5-1-2012

Accelerating noninvasive transmural electrophysiological imaging with CUDA

Martin Corraine

Accelerating Noninvasive Transmural Electrophysiological Imaging with CUDA

By

Martin Andrew Corraine

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Dr. Sonia Lopez Alarcon
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, NY
5/2012

Approved By:

Dr. Sonia Lopez Alarcon
Primary Advisor – R.I.T. Dept. of Computer Engineering

Dr. Linwei Wang
Secondary Advisor – R.I.T. PhD Program of Computing and Information Sciences

---

Dr. Roy Melton
Secondary Advisor – R.I.T. Dept. of Computer Engineering

Acknowledgements

I would like to thank my advisors Dr. Sonia Lopez Alarcon, Dr Linwei Wang, and Dr. Roy Melton for all their guidance. This research would not have been possible without their help. Thanks!

Abstract

The human heart is a vital muscle of the body. Abnormalities in the heart can disrupt its normal operation. One such abnormality that affects the middle layer of the heart wall (myocardium) is called myocardial scars. Just like any tissue in the body, damage to healthy tissue will trigger scar tissue to form. Normally this scar tissue is benign. However, myocardial scars can disrupt the heart's normal operation by changing the electrical properties of the myocardium. It is the most common cause of ventricular arrhythmia and sudden cardiac death. Leading edge research has developed a technique called Noninvasive Transmural Electrophysiological Imaging (NTEPI) to help diagnose myocardial scars.

However, NTEPI is hindered by its high computational requirements. Due to the parallel nature of NTEPI, Graphics Processing Units (GPUs) equipped with the Compute Unified Device Architecture (CUDA) by Nvidia can be leveraged to accelerate NTEPI. GPUs were chosen over other alternatives because they are ubiquitous in hospitals and medical offices where NTEPI will be used.

This project accelerated NTEPI with CUDA. First, NTEPI was profiled to determine where most of the time was spent. This information was used to determine what functions were chosen for CUDA acceleration. The accelerated NTEPI algorithm was tested for accurateness by comparing the outputs of the baseline CPU version to the CUDA version. Lastly, the CUDA accelerated NTEPI algorithm was profiled on three GPUs with different costs and features. The profiling was used to determine if any bottlenecks existed in the accelerated NTEPI algorithm. Lastly, CUDA specifications were identified from this profiling data to achieve the highest performance in NTEPI with and without cost as a factor.

Table of Contents

List of Figures

List of Tables

Glossary

NTEPI	Noninvasive transmural electrophysiological imaging
CUDA	Compute unified device architecture
CPU	Central Processing Unit
GPU	Graphics Processing Unit
CCAP	Compute CAPability
V+	Matrix-vector addition
V-	Matrix-vector subtraction
BLAS	Basic Linear Algebra Subprograms
LAPACK	Linear Algebra PACKage
ACML	AMD Core Math Library (Includes implementations of BLAS and LAPACK)
MAGMA	Matrix Algebra on GPU and Multicore Architectures
CUBLAS	Compute Unified BLAS
CGMA	Compute to Global Memory Access

Chapter 1 Introduction

Finding new noninvasive ways to image the heart's electrical System is vital to combating ventricular arrhythmia and sudden cardiac death. High-resolution delay Contrast Enhanced (CE) imaging is one imaging technique that aims to solve this problem. However, myocardial scars imaged with this method are correlated to but not identical to electrically defined scar substrates. NTEPI was developed to fill this void in diagnostic medicine. A drawback is the high computational complexity of the algorithm in transforming Body Surface Potentials (BSPs) to electrical activity in the myocardium. Long processing times are not acceptable in the medical world. Therefore, this thesis aims to improve the performance of NTEPI using Graphics Processing Units (GPUs) equipped with compute Unified Device Architecture (CUDA).

CUDA is a GPU architecture. GPUs are designed to render graphics: a highly data parallel computation. In addition, this architecture allows programmers to harness the computational power of the GPU in a general purpose manner.

CUDA was chosen as the accelerator for several reasons: First NTEPI can take advantage of the parallelism in CUDA enabled GPUs. Second, CUDA enabled GPUs are ubiquitous in desktops. Lastly, GPUs are significantly cheaper than Beowulf clusters. Since NTEPI contains mostly linear algebra operations the following libraries are used: Compute Unified Basic Linear Algebra Subprograms (CUBLAS) and Matrix Algebra on GPU and Multicore Architectures (MAGMA). Custom written kernels are also used to achieve the highest performance possible.

