated over and over again in massive amounts. This is perfect for utilizing CUDA's massive amounts of threads. Most of these algorithms can be processed independently of each other, making it ideal to spawn off threads to perform these calculations simultaneously.

In chapter 2, we give an overview of what GPGPU is, and goes into depths of the benefits of using CUDA. The chapter discusses CUDA's architecture, including its memory model, its thread hierarchy, and programming model. We showed the type of algorithms that benefit the most out of CUDA, and how to program in order to reap the maximum of CUDA's benefits.

In Chapter 3, we present examples to the reader of what a typical CUDA program looks like from beginning to end. It has a complete breakdown of what each method call does. The experiment is

done using two well know image processing algorithms: Gaussian Blur and Sobel Edge Detection. The implementation contains both the sequential version and the parallel version. This allows the reader to compare and contrast the performance differences between the two executions.

Chapter 3 gives the reader an idea of the type of algorithms that are well fitted for CUDA. It is an example of how a sequential algorithm can be craftily broken down such that it can be run in parallel and achieve the same results, but faster. Creative techniques like these are required to be made when programming in the parallel model.

Chapter 4 shows the results of the experiment. It provide several executions of the same algorithm against different images. It affirms the claim that the larger the data set, the better the benefits are in using CUDA. For one of the smaller test cases, the performance increase is only 22%. The gain becomes 234% when we process an image 29 times bigger.

This thesis gives an introduction to CUDA and its benefits, but it does not stop here. A lot of future work can be done. Experiments can be done by using different size grids and blocks. The experiements are likely to improve with smarter memory usages. A lot can still be explored beyond this thesis.

CUDA, though it is ready for commercial use, is still a very young product. *FERMI* is the next generation currently available that is better than CUDA. CUDA blocks can hold up to 512 threads while FERMI blocks can hold up to 1536 threads. Another advantage is that FERMI supports the execution of multiple kernels simultaneously. CUDA must execute kernels sequentially. As technology advances, there are sure to be products that are better and better.

# Appendix A: Glossary

**Block** - A name for a container that represents a group of threads. Threads belong in a block, which then belongs in a grid. Blocks can be partitioned into several dimensions to make indexing the threads easier. Threads within the same block can communicate with each other.

**Central Processing Unit (CPU)** - A serial processor on a computer that is optimized for high performance on sequential operations.

**Compute Unified Device Architecture (CUDA)** - A parallel computing architecture developed by NVidia for massively parallel high-performance computing.

**Constant Memory** - Similar to global memory, except this is read-only for the device. It is optimized for faster parallel data access.

**CUDA C Compiler (CUDACC)** - This compiles the GPU file produced by the NVCC and creates CUDA object files.

**Device** - In the context of a CUDA program, the device is everythign that is in the graphics card. This includes the GPU, the memory that is in the graphics card, etc.

**FERMI** - The next generation CUDA architecture that is faster and more powerful than CUDA

**General Purpose GPU (GPGPU)** - A type of computing that utilizes the computational power of the GPU in computing that are not necessarily graphics related. For example, using the GPU to solve a matrix.

**Global Memory** - Variables declared in the global memory space lasts for the entire duration of the application and can be accessed by anythread across any grid. Both the host and the device can read and write to this.

**Graphics Processing Unit (GPU)** - A stream processor on a graphics card specialized for compute-intensive, highly parallel computation.

**Grid** - A name for a container that represents all the threads of a single kernel execution. A grid contains a set of blocks, which contains a set of threads.

**Host** - In the context of a CUDA program, the host is everything that is not on the graphics card. This can be the CPU, the memory that is on the computer, etc.

**Kernel** - A function or method that is executed on the device.

**NVidia C Compiler (NVCC)** - A compiler that parses the source code (.cu) and creates two resulting files: One for processing on the GPU and one for processing on the CPU.

**Parallel Thread eXecution (TPX)** - A type of file that is produced by the CUDACC. These files are recognized by device drivers that are installed with NVidia graphics cards.

**Register Memory** - This type of memory is allocated on the thread level, and are private to each individual thread.

**Shared Memory** - This type of memory is on the device, and the host has no access to. It is allocated on the block level and can only be accessed by threads of that block.

**Single Instruction Multiple Data (SIMD)** - A type of programming paradigm in which a set of threads execute the same instructions but against a different dataset. The set of threads execute the same instructions in locksteps.

**Single Instruction Multiple Thread (SIMT)** - A type of architecture that is used for the management of threads. When an instruction is issued, a SIMT unit selects a group of threads

that can execute that instruction.

**Single Program Multiple Data (SPMD)** - The same as SIMD except the threads do not have to execute the same instructions in locksteps. Threads are allowed to branch in the program and execute a different set of instructions.

**Special Function Units (SFU)** - The units in a SM that specializes in floating point functions such as square root and transcendental functions.

**Streaming Multiprocessor (SM)** - This contains a group of SPs, and 2 SFUs, shared memory, and cache.

**Streaming Processor (SP)** - This is where the actual computation happens. It contains its own MAD and MUL units.

**Streaming Processor Array (SPA)** - This refers to a group of streaming processors inside the GPU. This is where all the computation takes place.

**Texture/Processor Clusters (TPC)** - This is a member of the SPA. Each TPC contains a geometry controller, a SM Controller, a texture unit and 2 SMs.

**Warp** - A construct developed for thread scheduling within the SM. A warp contains a group of threads. Thread executions are usually done in a warp group.

