GPU Programming for Mathematical and Scientific Computing

Ethan Kerzner and Timothy Urness
Department of Mathematics and Computer Science
Drake University
Des Moines, IA 50311
ethan.kerzner@gmail.com
timothy.urness@drake.edu

Abstract

Graphical processing units used for mathematical and scientific computing are known as general purpose graphical processing units (GPGPUs). This paper is an introduction to the most popular GPGPU technology, NVIDIA's Compute Unified Device Architecture (CUDA). We approach CUDA from the perspective of a software developer, discussing the structure and organization of programs to explain the function of the GPU. We propose a potential application of GPGPU programming, the parallel analysis of a Hidden Markov Model, and discuss arithmetic error on the graphical processing unit.

1 Introduction

Improving computer performance is generally achieved by increasing the clock rate of the central processing unit. Moore's Law implies the speed of processors should double every eighteen months. However, the physical components of processors are reaching physical boundaries of size and power consumption. As a result processor performance is shifting toward parallelism, increasing the number of concurrent operations. This is apparent in the multiple core processors now available in most personal computers.

Graphical processing units (GPUs) are the paragon of parallel computing and multicore architecture. They are designed to perform calculations on large amounts of independent data such as pixels. A typical GPU contains many multiprocessors with hundreds of processing cores. Parallel performance is measured in computational throughput or bandwidth instead of clock rate. The throughput of a high end graphics card is on the order of four teraflops per second. In recent years, graphics cards are emerging as a major computing power in fields other than graphics.

General Purpose Graphical Processing Units, GPGPUs, refers to the use of GPUs for mathematical and scientific computing. The two largest graphics card manufacturers, NVIDIA Corporation and Advanced Micro Devices (AMD), provide software development kits for writing programs on their GPUs. These platforms are called Close to Metal and Compute Unified Device Architecture, CUDA, respectively. As of November, 2009, both companies have agreed to develop hardware for a new standard, OpenCL. However, NVIDIA's CUDA technology is currently dominant [1].

Most new NVIDIA graphics cards are CUDA capable. CUDA programs are generally created using a language integration interface which allows for the execution of C functions on graphics cards, the *C Runtime for CUDA*. This paper focuses on *C For CUDA*, a derivation of C for GPU computing. Other high level languages can run on graphics cards as well including JAVA with *JaCUDA*, and Python with *PyCUDA* [1]. NVIDIA provides tools for aspiring GPU programmers for free on their website. These resources include necessary drivers, a toolkit with compilers and libraries as well as a software development kit with sample code.

GPGPU programming is writing massively parallel programs for execution on personal computers. This paper provides an introduction to creating CUDA programs for mathematical and scientific computing. Section 2 is a description of CUDA programs, specifically memory, thread organization and the CUDA compiler. An application of CUDA programs, the parallel analysis of a Hidden Markov Model, is discussed in Section 3. Issues of floating point arithmetic involved with this model and their performance considerations are examined in Section 4. A brief summary is in Section 5.

2 Description of CUDA

C and CUDA programs are structured similarly; almost all of the features of C can be used in CUDA with a few additional libraries for controlling operations on the GPU. Programs are run mainly on the central processing unit (host) and *kernels* of parallel code are executed on the GPU (device).

The structure of CUDA programs is explained by the organization of processing units. The main function is executed by a single thread on the CPU. Kernels consist of many lightweight threads and are executed by any function on the host. Kernel launches change control of the program from CPU to GPU, the CPU pauses while the GPU is running. After the kernel finishes executing, the CPU resumes running the program.

2.1 Memory

Programs execute on two different processing units, the host and the device, each with its own memory. The host relies on traditional random access memory (RAM), while the device typically uses Graphical Double Data Rate (GDDR) RAM. The GPU cannot access the memory of the CPU and special functions are required to transfer between the two. These functions must be launched on the host. Examples of such functions include *cudaMalloc*, very similar to *malloc* in C and *cudaMemcpy* which transfers data between host and device [1].

Typically a CUDA program begins with initialization of variables and memory allocation, followed by data input on the host. These data are copied to the device using the *cudaMemcpyHostToDevice* function. Information stored in memory on the device will remain for the duration of the program; it is preserved between kernel launches. After kernels have run, data are copied back from the device to the host.