The remainder of the thesis is organized in the following manner. Chapter 2 provides the background on the research conducted. Chapter 3 describes how each function in NTEPI was accelerated using CUDA. Chapter 4 presents the various strategies used to accelerate NTEPI by leveraging the work in Chapter 3. Chapter 5 summarizes and concludes the research conducted.

Chapter 2 Background

In this chapter, the background of this thesis will be explained. First, relevant details on CUDA will be described. Next, the human heart will be briefly described since NTEPI focuses on this part of human anatomy. Electrocardiography will then be introduced because this process is used to acquire data for NTEPI. The Inverse Problem of Electrocardiography (IPECG) will be explained to finish laying the groundwork for NTEPI. NTEPI will then be briefly discussed. Lastly, previous work related to NTEPI will be presented.

2.1. CUDA

Graphics Processing Units (GPU) are application specific processors that are designed to render graphics, a highly data parallel computation. GPUs contain several simplistic processors called Streaming Multiprocessors (SM) that are used by several thousand hardware-managed threads. Due to the data on which GPUs work, things like branch prediction and superscalar architectures are not needed to achieve high performance. Therefore, more area on the die can be dedicated to Arithmetic Logic Units (ALUs).

GPUs in their infancy were dedicated to rendering graphics for video games. People then started to notice the computational power within these chips could be used for other applications. Using GPUs as general purpose processors became known as General Purpose Computation on GPUs (GPGPU). GPGPU helped start the development of CUDA.

Long before CUDA, the first GPU architecture was a fixed function architecture. There were processing elements that were dedicated to performing either vertex or pixel shader operations. In some cases, this architecture caused load imbalances and hurt performance. The successor was the unified architecture. All the processing elements were able to perform vertex and pixel shader operations. CUDA combined the unified architecture and added additional hardware to run general purpose applications on a GPU. For example, the ALUs were designed to be mostly IEEE 754-2008 compliant with floating-point numbers. In addition, an instruction set tailored for graphics as well as general purpose applications was leveraged [1]. Also, the processing elements were

allowed to arbitrarily read and write to memory [2]. Together, this new hardware allows the GPU to excel not only at graphics but also at general purpose applications.

NVIDIA created an application programming interface (API) to access the CUDA hardware in a general purpose way. The alternatives would be to painstakingly apply graphic APIs like OpenGL and Direct3D to general purpose computing or use a generic API called OpenCL for heterogeneous computing [2] [3]. The CUDA API added libraries to easily access the GPU hardware. Also, keywords were added to the C programming language. Together they formed the CUDA C programming language.

2.1.1 Launching a Kernel

A CUDA program consists of code that runs on one or more hosts (CPUs) and one or more devices (GPUs). This setup constitutes a heterogeneous computing System since there are at least two different types of processors that are executing the program. A CUDA program will execute sequential code on the host and parallel code on the device by invoking kernels. When a kernel call is encountered a few chores must be completed before the computation can be done (illustrated in Figure 1). First, the host instructs memory to be copied from the host to the device. Second, the host instructs the device to begin executing the kernel. By default, a kernel is called asynchronously with respect to the host, meaning control will immediately return back to the host. An asynchronous kernel call merely queues the kernel call on the device. This can be advantageous when the host can do computations while the device is executing a kernel. Third, the device begins executing the kernel. Fourth, the host instructs memory to be copied from the device to the host. As illustrated by the diagram, the host delegates much of the work but is responsible for none of the computation.

