Computed Unified Device Architecture(CUDA) Tutorial

Author: Trevor Cickovski
Written For: Computer Science Department
Eckerd College
USA
Questions: *tcickovs@fiu.edu*

Note: For the most updated CUDA documentation,
please use NVIDIA's CUDA Programmer's Guide and other documentation,
available here: `http://docs.nvidia.com/cuda/`

August 3, 2016

Abstract

I have written this tutorial to provide general guidance for teaching and using the facilities of CUDA in the most effective and productive way. The tutorial is designed for Professors and Instructors at Eckerd College, and thus will reference Eckerd courses and available computing facilities at the time of its release.

Available for any system with an NVIDIA graphics card, CUDA is a programming language that extends C to make available the high performance computing facilities (multithreading, manycores) available on modern graphics cards. This tutorial assumes a basic knowledge of the C programming language and will focus on the additional knowledge required to efficiently utilize tools unique to the CUDA language.

CUDA is appropriate for many computer science courses. At the time of this release, it has been used in CS310 (Computer Architecture) in its parallelism unit where we studied Single Instruction Multiple Data (SIMD) architectures. From a higher level perspective, it was also used in CS320 (Programming Languages). It also has potential applications in computer graphics courses, as well as any other special topic for which performance is critical (i.e. bioinformatics).

Acknowledgements

Eckerd College is grateful for support from NVIDIA through their CUDA Teaching Center Program in enabling the computer science department to include CUDA in their curriculum.

Contents

Chapter 1

Introduction

Throughout the history of computing, the graphics card was used primarily for one fundamental role: taking RGB image data and displaying it efficiently. For any display with reasonable resolution, this would be an impossible task without some form of parallelism. Every keystroke, mouse movement, or event trigger affects the display in some fashion. Modern multimedia applications exhibit high-resolution graphics (for high definition, 2.1 megapixel is common at the time of this release [3]).

Thus while general computing tasks may run efficiently on a multicore (i.e. 16-core) CPU, graphics processing has long run on a separate, manycore and multithreaded architecture, the Graphics Processing Unit (GPU). Since pixel data is in RGB (integer, 3-4 bytes per pixel) format, GPU hardware has been traditionally optimized for integer processing, using small memories. For this reason, for many years general-purpose computing and graphics processing were mutually exclusive, running on their unique individual designs that were adequate for their approaches. Figure 1.1 illustrates the general GPU hardware structure. Each core consists of a *block* of threads. Individual threads contain their own registers and local memories, and all threads within the same block share an on-chip cache, known as *shared memory*. Global (read-write), constant (read-only) and texture memory (reserved for special operations) are shared by all threads. None of these had to be exceptionally large. Note that any image data that the GPU processes needs to be in one of these memories. Most often this was done by a single expensive copy from the CPU, as shown in the Figure.

Interest in the GPU for general purpose computing (a term known as GPGPU computing [4]) arose fairly recently and was a result of many changes in the computing field. Data processing applications became far more intensive [14, 8, 6]. In addition, many called into question the future long-term reliability of Moore's law once transistor size dropped below the size of the atom [2], projected by 2022. While many alternative techniques such as quantum computing have been proposed, until they become reliable and household the best alternative will be parallelism. The hardware for graphics cards provided the best opportunity for this.

Initially released in 2007, CUDA provides access to the GPU parallel framework through a C programming interface. Their compiler translates CUDA source code into an assembly language PTX [10], which will in turn execute on NVIDIA graphics hardware. Other languages such as OpenCL [13] and Brook [15] have been developed for similar purposes. One advantage to OpenCL in particular is that it can run on any graphics card, however CUDA will provide the best option for maximizing efficiency on an NVIDIA graphics card. In addition, CUDA builds upon C in a minimalist fashion, only requiring mastery of a few new features to understand the language.