GPUs have a memory hierarchy similar to that of a CPU. Both have fast memory local to each processor (cache), and main memory (RAM). The fast local memories of a GPU are registers and shared memory. Each processing core on the GPU has access to registers and shared memory. Different forms of memory are discussed extensively in [2] and [3].

On the GPU registers are local to each thread and are used for loop control and storing instructions. *Shared memory* is small storage relative to groupings of threads called blocks. Having access to the same memory allows groups of threads to cooperate on their calculations. However, bank conflicts arise when threads simultaneously attempt to access the same data. If data exists in the device memory it will remain there the duration of the program. It must be explicitly moved into shared memory by a block during the kernel launch. Data remains in shared memory only for the duration of the blocks execution.

There are many strategies for efficient use of GPU memories. Similar to the CPU, the premise is to limit access to global memory during kernel execution. Threads are good at performing computations quickly; having them access memory significantly impacts performance.

2.2 Thread Organization

CUDA is single program multiple data which means one set of instructions is executed by many threads. A kernel launch creates up to 64K threads on the device. There is little computational overhead in creating threads. The quantity and organization of threads is task specific and can be determined statically or dynamically [1].

Each thread has a unique set of identifiers used to control memory access and data processing. Threads are grouped into blocks, and blocks are grouped into a grid. Threads and blocks are organized in three dimensional arrays. Each thread has a unique identity within its block determined by its indices. These indices are defined for every thread of the kernel and can be accessed by the programmer to control thread execution and memory access.

Blocks, which are groups of threads, are an important part of CUDA architecture. Each block is launched on single multiprocessor on the device. These multiprocessors have many cores which execute threads individually. Because of their proximity, threads within a block can synchronize execution and share memory. Also, NVIDIA designed CUDA with the objective of scalability. If a kernel is launched with too many blocks for the number of multiprocessors, the blocks are executed sequentially, although within a block the execution of threads is in parallel. Increasing the number of multiprocessors on the device increases the number of concurrent blocks executing. This does not affect the behavior of the program because the blocks are independent [3].

The code in Figure 1 is a simple CUDA program to demonstrate the launch of CUDA kernel, the organization of threads within a block, and the transfer of data between host and device.

The function starting with *__global__ device* is a kernel to be executed on the device. The variable, threadIdx.x is defined for every thread launched during execution. It is specified in the launch parameters that there will be a one dimensional block containing 256 threads launched. The final value of hostArray is [0, 1, 2, ... 255].

```
__global__ device simpleKernel(float* array){
   array[threadIdx.x] = threadIdx.x;
}

int main(){
   //intialize variable and allocate memory
   float* host_array, device_array;
   int array_size = 256;
   malloc(host_array, array_size*sizeof(float));

   cudamMalloc(device_array, array_size*sizeof(float));

   //specify launch parameters, and run kernel
   dim3 gridDim(1,1,1);
   dim3 blockDim(256,1,1);
   simpleKernel<<<gridDim, blockDim>>>(device_array);

   //copy results back to host
   cudaMemcpy(host_array, device_array, array_size*sizeof(float),
   cudaMemcpyDeviceToHost);
   return 0;
}
```

Figure 1: A simple CUDA Program

Additionally, independent execution of blocks on the device allows for creation of tiled algorithms. Tiled algorithms break up large tasks into smaller portions. Each portion is then executed in parallel. A loop is used so that the tiles are executed over the entire data to be processed. Once an algorithm is created for data of a specific size, implementing a loop to run it over an arbitrary size input is a simple task [2].

2.3 CUDA Compiler

The CUDA Compiler, *nvcc*, is used to compile code on the device. When compiling a program, all non-CUDA types are forwarded to a C compiler on the host. CUDA programs can be divided into separate files of host code, device code, and any necessary header files linked together at compile time. Setting up CUDA programs to compile correctly on the host and device is a daunting task. However, the CUDA Software Development Kit (SDK) includes sample programs ready to compile.

The CUDA SDK is available on Microsoft Windows, Mac OS X, and most major Linux distributions [1]. Sample code, and necessary makefiles or Visual Studio Solution files are included. NVIDIA also provides tools for emulating GPUs, to develop CUDA programs without having access to CUDA enabled GPUS. We used the makefiles and project templates provided with CUDA 2.3 in our applications of GPGPU programming.