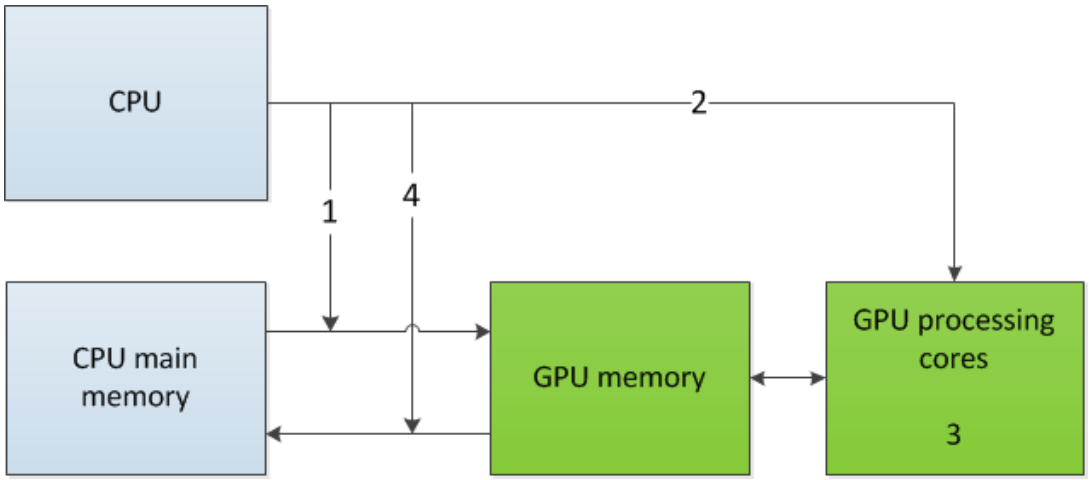


Figure 1: Steps to launch a kernel [9]

The overhead of transferring data back and forth between the host and the device can never be ignored because it adds to the total execution time. The *communication to computation ratio* (c-c ratio) helps determine a program's efficiency. A high c-c ratio characterizes an inefficient program that spends the majority of its time communicating rather than computing. Communication is not useful work and therefore must be limited or hidden to achieve the highest performance.

2.1.2 Threads

Each time a kernel is launched, a grid of lightweight hardware threads are spawned and managed by CUDA [2] [4]. A grid is composed of one or more blocks, and blocks are composed of one or more threads, illustrated in Figure 2. The grid and block can have up to three dimensions depending on the Compute CAPability (CCAP) of the GPU [1]. The CCAP describes what version of the CUDA architecture the GPU has.

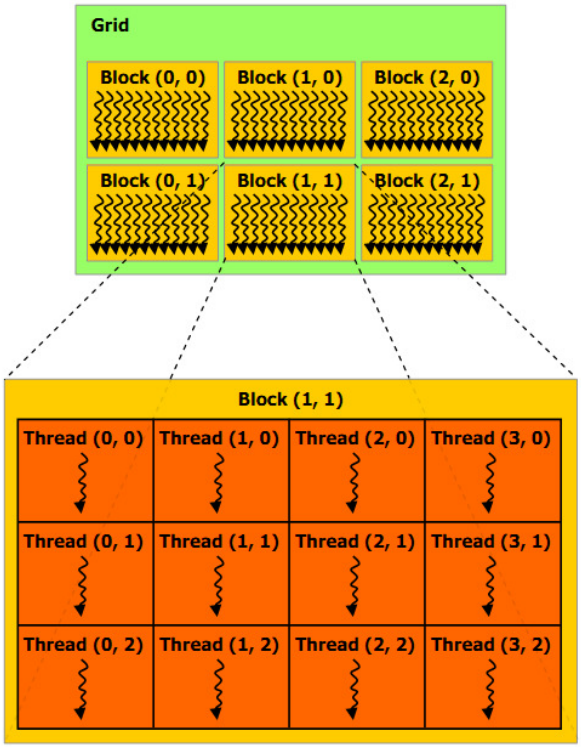


Figure 2: CUDA thread hierarchy [1]

CUDA threads are scheduled entirely by the hardware. In contrast, host threads are scheduled by the operating System. The unit for scheduling threads is called a warp. Warps are groups of 32 consecutive threads in a block [4] [1]. For example, A one dimensional block, *threadIdx.x* values from 0 to 31 form a warp and 32 to 63 form another warp. When a block has more than one dimension, the highest dimensions will have priority over the lower dimensions. For example, the threads with *threadIdx.y* equal to 0 will first be placed in linear order. Then threads with *threadIdx.y* equal to 1 will be placed into linear order and so on [4]. A block with two dimensions will be linearized as illustrated in Figure 3.