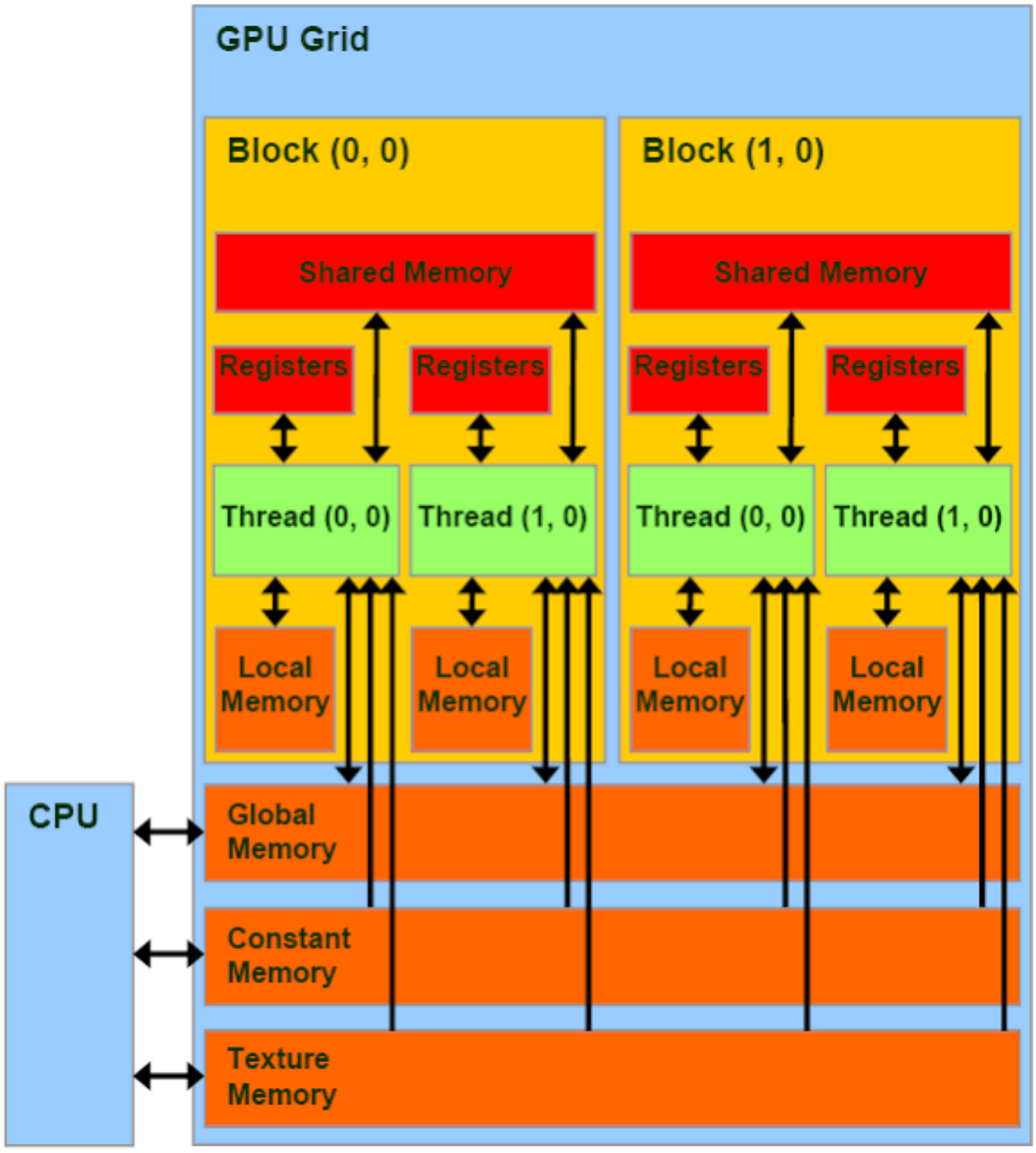

Figure 1.1: Sample GPU configuration, showing individual blocks (cores) and internal threads. Each thread has its own local memory and registers. A single shared memory unit is shared by all threads in the same block. All threads have access to a single global, texture and constant memory. From [9].

Chapter 2

Using CUDA

2.1 Getting Started

2.1.1 Installation

CUDA is available on NVIDIA's website. As of this release, the most up-to-date version is CUDA 7.5, available here:
`https://developer.nvidia.com/cuda-downloads`

However, one should check back and stay up-to-date with the newest version of the software, as releases with new useful features and compatibilities are quite common (ranging from every three to ten months [11]). CUDA installation is also simple on any Linux system with `apt-get`, by running: `sudo apt-get install cuda`.

Several Linux systems at Eckerd already have CUDA installed and some can be accessed remotely. The computational server, `vanhalen.eckerd.edu`, contains an NVIDIA GTX780 graphics card. This server resides in the computational research lab MPC113, as do two computational workstations `styx.eckerd.edu` and `journey.eckerd.edu`, also with GTX780s. The latter can be accessed locally. Finally there are two workstations accessible remotely, `starship.eckerd.edu` and `dantzig.eckerd.edu`, each with GTX680 cards.

2.1.2 Compilation

CUDA assumes a default extension of `.cu` for its source files. Installing CUDA will place the NVIDIA compiler `nvcc` in an accessible directory on your computer. Assuming that directory is in your system `PATH`, the command to compile any `.cu` file is:

```
nvcc file.cu
```

Like C, this will produce executable code in a file `a.out`. Also the same as C, if you would like a different executable name you can use the `-o` flag:

```
nvcc file.cu -o executable
```