3 Application: Hidden Markov Model

GPGPU technology is emerging in the world of scientific computing. One application of CUDA is the analysis of a Hidden Markov Model (HMM). HMMs are stochastic models useful in machine learning, speech recognition and bioinformatics [4]. This section discusses the challenges of implementing such analysis in CUDA.

3.1 Description of HMMs

HMMs are discrete stochastic models consisting of states and emissions. At each time step, the model has probabilities defined for transitioning between states and specific emissions at each state. Our notation of HMMs is in Figure 2.

Set of states: $Q = \{q_1, q_2, q_3 \ldots q_n\}$

Transition Probability Matrix A, such that $A_{i,j}$ is $P(q_i \textit{ at time t-1, } q_j \textit{ at time } t)$

Set of Emissions: $E = \{b_1, b_2, b_3 \ldots b_k\}$

Emission Probability Matrix B such that $B_{i,j}$ is $P(\textit{Emission i from state j})$

Initial State Probability: $\Pi = \{\pi_1, \pi_2, \pi_3, \ldots, \pi_n\}$

Figure 2: The Definition of a Hidden Markov Model

There are three questions which can be asked about a HMM and a given observation sequence $O = \{o_1, o_2, o_3, \ldots, o_n\}$:

1. How can we compute the probability of observing O from our model?
2. How do we determine the most likely series of states which explains O?
3. How do we adjust the parameters of the model to maximize the probability of a given observation sequence?

The algorithms for answering these questions are *The Forward Algorithm*, *The Viterbi Algorithm*, and, *The Baum-Welch Algorithm* respectively. A full description of HMM algorithms is found in [4] and [5]. We focus on *The Forward Algorithm* implemented in CUDA.

3.1The Forward Algorithm

We use *The Forward Algorithm* to find the probability of a given observation sequence. The input of our algorithm is a HMM and a list of observations, O. The algorithm is defined in two stages, *initial step forward*, and followed by many repetitions of *step forward*. The output of this algorithm is the probability of the model emitting observation sequence in question.

We define a new term, α (time t, state q, observation o_t), the probability of at time t being in state q and emitting observation o_t. During the initial step forward, the α for each state is calculated using the initial probability Π, and the emission probability, B. For each following time step α is calculated recursively, relying on α (t-1, q, o_{t-1}) to find α (t, q, o_t) The equations for calculating each α are shown in Figure 3.

At time t = 0, the α are calculated:

$$\alpha(t=0, \mathrm{q}=\mathrm{q}_1, \mathrm{o}=\mathrm{o}_1) = \pi_1 * B_{1,1}$$

At time t the $\alpha(t, \mathrm{q}=\mathrm{q}_1, \mathrm{o}=\mathrm{o}_1)$ is calculate recursively by finding the probability of moving to q_1 from every other state, multiplied by the emission probability of o_t

$$\alpha\,(\mathrm{t},\ \mathrm{q}_1\ \mathrm{o}_\mathrm{t}) = (\alpha\ (\mathrm{t}\text{-}1,\ \mathrm{q}_1, \mathrm{o}_\mathrm{t}) * A_{1,1} + \alpha\ (\mathrm{t}\text{-}1,\ \mathrm{q}_2, \mathrm{o}_\mathrm{t}) * A_{2,1} + \ldots\ \alpha\ (\mathrm{t}\text{-}1,\ \mathrm{q}_\mathrm{n}, \mathrm{o}_\mathrm{t})) * B_{t,i}$$

Figure 3: Calculation of α

At each step forward we calculate α for all states in the model. These are stored in a three dimensional array. The output of the algorithm is the sum of the final row of the array, the probability of ending in any state with the desired observations.

3.2 CUDA Implementation

At each time step, α values are calculated independently. This is the portion of the code executed in parallel. In our implementation, we focused on a small, fixed size HMM. In an actual implementation our sample algorithm would be modified to execute over an arbitrary size input.

The Forward Algorithm implemented in CUDA follows the typical structure of a program. Before execution, the arguments are copied into memory on the GPU. A three dimensional array α is allocated in device memory. The α array persists the length of the program.