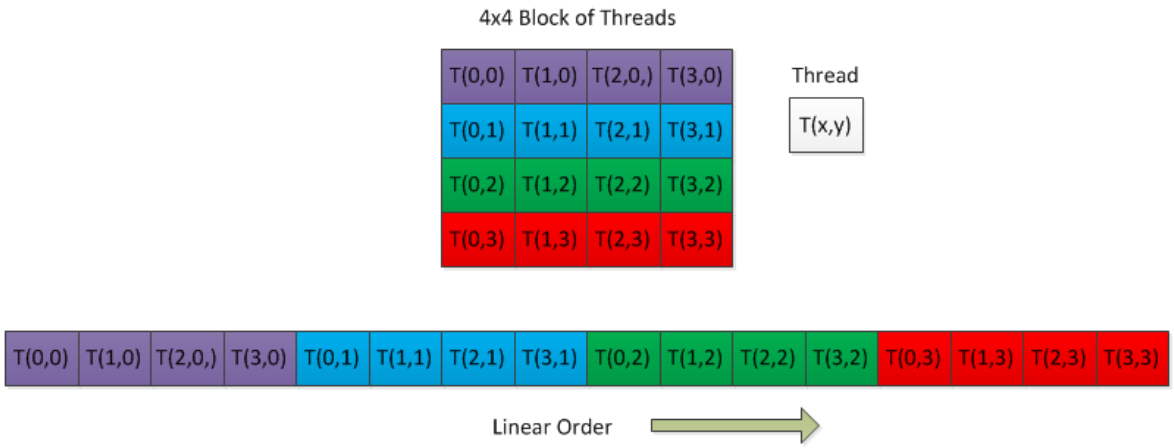


Figure 3: Linear order of threads of a two dimensional block

Depending on the CCAP of the GPU, either one or two warps will be selected at a time from the available blocks assigned to the SM. Once the threads in a warp encounter a read or write to global memory (long latency off-chip GPU memory), the warp is swapped with another available warp to hide the long latency. If no other warp is available, then the SM will be idle. The more warps available to an SM, the better the latency will be hidden [4] [1] [5].

The blocks in a grid are assigned to SMs. There is a limit to the number of threads and blocks that can be assigned to a single SM. For example, the GTX 295 can have a maximum of 8 resident blocks per SM and a maximum of 1024 (32 warps) resident threads per SM. Moreover, each block has a maximum size of 512 threads. For example, a block size of 256 will yield a maximum of 4 blocks per SM and a maximum of 1024 threads per SM. Since the maximum number of threads is reached, no more blocks can be assigned to the SM. As another example, a block size of 64 will yield a maximum of 8 blocks per SM with a maximum of 512 threads per SM [1]. Again since the maximum number of blocks is reached, no more threads can be assigned to the SM.

2.1.3 Streams

All CUDA operations are placed in a CUDA stream. A CUDA operation is an operation that executes on a CUDA enabled GPU. For example, memory copies and kernel calls are common CUDA operations. A stream is a “sequence of [CUDA]

operations that execute in issue-order on the device" [6]. When no stream is specified, the null or default stream is used. However, the programmer can create additional streams to execute CUDA operations. Depending on the CCAP of the GPU, streams can allow the overlap of computation and communication and can allow up to 16 concurrent kernel calls to execute on the GPU at the same time. If the hardware supports the overlap of computation and communication, some prerequisites must be satisfied. The memory on the host must be allocated as *page locked (*the operating System cannot swap out the page to virtual memory). In order for CUDA to execute kernels concurrently there have to be hardware resources available for the additional kernels to run at the same time and a stream for each kernel. This feature will work best with several lightweight kernels that do not saturate the hardware resources [1].

Default stream

CUDA operations executed in the default stream are executed as if *cudaDeviceSynchronize* is called before and after each call. This function will force the host to wait until all preceding commands in all streams of all host threads have completed [1]. In short, these operations are synchronous with respect to the host and the device. However, there are exceptions to this rule. These exceptions are asynchronous with respect to the host and are outlined below [6]:

- Kernel launches in the default stream
- *cudaMemcpyAsync*
- *cudaMemsetAsync*
- *cudaMemcpy* within same device
- Host to device *cudaMemcpy* of 64kB or less

This asynchrony allows the host to invoke a kernel and then return immediately and execute a host function. The host function and the kernel will execute concurrently.

Synchronization

There are numerous methods in performing explicit synchronization with streams in CUDA. However, there are CUDA operations that implicitly synchronize all other CUDA operations. These operations (outlined below) will force synchronous behavior when one of these is issued between two commands from different streams [6] [1].

- Page-locked memory allocation
- Device memory allocation
- Device memory set
- Device ←→ device memory copy
- Non-synchronous version of memory operations
- Change to L1/shared memory configuration (CCAP 2.x)
- Any CUDA operation to the default stream

Stream Scheduling

When CUDA operations are scheduled to execute on the device, they are placed in one of three engine queues (assuming CCAP 2.x). There is one queue for executing kernels, one for copying memory from the host to the device, and another for copying memory the reverse direction. CUDA operations are dequeued from an engine queue and executed on the device if the following conditions are all true [6]:

- Preceding calls in the same stream have completed
- Preceding calls in the same queue have been dispatched to the hardware
    - Regardless of whether the operation has finished
- Resources are available on the device

To help explain when kernels will execute concurrently, take this simple example. There are two kernels scheduled to execute on the device. Each kernel was scheduled with a different non-null stream. When the first kernel is dispatched to the hardware, the second kernel will also be dispatched if the following conditions are true: all the thread-blocks of the previously dispatched kernel have been scheduled to SMs and there are still SM resources available [6].