# Bibliography

[1] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, (New York, NY, USA), pp. 215–226, ACM, 2004.

[2] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra, "Physically-based visual simulation on graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, (Aire-la-Ville, Switzerland, Switzerland), pp. 109–118, Eurographics Association, 2002.

[3] M. C. Lin and D. Manocha, "Interactive geometric and scientific computations using graphics hardware," in *SIGGRAPH 2003 Course Notes, vol. 11. ACM SIGGRAPH*, ACM SIGGRAPH, 2003.

[4] NVidia, "Nvidia cuda c programming guide." `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`, 2012.

[5] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, (New York, NY, USA), pp. 703–712, ACM, 2002.

[6] N. A. Carr, J. D. Hall, and J. C. Hart, "The ray engine," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, (Aire-la-Ville, Switzerland, Switzerland), pp. 37–46, Eurographics Association, 2002.

[7] J. Zhou and K. A. Ross, "Implementing database operations using simd instructions," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, (New York, NY, USA), pp. 145–156, ACM, 2002.

[8] W. Fang, K. K. Lau, M. Lu, X. Xiao, C. K. Lam, P. Y. Yang, B. He, Q. Luo, P. V. S, and K. Yang, "Parallel data mining on graphics processors," tech. rep., 2008.

[9] N. Bandi, C. Sun, D. Agrawal, and A. El Abbadi, "Hardware acceleration in commercial databases: a case study of spatial operations," in *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pp. 1021–1032, VLDB Endowment, 2004.

[10] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, (New York, NY, USA), pp. 46:1–46:12, ACM, 2007.

[11] C. Sun, D. Agrawal, and A. El Abbadi, "Hardware acceleration for spatial selections and joins," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, (New York, NY, USA), pp. 455–466, ACM, 2003.

[12] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver, "Fast computation of generalized voronoi diagrams using graphics hardware," in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, (New York, NY, USA), pp. 277–286, ACM Press/Addison-Wesley Publishing Co., 1999.

[13] S. Krishnan, N. H. Mustafa, and S. Venkatasubramanian, "Hardware-assisted computation of depth contours," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete*

*algorithms*, SODA '02, (Philadelphia, PA, USA), pp. 558–567, Society for Industrial and Applied Mathematics, 2002.

[14] P. K. Agarwal, S. Krishnan, N. H. Mustafa, and Suresh, "Streaming geometric optimization using graphics hardware," in *In Proc. 11th European Sympos. Algorithms, Lect. Notes Comput. Sci*, pp. 544–555, Springer-Verlag, 2003.

[15] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms," in *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, (New York, NY, USA), pp. 908–916, ACM, 2003.

[16] M. J. Harris, W. V. Baxter, T. Scheuermann, and A. Lastra, "Simulation of cloud dynamics on graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, (Aire-la-Ville, Switzerland, Switzerland), pp. 92–101, Eurographics Association, 2003.

[17] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys, "A multigrid solver for boundary value problems using programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, (Aire-la-Ville, Switzerland, Switzerland), pp. 102–111, Eurographics Association, 2003.

[18] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," in *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, (New York, NY, USA), pp. 917–924, ACM, 2003.

[19] I. Buck, "Brook specification v0.2." `http://merrimac.stanford.edu/brook/`, Oct. 2003.

[20] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, (New York, NY, USA), pp. 325–335, ACM, 2006.

[21] M. D. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin, "Performance evaluation of gpus using the rapidmind development platform," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, (New York, NY, USA), ACM, 2006.

[22] Peakstream, "The peakstream platform: High productivity software development for multi-core processors," tech. rep., 2006.

[23] A. Munshi, "Opencl: Parallel computing on the gpu and cpu." presentation at SIGGRAPH, 2008.

[24] B. Endre, "Nvidia gtx-275." `http://www.bjorn3d.com/2009/04/nvidia-gtx-275/`, 2009.

[25] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, Mar. 2008.

[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, Mar. 2008.

[27] T. Halfhill, "Parallel processing with cuda." Nvidia's High-Performance Computing Platform Uses Massive Multithreading, 2008.

[28] M. Harris, "Tesla gpu computing." `http://www.cse.unsw.edu.au/~pls/cuda-workshop09/`, 2009.

[29] NVidia, "Isc 2009 cuda tutorial." `http://gpgpu.org/isc2009`, 2009.

[30] M. J. Atallah in *Algorithms and theory of computation handbook*, (Boca Raton, FL), CRC Press, 1998.

[31] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2010.

[32] M. S. Nixon and A. S. Aguado, "Feature extraction and image processing." Academic Press, 2008.

[33] R. Fisher, S. Perkins, A. Walker, and E. Wolfart, "Gaussian smoothing." `http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm`, 2003.

[34] W. Gao, X. Zhang, L. Yang, and H. Liu, "An improved Sobel edge detection," in *International Conference on Computer Science and Information Technology*, 2010.

# Vita

Graduate College

University of Nevada, Las Vegas

Jia Tse

Degrees:

Master of Science in Computer Science 2012

University of Nevada Las Vegas

Thesis Title: Image Processing with CUDA

Thesis Examination Committee:

Chairperson, Dr. Ajoy K. Datta, Ph.D.

Committee Member, Dr. Lawrence L. Larmore, Ph.D.

Committee Member, Dr. Yoohwan Kim, Ph.D.

Graduate Faculty Representative, Dr. Venkatesan Muthukumar, Ph.D.