The Forward Algorithm is described in Figure 4. Assume it is launched with a single block of threads such that one thread corresponds to each state in the model. The code is outlined for a single thread and executed on enough threads for the entire model. In our implementation we placed a loop around this function to execute for the desired number of time steps.

The matrices are structures with fields of arrays, height, width and depth to offset to the proper elements. For simplicity, it does not take advantage of shared memory nor does it efficiently allocate work among threads. These are two potential improvements for our model. The critical concern of the algorithm is in its arithmetic. The arithmetic hardware of CUDA is optimized for IEEE 754 Single Precision Floating Point Values (floats). Even in our basic example, the range of floating point values was not large enough; the results of our calculations were inaccurate because of numerical error.

```
__global__ device forward_algorithm_kernel(Matrix alpha, Matrix A,
Matrix B, int* o, int t){

        for(i = 0; i < alpha.width; ++){
                temp = temp + alpha[t][i][t] * A[threadIdx.x][i];
         }

         temp = temp * B[o[t]][ThreadIdx.x];
         alpha[t+threadIdx.x][i][t] = temp;
}
```

Figure 4: A Trace of the Forward Algorithm

4 Floating Point Arithmetic

The standard metric for measuring computation throughput is single precision floating point operations per second. However, floats do not have sufficient precision for mathematical and scientific computing. In our example, we observed underflow error as a result of the exponent of a floating number being outside the range of possible representations. We explored two methods of avoiding underflow error in CUDA programs. One solution from [5] consisted of using logarithms to scale values within the range of floating point representation. The other possible solution was the transition to double precision arithmetic. We compared implementation of these solutions in CUDA programs.

4.1 Comparison of Floating Point Methods

The use of logarithms does not increase precision of floating point values; it only scales the input to within a more acceptable range of values. We used two algebraic identities in our calculations involving logarithms. For multiplication: $\log(x) + \log(y) = \log(x * y)$ and for addition, the method implemented in [5]: $\log(1 + e^{(x-y)}) = \log(x+y)$. Logarithms also increase the computational overhead of calculations on the processor, and have the potential to result in more numerical error. However, working with logarithms allows continued use of floats and peak arithmetic performance on the GPU.

Double precision floating point values (doubles) are represented with twice the bits of single precision values. As a result, they can represent a substantially larger range of values. However, double precision arithmetic is slower than single precision arithmetic, and is not yet standard on all GPUs. Furthermore, double precision values require twice the memory which impacts performance because of increased transfer times between host and device as well as from global to shared memory.

4.2 Parallel Primitives

Parallel primitives are algorithms used as foundations for complex parallel methods [7].We used parallel primitives to measure the performance of CUDA GPUs using our two proposed solutions for the error in floating point arithmetic. The two primitives we examined are parallel prefix sum scan and array reduction. Both algorithms were modified two compare the performance of logarithmically scaled floating point and double precision arithmetic.

The input of array reduction is an array of values. Reduction applies a single binary operation, such as addition, to all the elements of that array and returns just that value, the sum of the array. We used an optimized algorithm for array reduction described by [7]. It was modified respectively for both forms of our arithmetic. A logadd method, described in the previous section was implemented on the device for our computations involving logarithms.

Prefix sum scan is the second parallel primitive we examined. The input of prefix sum scan is an array of values. The output is an array which every element is the sum of all preceding elements in original array. Prefix sum scan is important for parallel data processing because it balances thread work and exemplifies the challenges in memory conflicts. The work efficient prefix sum scan method we used was modified from [6].

Our parallel primitives were focused on simplicity. Using more computationally intense benchmarks would significantly improve our results. Even in our example without memory transfer, a large portion of execution time consisted of threads reading data from global memory. Having threads perform more computations per memory access would give us a more accurate comparison of the difference between arithmetic types on the GPU.

4.3 Performance Results

We recorded the performance of our parallel primitives on the machine described in Table 1. We used CUDA timers to record the GPU time for each kernel call. The tests were run 100 times on arrays of 512 elements and the results were averaged. We expanded our measure of performance to include memory transfer between the host and device. This is a more realistic measure of performance because total execution time includes such transfers. The results are shown in Table 2 and Table 3.