CUDA operations are dispatched to the hardware in the order they were scheduled. This fact can adversely affect how effectively CUDA operations run concurrently in the device. If CUDA operations are not scheduled in the correct order, an operation could block all other operations in the same queue no matter if they were scheduled with different streams [6].

An example of scheduling kernels using multiple streams is shown in Figure 4. This example demonstrates how the order in which kernels are scheduled can greatly impact the performance. Stream 1 contains the following kernels (the number in the parentheses is the amount of time the kernels take executing): *Ka1(2)* and *Kb1(1)*. Stream

2 contains the following kernels: *Kc2(1)* and *Kd2(2)*. All kernels fill half of the SM resources. Therefore, any two kernels can execute concurrently without any problems. In each of the scenarios, the order in which the host thread schedules kernels to execute on the device is designated by "compute queue."

The scenario on the left will be discussed first. As illustrated by the diagram, all the kernels in stream 1 are scheduled and then all the kernels in stream 2 are scheduled. Since *Ka1* was scheduled first, this kernel is the first kernel that will be dispatched on the hardware. Once *Ka1* finishes execution, the next kernel that can be dispatched is *Kb1*. Since this kernel is within the same stream, it cannot be executed concurrently with *Ka1*. The next kernel scheduled was *Kc2*. Since this kernel is in a different stream it can be run concurrently with *Kb1*. The last kernel scheduled is the last kernel dispatched. This kernel must wait until *Kc2* finishes execution since they reside in the same stream.

The next scenario is the middle case. This case makes an improvement in the scheduling order of kernels, indicted by the arrows. Again kernels are dispatched in the order they were scheduled. The first kernel scheduled is *Ka1* and therefore is dispatched first. The second kernel scheduled is *Kc2*. Since this kernel is in a different stream it can be run concurrently with *Ka1*. The next kernel scheduled was *Kb1*. It is dispatched on the hardware after *Ka1* finishes execution since they reside in the same streams. Afterward, *Kd2* is dispatched while *Kb1* is executing because they reside in different streams. Although this example improves upon the first scenario (right), there is still room for improvement. Notice how there is space between *Kc2* and *Kd2*.

The last example makes another small adjustment in the scheduling order of kernels. This adjustment in the order allows *Kd2* to be dispatched right after *Kc2* finishes execution, thereby removing the idle time in the middle scenario.

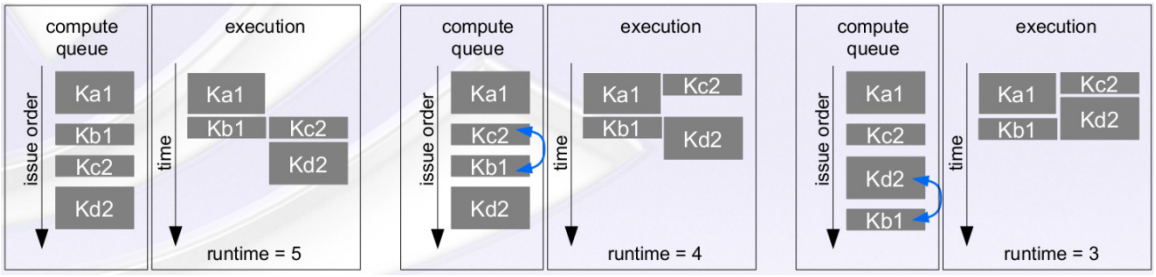


Figure 4: How the issue order can affect execution time of kernels in multiple streams. Left is depth first order, middle is breadth first order, and right is a custom order [6].

2.1.4 Memory

CUDA offers several types of memory to the programmer. The first and most abundant is global memory. Since it is off-chip, the memory will have long latency but high bandwidth. It can be accessed by all threads and persists between kernel calls. Global memory is only cached on devices with CCAP 2.x and above. Texture, constant, and local memory all reside in global memory but have different properties. For example, texture memory is cached. It is optimized for 2D spatial locality and helps reduce global memory bandwidth. Constant memory is also cached. It is read only and helps to reduce global memory bandwidth. Local memory is used for register spilling. For instance, if a kernel uses more registers than the hardware supports, then local memory is used. Next, shared memory is an on-chip memory with very fast access times. However, threads within the same block can only share data. There is a very limited amount available per SM. Register memory is the last memory available. It is the fastest but the least abundant memory. Every thread has its own private set of registers allocated to it [1]. Device memory is summarized in Table 1.