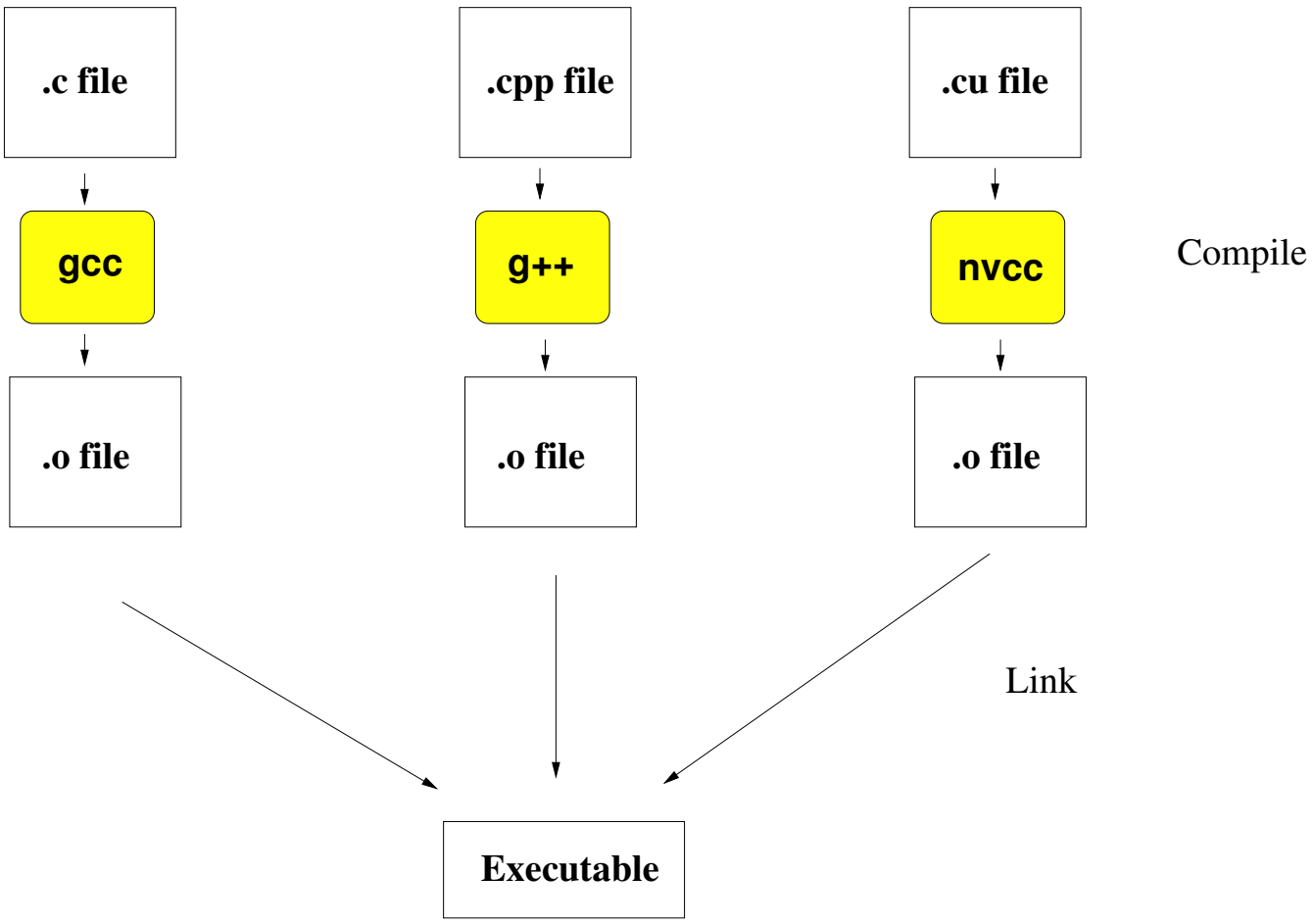

Figure 2.1: Process of compiling and linking CUDA files with other C and C++ libraries.

This alone will be enough to compile and run a single CUDA program. In the following subsections I outline a few features that you may desire, that I have encountered while teaching CUDA courses and that may prove useful.

Multiple CUDA files

As with C, `.cu` files can be combined into separate object files and later linked into a single executable as shown in Figure 2.1, using the `-c` flag. Since CUDA extends C and uses the same C runtime environment to invoke GPU functionality, these same object files can be linked with other object files from C and C++ source code. The command would be the same as C:

```
nvcc -c file.cu
```

This would automatically create `file.o`. The exact name of the object file can be changed using the `-o` flag, as with C:

```
nvcc -c file.cu -o objectfile.o
```

Dynamically Loaded Libraries

This is also a useful feature of NVIDIA which works similar to C. Producing object code that is loaded and linked dynamically requires the flags `-fpic` and `-shared`, in two discrete steps. Using `-fpic` when producing the object code will ensure that the code is position-independent, enabling its assembly to be placed anywhere with respect to the other instructions:

```
nvcc -c -fpic file.cu
```

As before the above produces `file.o`, which can be converted into a shared object file using the `-shared` flag:

```
nvcc -shared file.o -o libfile.so
```

Often dynamically loaded libraries use `lib` as a prefix, though that is not required.

Other Useful Flags

In this section I outline other flags that I sometimes have found useful.

- `-I`: Include the directory that follows in the search path for `#include`d header files (same as C).
- `-l`: Statically link the library that follows into the executable (same as C).
- `-L`: Include the directory that follows in the search path for statically linked libraries with `-l` (same as C).
- `-arch`: Each NVIDIA graphics card has a certain compute capability (Wikipedia contains a useful chart here: `https://en.wikipedia.org/wiki/CUDA`). Through my experience with CUDA, I have found that `nvcc` does not always assume the compute capability of the locally installed graphics card when compiling, and often assumes a lower one. This holds consequences as shown by the chart, since a lower compute capability implies a lower amount of multithreading. The `-arch` flag allows you to specify the compute capability - for example for compute capability 3.0 you could say `-arch=sm_30`.
- `-ptxas-options=-v`: Specifies the amount of different types of memory (registers, cache, and RAM) used by each function, this could be useful particularly in an architecture class.
- `-std=c++11`: Can be useful if you are compiling a CUDA file with C++ code (recently supported) and require C++11.

2.2 Writing CUDA Code

I now illustrate how to construct a CUDA source file, using a simple example from CS310 (Computer Architecture) at Eckerd during the fall of 2014. This example takes a vector of any size (within memory boundaries of course), initializes it with random integers between 0 and 9, and squares each element. The entire source code of this program is shown in Program 1, and the file was named `vecsquare.cu`. While a very basic example, it is good for learning the unique features of CUDA that are not available in C. To save space I am not printing array elements before or after, though that could be added very easily using regular C.

Which brings us to the first point: much of the code is purely just C, run on the CPU! In fact that will very often be the case with CUDA code. The ninety-ninety rule [1] states that 90% of execution time is spent running 10% of the code. CUDA capitalizes on this. In fact the only GPU code in this example is the three lines within the `vecsquare` function. Thus the rest of the program is structured as a normal C program, commencing execution from the top of `main()`. In a CUDA file the CPU thus acts as a "manager", which calls GPU functionality at the appropriate times.

2.2.1 Control Flow

To understand the functionality of the program, it is important to first see visually how squaring a vector will work between the CPU and GPU. This is illustrated by Figure 2.2 The first important thing to note is that the CPU and GPU *each have separate memories*. The CPU cannot operate on memory allocated on the GPU, and vice-versa. Each will trigger either a segmentation fault or unpredictable results.