GPU:	NVIDIA GTX 260
CPU:	Intel Core i7 2.67Ghz
CUDA:	CUDA 2.3
OS:	XP 64 Bit

Table 1: The Test Environment

Array Reduction without Memory Transfer	Execution time (ms)
Logarithms	0.240628
Double Precision	0.22267
Prefix Sum Scan without Memory Transfer	Execution time (ms)
Logarithms	0.079243
Double Precision	0.076619

Table 2: Execution time without memory transfer

Array Reduction including Memory Transfer	Execution time (ms)
Logarithms	0.240286
Double Precision	0.292925
Prefix Sum Scan including Memory Transfer	Execution time (ms)
Logarithms	0.048898
Double Precision	0.077230

Table 2: Execution time including memory transfer

Not including memory transfer to and from the device, both methods executed in about the same amount of time. Without memory transfer, execution times of similar algorithms for double precision and logarithmic arithmetic were within 8 percent of each other for both benchmarks. Considering memory transfer between the GPU and CPU in our benchmarks, the use of logarithms was superior, executing 22 percent faster than double precision algorithms for array reduction, and 57 percent faster for prefix sum scan. This is a result of the larger size of double precision values.

We also found that the use of logarithms substantially decreased the accuracy of calculations of the device. The unit of least precision for logarithmic values on the device is 2 [1]. This causes a large loss of accuracy in calculations using logarithms.

For our purpose, the transition to double precision arithmetic will make GPGPU computing a stronger platform for mathematical and scientific computing. The small decrease in performance we observed is offset by the large increase in precision of calculations.

5. Conclusion

GPGPUs, specifically, NVIDIA Corporation's CUDA Technology are emerging for use in mathematical and scientific computing. The main performance metric of GPGPUs is floating point operations per second. Floating point values are not sufficient for computations involving values of extreme sizes. The solution to this lack of precision is best fixed by the transition to double precision arithmetic. Double precision arithmetic is only available on the newest generation of NVIDIA graphics cards with Hardware version newer than 1.3.

CUDA is currently the dominant GPGPU platform. Commonly, CUDA Programs are created using the *C For CUDA* runtime environment, writing high level code similar to C that execute on the GPU. NVIDIA provides drivers and tools for learning CUDA on their website [1].

In addition to the CUDA tools provided by NVIDIA, there are many other advances being made in GPGPU technology. Heterogeneous programming refers to applications that take advantage of the different strengths of GPUs and CPUs. Open Computing Language (*OpenCL)* is an emerging open heterogeneous platform from the Krohnos Group adopted by many corporations including NVIDIA, ATI and Apple [8]. NVIDIA currently chairs the OpenCL working group and supports cross platform development for GPGPU computing to increase the availability of programs which use the parallel processing power of GPUs.

References

[1] NVIDIA Corporation. CUDA Technical Training: Introduction to CUDA Programming. [Online]. Available: http://www.nvidia.com/

[2] H. Wen Mei and J. Stratton, “Programming Massively Parallel Processors,” Online Course Material, University of Illinois, Urbana Champaign. Available: http://courses.ece.illinois.edu

[3] R. Farber, CUDA, Supercomputing for the masses: Part 1-6 [Online]. Available: http://www.ddj.com/hpt-highperformancecomputing/

[4] L. R. Rabiner, A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. Proceedings of the IEEE 77(2): 257 – 286, 1989

[5] C. Liu (May 6, 2009). cuHMM: a CUDA Implementation of Hidden Markov Model Training and Classification [Online]. Available: http://code.google.com/p/chmm

[6] M. Harris, S. Sengupta, and J. D. Owens. “Parallel Prefix Sum (Scan) with CUDA.” In Hubert Nguyen, editor GPU Gems 3, chapter 39, pages 851 – 876. Addison Wesley, August 2007. Available: http://graphics.idav/ucdavis.edu/publications

[7] M. Harris, Optimizing Parallel Reduction in CUDA. [Online] Available: http://developer.nvidia.com/compute/cuda

[8] Khronos Group. OpenCL Overview: The Open Standard for Parallel Programming of Heterogeneous Systems. [Online] Available: http://www.khronos.org/opencl