Table 1: Memory in CUDA (*Cached only on devices of compute capability 2.x) [7]

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	*	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	*	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

Memory coalescing is an important feature in CUDA. It allows a series of global memory requests to be lumped or coalesced into a single global memory request. The CCAP of the device will determine the requirements for memory coalescing. Devices

with CCAP 2.x will have more relaxed requirements for memory coalescing than with devices with CCAP 1.x. In general, the memory accesses of threads will be coalesced if consecutive threads access consecutive memory addresses. The most threads that can be coalesced together will be a half warp or 16 threads, shown in Figure 5. Memory accesses will not be coalesced if consecutive threads access nonconsecutive memory addresses [1].

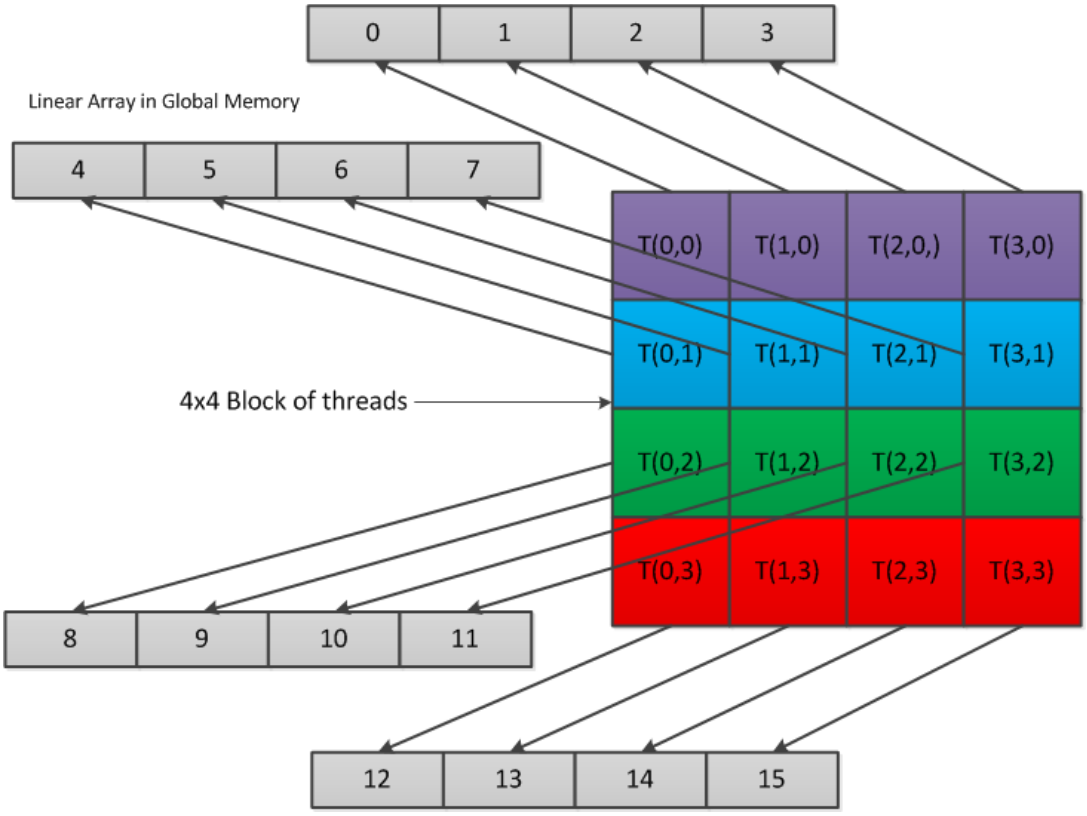


Figure 5: A coalesced memory request for a half-warp block of threads

Knowing when to use shared memory is very important to achieve high performance. Shared memory is useful when threads in a block access the same data more than once. Also, it can be used to avoid uncoalesced memory accesses from global memory. First, the data from global memory is loaded into shared memory using coalescing before any main computation. Next, half warps can access the data in shared memory without any penalty for non-sequential or unaligned accesses. Lastly, the data from shared memory is stored back in global memory using coalescence [7].