The vector of elements `A` gets first allocated and initialized with random values in CPU memory. Next, a large enough chunk `gpu_A` must be allocated in GPU memory, to hold the values of `A`. Once this chunk is available, the elements of `A` must be copied into the currently empty space `gpu_A`. GPU threads can then each take one element of `gpu_A` and square it, in parallel. Note that if `A` is very large, it is possible that there would not be enough threads available to cover all the elements. In that case, each thread should square more than one element of `gpu_A`. It is possible to write code that is flexible enough to handle both cases, however it will slightly slow down as a result of the additional checks. Once every GPU thread has finished its work, `gpu_A` must be copied back into `cpu_A`. Finally, both chunks of memory should be freed.

Possibly the most important note when doing GPU computing: by far, *the most computationally intensive part of a* CUDA *program will be the memory copies*. These should thus be avoided as much as possible. The best situation is to do what is shown in the Figure: one memory copy at the beginning, perform all computationally intensive tasks, and one memory copy at the end. This will not always be a practical option, but any optimized CUDA program should minimize these copies.

The next sections show how `vecsquare.cu` accomplishes the above list of tasks. Note that the `main()` function in Program 1 has been broken into components that correspond to the steps in Figure 2.2. Step (1), allocating and initializing CPU memory, is nothing new from C and so I leave this out. I also leave out freeing CPU memory with the `free` command in step (6), though will describe how to deallocate GPU memory. I also describe steps (3) and (5), the memory copies between the CPU and GPU, in the same section.

2.2.2 GPU Memory Allocation

The following two lines will allocate a contiguous `4*N`-byte chunk of data on the GPU:

```
int* gpu_A;
cudaMalloc(&gpu_A, N*sizeof(int));
```

Data sizes on the GPU are the same as the CPU; with an `int` occupying four bytes (`double` is eight, `char` is one, etc.). Access to this chunk is performed through a pointer to the start of it `gpu_A`, declared as you would in C. The `cudaMalloc` function, newly available through the CUDA compiler, will allocate the number of bytes specified by its second parameter contiguously starting from the point passed by the first parameter. Note that this parameter must be passed by reference, since the GPU memory manager must first find a chunk of memory large enough and assign its starting location to `gpu_A`, changing its value.

Program 1 `vecsquare.cu`

```
#include <stdio.h>

#define N 100000
#define THREADS_PER_BLOCK 512

__global__ void vecsquare(int* v) {
   int index = blockIdx.x*blockDim.x + threadIdx.x;
   if (index < N)
      v[index] = v[index] * v[index];
}

int main() {

   // (1) Allocate and initialize CPU memory
   int* A = (int*) malloc(N*sizeof(int));
   for (int i = 0; i < N; i++)
      A[i] = rand() % 10;

   // (2) Allocate GPU memory
   int* gpu_A;
   cudaMalloc(&gpu_A, N*sizeof(int));

   // (3) Copy CPU to GPU
   cudaMemcpy(gpu_A, A, N*sizeof(int), cudaMemcpyHostToDevice);

   // (4) Run GPU code
   int numblocks = N / THREADS_PER_BLOCK;
   if (N % THREADS_PER_BLOCK != 0)
      numblocks++;
   vecsquare<<<numblocks, THREADS_PER_BLOCK>>>(gpu_A);

   // (5) Copy GPU to CPU
   cudaMemcpy(A, gpu_A, N*sizeof(int), cudaMemcpyDeviceToHost);

   // (6) Free GPU and CPU memory
   cudaFree(gpu_A);
   free(A);

   return 0;
}
```

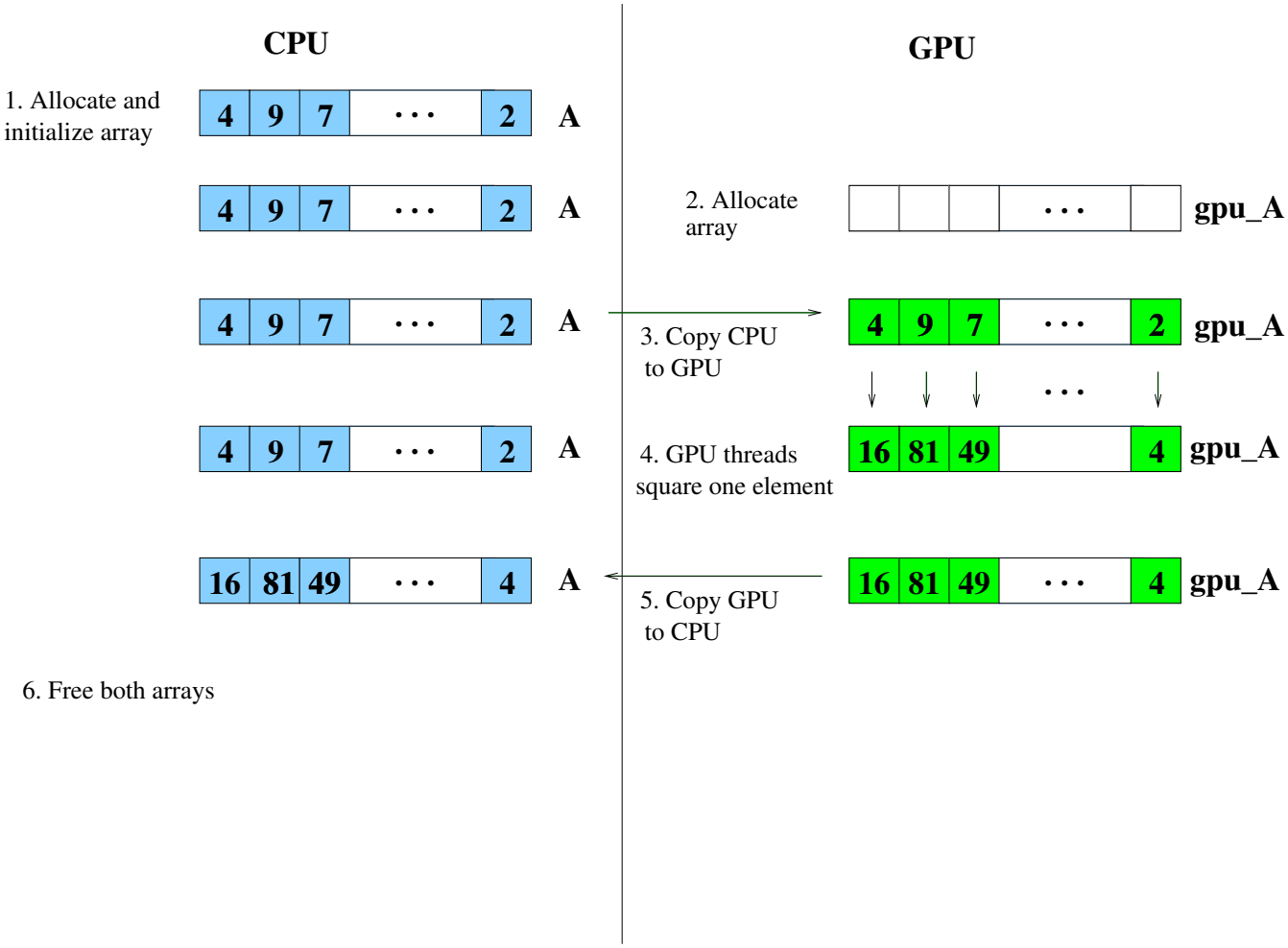

Figure 2.2: Control flow diagram illustrating the steps to squaring an array of integers in parallel on the GPU. Memory must first be allocated and initialized for the array on the CPU. Next, a large enough chunk gets allocated on the GPU and those values get copied from the CPU. GPU threads can then square elements in parallel. Finally, the array with the squared values is copied back to the CPU. Before the program terminates, both arrays must be deallocated.

2.2.3 Copying Between the CPU and GPU

The CUDA function `cudaMemcpy` is responsible for transferring data between the CPU and GPU memories. Dataflow can take place in either direction, specified by a parameter. This function accepts a total of four parameters: in order, (1) a pointer to the start of the target chunk, (2) a pointer to the start of the source chunk, (3) the amount of data to copy in bytes, and (4) the direction of dataflow (either CPU to GPU, or GPU to CPU). Hardware terminology often refers to the 'CPU' as the 'host' and the 'GPU' as the 'device'. Parameter #4 will either be: `cudaMemcpyHostToDevice` (CPU to GPU) or `cudaMemcpyDeviceToHost` (GPU to CPU), both unique constants defined by CUDA. The following thus copies `4*N` bytes from `A` to `gpu_A`:

```
cudaMemcpy(gpu_A, A, N*sizeof(int), cudaMemcpyHostToDevice);
```

And the following does the reverse (`4*N` bytes from `gpu_A` to `A`):

```
cudaMemcpy(A, gpu_A, N*sizeof(int), cudaMemcpyDeviceToHost);
```

Note that in this case both parameters are passed by value, since neither of their addresses change (only the contents in their respective locations). The important thing to remember (especially for students) is that *the target is always first*. This holds parallels with many assembly languages.

2.2.4 Calling GPU Code

We now come to the meat and potatoes of our CUDA example, the actual execution of the GPU code. While this was grouped into a single step, it really involves several. First, we must specify how many cores and threads from the graphics card that we want to use. We also must write the actual GPU function (these are called CUDA *kernels*) `vecsquare`. And finally, we must call `vecsquare` using the appropriate number of GPU cores and threads.

Number of Cores and Threads

As mentioned in the Introduction, a graphics card offers strong opportunities for parallelism because of its large number of cores which are each largely multithreaded. The GTX780 for example contains 2,304 cores that each can handle as many as 1,024 threads. Full specifications for any GEForce card can be attained through its website (`www.geforce.com/hardware`). The maximum number of threads per core is determined by the device's compute capability, and as mentioned an easy-to-use chart is available through Wikipedia (see section 2.1.2).

We should now move away from the terminology 'core' and towards the term 'block', which holds more commonality with the syntax of CUDA. GPU threads are allocated by the device in units of *blocks*, and each block is assigned to one core. The CUDA programmer must define the number of threads per block they desire, not exceeding the maximum possible defined by the compute capability (for the GTX780, this would be 1,024). At the top of `vecsquare`, note that we had defined `THREADS_PER_BLOCK` to equal 512. It is not always best to use the maximum, since higher numbers of threads will place a larger burden on the thread scheduler and potentially cause slower execution. Trial-and-error is the best solution to finding the optimal number of threads per block for your application (and it will vary based on application). On Kepler architectures (a common one for GEForce) a generally good rule of thumb is to use a multiple of 32, as threads are executed in units of 32-thread *warps* and this will ensure no empty warp slots.

Since the number of threads per block is hardcoded, we now just must determine the number of blocks. There must be enough blocks such that there are at least `N` total threads, since we wanted one thread per

element of A which is size N. We can thus start by dividing N by THREADS_PER_BLOCK. However unless N is a multiple of THREADS_PER_BLOCK, we will undershoot. We thus add one additional block if that happens. This is illustrated by the following lines of code:

```
int numblocks = N / THREADS_PER_BLOCK;
if (N % THREADS_PER_BLOCK != 0)
    numblocks++;
```

For our particular case (N is 10000 and THREADS_PER_BLOCK is 512), we will allocate a total of 20 blocks. This will overshoot in terms of thread count ($20 * 512 = 10240$), which we must do to ensure that every element gets squared. We will just have to watch when writing our kernel code, described in the next section.