Knowing how to use shared memory is also important in achieving high performance. Shared memory has 16 banks for CCAP of 1.x and 32 banks for CCAP of

2.x. Each bank can hold four bytes of data. Multiple banks are used because each bank can be accessed in parallel with one another. Since each bank has a bandwidth of four bytes per two clock cycles, shared memory has a bandwidth that is equal to the number of banks times the bandwidth of a single bank. For CCAP 1.x, a warp accessing shared memory with no bank conflicts is split into two accesses, one for each half warp. For CCAP 2.x, a warp accessing shared memory with no bank conflicts is not split into multiple requests since there are 32 banks rather than 16. However, the number of shared memory requests can increase when there are bank conflicts. A bank conflict occurs when "two or more threads access any bytes within different 32-bit words belonging to the same bank [1]." When a conflict occurs, the hardware splits the memory request into as many separate conflict-free memory requests as necessary. Each additional memory request reduces the effective bandwidth [1] [7].

To help illustrate when bank conflicts occur, take the linear addressing patterns in Figure 6 as examples. The access pattern on the left and the right produce no bank conflicts because each thread is accessing a different bank. The middle example has bank conflicts because there are two threads accessing two different words in a single bank.

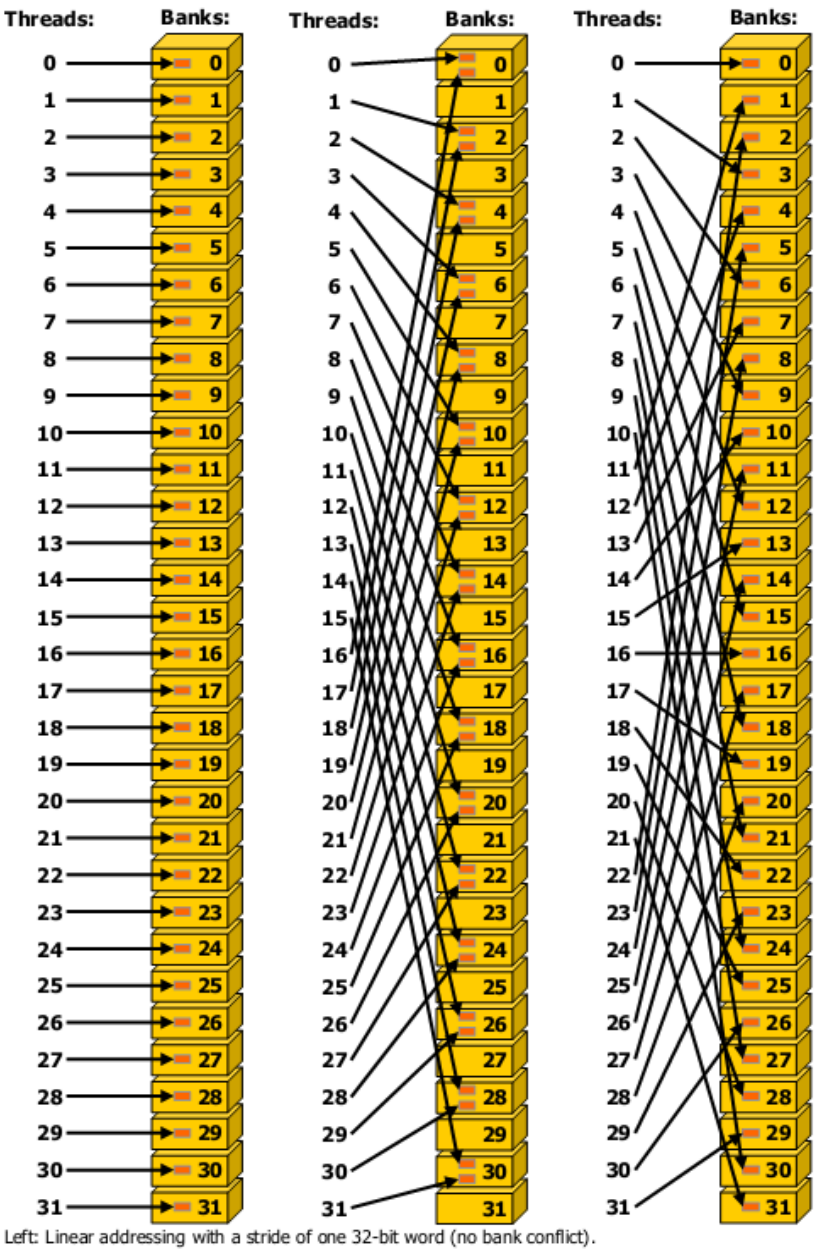
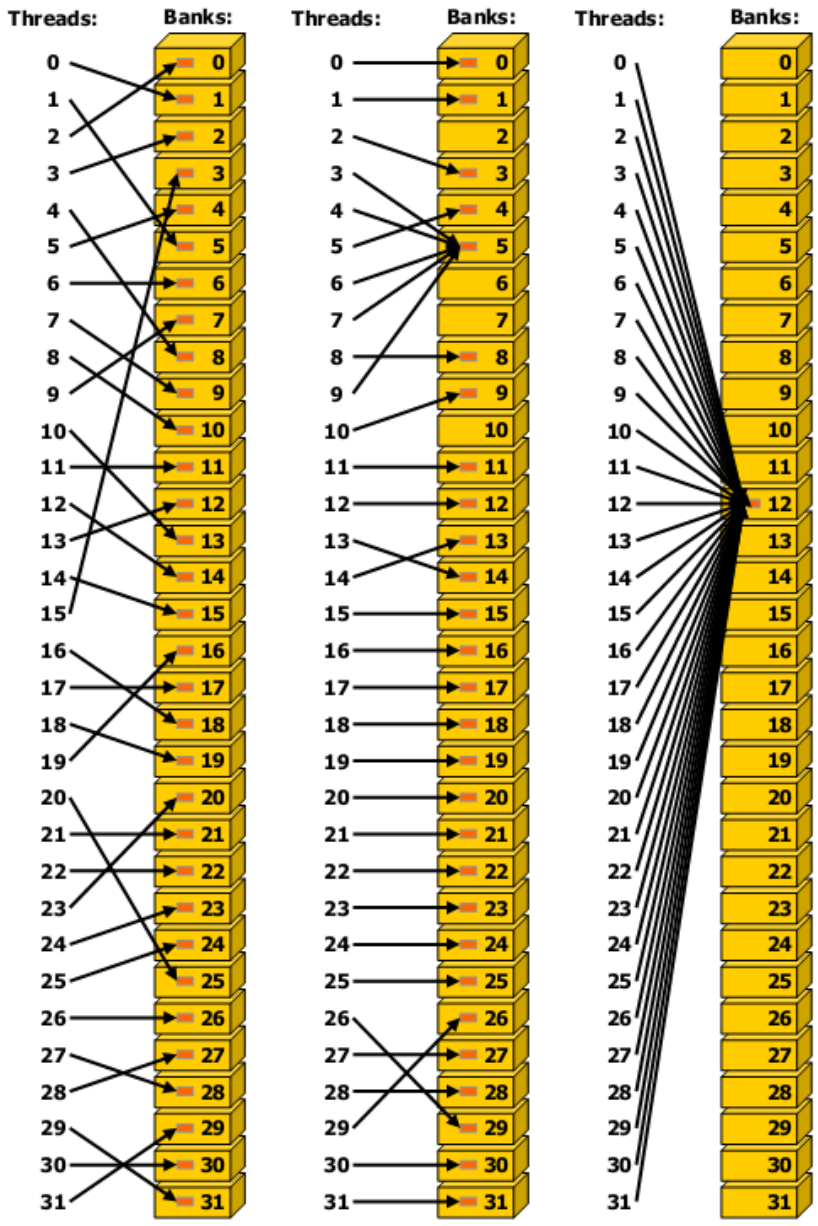