Kernel Code

Recall that our GPU code was contained within a single kernel vecsquare:

```
__global__ void vecsquare(int* v) {
    int index = blockIdx.x*blockDim.x + threadIdx.x;
    if (index < N)
        v[index] = v[index] * v[index];
}
```

A GPU kernel is signified by the __global__ classifier which precedes the function return type. vecsquare accepts a single parameter v for the input vector, and each thread will square its respective elements in place. Note this only works because each thread is accessing only its individual element of v; if multiple elements were being accessed and/or changed race conditions would occur. There are ways to deal with this that avoid using two arrays, which are later discussed in Optimizations. v is assumed to be allocated in what is known as GPU *global memory* which is shared by all threads of all blocks (CUDA thus uses a shared memory architecture). This memory is the largest, but also the slowest. Other types of memory are also discussed later.

With every thread able to access v, CUDA threads concurrently execute vecsquare from top to bottom. The parameters that differ between threads are used in the first line of the function:

- blockIdx.x: The block number of the current thread (you could also see this as an identifier for each core).
- blockDim.x: The number of threads per block (for this particular application, 512).
- threadIdx.x: The unique identifier for this thread, within its block. Note: This is NOT unique across multiple blocks, which is why we also need blockIdx.x.

We can thus envision threads in the block structure form recalled in Figure 1.1. Each *block* will have its own unique identifier, as will each *thread* within each block. Therefore, the tuple (*blockid*, *threadid*) uniquely identifies each GPU thread. To correctllyy access v however, we must be able to compute a unique single index for each thread using *blockid* and *threadid*. We can observe that for our example with a block dimension of 512 threads, each block will have threads numbered 0-511. These threads in block 0 should square elements 0-511 of v. Those same threads in block 1 should square elements 512-1023 of block v, and so on for block 2, etc. By taking the block identifier and multiplying by 512 (but we can generalize

it to use the 'block dimension'), and adding the thread identifier, we achieve this pattern as shown by the following Table:

blockId	*threadId*	$blockId * blockDim + threadId$
0	0	0
0	1	1
0	2	2
...	...	...
0	511	511
1	0	512
1	1	513
...	...	...

The unique index for each thread can thus be computed by this expression:

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

One may question the usage of the `.x` member of each identifier. CUDA actually allows blocks to have multiple dimensions of threads, so there are members for `.y` and `.z` (these are assumed to be one if unspecified). The *y*- and *z*-dimension are helpful for instance if a kernel performs a *2D* or *3D* matrix computation, allowing threads to be accessed by *(x, y, z)* coordinates that map more directly to the structure of the underlying data. Even if multiple dimensions are used however, the total number of threads per block can still not exceed the compute capability boundary. For example if I were running *2D* thread blocks on the GTX780 and desired a *y*- dimension of 32 threads, I could not have an *x*-dimension larger than 32 since $32 * 32 = 1024$. Most often however, an *x*- dimension will be sufficient.

The `if` statement that follows will check to make sure that the element of `v` being accessed by this thread is not out of bounds. We do this because as mentioned we will likely overshoot on our thread count, since CUDA does not allow partial thread blocks to be allocated. These two operations: computing unique thread indices and bounds checking are very common within CUDA kernels.

```
if (index < N)
    v[index] = v[index] * v[index];
```

Once we ensure that our index is in bounds, we can perform the in-place squaring, multiplying the corresponding element of `v` by itself, squaring it, and placing it back into `v` at the same spot.

Kernel Call

A CUDA kernel call proceeds very similarly to a C function call, with the extra requirement that you must specify the number of threads and blocks. These are passed within the <<< and >>>, uniquely defined in the CUDA language. Thus the following line calls `vecsquare` with the correct number of threads and blocks from `main()`.

```
vecsquare<<<numblocks, THREADS_PER_BLOCK>>>(gpu_A);
```

Note that `gpu_A` must be passed into `v`, since GPU threads can only access GPU memory.

The kernel call is also the place to specify multi-dimensional blocks, if so desired. If a single integer is specified for the amount of threads per block, it is assumed for all blocks to only have *x*-dimension. CUDA provides a `dim3` structure that can alternatively be passed as this parameter. Thus the above would be equivalent to:

```
dim3 myDim(THREADS_PER_BLOCK, 1, 1);
vecsquare<<<numblocks, myDim>>>(gpu_A);
```

But you just as easily do a $32x32$ block:

```
dim3 myDim(32, 32, 1);
vecsquare<<<numblocks, myDim>>>(gpu_A);
```

Note the above would require modifying `vecsquare` to also use `threadIdx.y`, or not all elements will be squared.

2.2.5 GPU Memory Deallocation

Deallocating memory on the GPU involves using the CUDA equivalent of `free`, which is `cudaFree`. This function can accept any valid pointer to GPU memory and will free the chunk pointed to by its parameter for other uses. A freed pointer can be re-allocated memory through `cudaMalloc`, though the memory address to which it points will likely change.

Chapter 3

Advanced CUDA Programming

In this section, I describe some additional techniques of which to be aware when programming with CUDA. For example, up to this point we have only used one type of GPU memory. However, the GPU (like the CPU) does have a memory hierarchy. An important distinction from the CPU is that the programmer has much more power on the GPU to control which data goes in which memory. In addition, we have made no attempts up to this point to optimize GPU code. However, with the typically large-scale data involved with GPU applications, even small optimizations can create enormous improvements in runtime. It should be emphasized, however, that implementing GPU optimizations and measuring their performance gains is a trial-and-error process. Depending on the hardware configuration of the graphics card and the particular application, the same optimization may yield different gains (losses) in performance.

3.1 Using Shared Memory

Shared memory on the GPU is a type of *cache* memory that is accessible to all threads in the same block (thus each core contains one shared memory unit). Different types of graphics cards have different amounts of shared memory per core, typically in the KB range at the time of this release.

Shared memory is useful for situations where, for example, you need to perform some data processing on an original dataset, but must retain the original dataset. In bioinformatics, DNA processing is a good example. However, an even simpler example (one that has been used in CS310) is Program 2, which takes a string of characters and reverses them. The string size is assumed to have been stored in an integer constant `N`, which has been `#defined`.

Program 2 GPU Kernel that reverses a string.

```
__global__ void reverse_Global(char* data, char* reversedData) {
    int index = blockIdx.x*blockDim.x + threadIdx.x;

    if (index >= N) return; // Out of bounds

    else
        reversedData[(N-1)-index] = data[index];
}
```

The `index` for each thread was calculated using a similar formula as the previous example, and like that example the `index` was checked to be sure to be in bounds, in case of overshooting the thread count. If its `index` is valid, each thread takes its unique element of the input string `data`, and copies it to the appropriate spot in `reversedData` (character 0 will be placed in spot `N`-1, character 1 in spot `N`-2, and so on).

One part of this code that is massively inefficient is the use of the second parameter `reversedData`. This is because we are not certain of the order of thread execution, and so cannot simply swap characters in `data`. That second parameter will require a call to `cudaMalloc` to allocate `N` bytes in GPU memory, and also a call to `cudaMemcpy` to copy those `N` bytes from the GPU to CPU upon kernel completion. Both of these operations, particularly for large `N`, contribute to the bottleneck of such an application. It would be much better to perform just one of each of these operations for `data`, rather than for both `data` and `reversedData`.

This can be done using shared memory as shown in Program 3, which will instead store characters in the reversed string in cache memory. `tmp` is declared to be an array of 1024 characters, available on each core but importantly, not shared across cores. We thus assume that we are calling this kernel with 1024 threads per block, and thus every thread in each block can simply use its `threadIdx.x` to gain a unique spot in its respective shared memory. Each thread places its unique element of `data` in their spot in shared memory, and then only copies that element back to `data` in the reversed spot after a call to `__syncthreads()`. `__syncthreads()` acts as a barrier. All active threads must reach a call to `__syncthreads()`, and only then can all continue. Note this requirement enforces us to also include a `__syncthreads()` statement for the threads that are out of bounds, since otherwise these threads would never reach a `__syncthreads()` causing the valid threads to infinitely wait. `__syncthreads()` should be used only as necessary, as it does cause all threads to be bottlenecked by the speed of the slowest. However for large applications, the savings gained through shared vs. global memory, which are similar to those gained by general SRAM vs. DRAM when performing CPU processing, may very well outweigh these costs.

Program 3 GPU Kernel that reverses a string, using shared memory.

```
__global__ void reverse_Shared(char* data) {
    int index = blockIdx.x*blockDim.x + threadIdx.x;

    if (index >= N) {
            __syncthreads();
            return; // Out of bounds
    }
    __shared__ char tmp[1024];
    tmp[threadIdx.x] = data[index];
    __syncthreads();   // THREAD BARRIER

    data[(N-1)-index] = tmp[threadIdx.x];
}
```

3.2 Other Types of Memory

CUDA makes two other types of memory available: constant and texture. Constant memory is a very small global cache that is initialized at the start of program execution and cannot be changed after that point. It is most often used for small, simple operations that need only be executed once, i.e.:

```
__constant__ unsigned int core_courses = {143, 221, 310, 320};
```

Texture memory can also prove efficient on many graphics cards as it is accessed by separate Texture Mapping Units (TMUs), which historically were used in later rendering stages of graphics pipelines. If the computational units become heavily utilized, it can be optimal to place constant data into texture memory using the `__restrict` keyword. We could have done this for example with the first `reverse_Global`, since `data` was not modified:

```
__global__ void reverse_Global(const char* __restrict data, char* reversedData) {
```

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

    if (index >= N) return; // Out of bounds

    else
        reversedData[(N-1)-index] = data[index];
}
```

3.3 Avoiding Thread Divergence

CUDA threads, executing concurrently, will achieve maximum performance if they each take roughly the same amount of time to execute. As threads are launched in units of warps, there will always be a bottleneck associated with the slowest thread in each warp. With respect to assembly language, the type of instruction that will be the biggest enemy to achieving optimal functionality here is the *branch*. This is because some threads will take the branch and some will not, so different instruction counts get executed by each thread, in addition to some threads incurring overhead depending on the branch prediction strategy. When kernel threads consume a wide variety of execution times for their respective tasks, this is called *thread divergence*.

Thus when possible, conditional statements should be avoided in kernel functions, as should loops (a logical extension, since they involve some kind of condition). Take the following simple example below, which takes an integer and rounds it to the next largest even number. In other words if the integer is odd you add one to it, otherwise keep it the same.

```
if (x % 2 == 1)
    x = x + 1;
```

If this code snippet appeared in a kernel function, certain threads would execute the statement `x = x + 1` and others would not, bottlenecking each warp with (likely) those that executed the statement. With respect to avoiding thread divergence, it would be better to run the equivalent code:

```
x = x + (x % 2);
```

Despite this being less readable, it avoids the conditional clause and any branch statements at all, creating roughly the same amount of work for each thread and improving overall efficiency. There are other techniques that can be used to avoid thread divergence, for example you can use loop unrolling if you know the number of loop executions at compile time. Particularly if CUDA is used in a hardware course, reviewing optimization techniques used in assembly language would provide some guidance on helpful techniques to optimize CUDA code.

3.4 Optimizing Memory Accesses

When writing CUDA code, it is good practice for the programmer to constantly keep in mind the type of memory they are accessing, and if it is optimal. For example, it will be far cheaper to access the local on-chip memory of a thread compared to the larger global memory (which uses DRAM technology). Consider the following code segment:

```
__global__ void f(int* arr) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < arr[i]; j++) {
            ...
            ...
        }
    }
}
```

It is clear from this code that `arr` is a structure in global memory. Note that each iteration of the inner `j` loop does a comparison between `j` and an element `arr[i]`, which is in global memory but does not change between iterations of the `j` loop. It is thus more efficient to store that element in a local variable, just outside of the `j` loop:

```
__global__ void f(int* arr) {
    for (int i = 0; i < M; i++) {
        int tmp = arr[i];
        for (int j = 0; j < tmp; j++) {
            ...
            ...
        }
    }
}
```

If `arr[i]` is very large, a large number of accesses to the slowest global memory would be saved through this one small change. Minimizing accesses to slower memory components and enforcing faster ones whenever possible is the best practice. As with general C programming, the keyword `register` can recommend to the compiler to place a significantly accessed variable in register memory.

3.5 Operation Count and Strength Reduction

Reducing operation counts and strength are basic techniques that can be used to make GPU code more efficient. They can actually be used to make any general code more efficient, but often have larger implications for GPU code because of the generally computationally intensive nature of such kernels, and should be used whenever possible. Take for example the code snippet below:

```
    float root1 = (-b + sqrt(b*b - 4*a*c)) / 2*a;
    float root2 = (-b - sqrt(b*b - 4*a*c)) / 2*a;
    ...
}
```

Local variables should be used to reduce operation *counts*, or the number of arithmetic operations performed. Since `root1` and `root2` use almost the same expression, we can store portions of those expressions in local variables, as follows:

```
    float sqrt_discrim = sqrt(b*b - 4*a*c);
    float _2a = 2*a;
    float root1 = (-b + sqrt_discrim) / _2a;
    float root2 = (-b - sqrt_discrim) / _2a;
    ...
}
```

Note that by increasing our use of local variables, we took our code from one addition, three subtractions, eight multiplications and two divisions (fourteen total) to one addition, two subtractions, four multiplications and two divisions (nine total), saving clock cycles. We also saved one function call, to `sqrt`.

In addition, the type of operation can significantly impact runtime. For example, multiplication consumes many more clock cycles than addition and subtraction. Division is very inefficient on GPUs [5]. It is thus best to use strength reduction, or reduction to an operation with fewer clock cycles, whenever possible. We can do this with `2*a` for example, and make it `a+a`. We also can observe that `2*2*a` is equal to `4*a`, and use this in the computation of `4*a*c`, in the end creating the following:

```
    float _2a = a+a;
    float _4a = _2a+_2a;
```

```
    float sqrt_discrim = sqrt(b*b - _4a*c);
    float root1 = (-b + sqrt_discrim) / _2a;
    float root2 = (-b - sqrt_discrim) / _2a;
    ...
}
```

This final optimized code has three additions, two subtractions, two multiplications and two divisions (nine total). Though the count is the same as before, two multiplications became additions, saving clock cycles.

3.6 Other Useful Tools

In addition to these, there are many other techniques that can be used to improve GPU code. I recommend the following resources, and have included their citations and general purposes:

1. **CUDA Programmer's Guide**, `http://docs.nvidia.com/cuda/`. A comprehensive guide to the features of CUDA and their relations to the underlying hardware of the graphics card. Useful examples to illustrate concepts.
2. **CUDA By Example**, [7]. A useful paperback guide on how to accomplish tasks in CUDA, demonstrated primarily through code samples.
3. **Computer Organization and Design, Appendix C** [12]. Contained in the appendix of a popular textbook for hardware courses, this resource provides a solid presentation of the architecture of the GPU.
4. **GPU Technology Conference**, `http://www.gputechconf.com/`. An annual conference that grants public access to its talks each year, which often outline useful techniques and applications for GPU code. Many of the aforementioned techniques, for example, I learned while attending this conference in 2014.

Finally, please feel free to contact me (`tcickovs@fiu.edu`) at any time with questions on material in this tutorial, or general CUDA questions and their application to course material at Eckerd College.

Bibliography

[1] Jon Bentley. Programming pearls: Bumper-sticker computer science. *Communications of the ACM*, 28(9):896–901, 1985.

[2] Brooke Crothers. End of Moore's Law: It's Not Just About Physics. online, 2013.

[3] Joseph Cornwall. 1080p and the acuity of human vision. *Online A/V Magazine*, 4(2):1, 2007.

[4] J. Fung, F. Tang, and S. Mann. Mediated reality using computer graphics hardware for computer vision. In *Proceedings of the International Symposium on Wearable Computing 2002*, ISWC'02, pages 83–89, 2002.

[5] W. W. Hwu. *GPU Computing Gems: Emerald Edition*. Elsevier, 2011.

[6] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[7] E. Kandrot and J. Sanders. *CUDA By Example: An Introduction to General-Purpose GPU Computing*. Addison-Wesley, 2010.

[8] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. ArrayFire: a GPU acceleration platform. volume 8403, pages 84030A–84030A–8, 2012.

[9] Nitin Gupta. Texture Memory in CUDA. online, 2014.

[10] NVIDIA. PTX ISA Version 2.3. online, 2013.

[11] NVIDIA. CUDA toolkit archives. online, 2016.

[12] D. Patterson and J. L. Hennessy. *Computer Organization and Design*. Elsevier, 5 edition, 2014.

[13] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[14] J. C. Sweet, R. J. Nowling, T. Cickovski, C. R. Sweet, V. S. Pande, and J. A. Izaguirre. Long timestep molecular dynamics on the graphical processing unit. *Journal of Chemical Theory and Computation*, 9(8):3267–3281, 2013.

[15] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using data parallelism to program gpus for general-purpose uses. *SIGOPS Oper. Syst. Rev.*, 40(5):325–335, October 2006.