Figure 6: Examples of strided shared memory access (CCAP 2.x) [1]

The next example shown in Figure 7 below represents three random access patterns. The left example produces no bank conflicts because each thread is accessing a different bank. In contrast, the right example produces no conflicts because multiple threads are accessing the same word in the same bank. This situation results in a broadcast to the threads in the warp. The middle example combines the left and right scenarios; therefore there are no bank conflicts in this scenario either.



Left: Conflict-free access via random permutation.
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.
Right: Conflict-free broadcast access (all threads access the same word).

Figure 7: Examples of irregular and colliding shared memory accesses (CCAP 2.x) [1]

Using shared memory with the type *double* is a special case. A *double* is 8 bytes, but a shared memory bank can hold only 4 bytes. For devices of CCAP 1.x, a bank conflict occurs only "if two or more threads in either of the half-warps access different addresses belonging to the same bank." For CCAP 2.x, there will be no bank conflicts if the thread id is used as the index into the shared memory array [1].

2.1.5 Single-Precision vs. Double-precision Arithmetic

Single-precision arithmetic achieves about thrice the floating-point operations per second (FLOPS) as double-precision arithmetic, shown in Figure 8. Therefore, it is crucial to use single-precision arithmetic whenever possible. However, some applications require the use of double-precision because of the magnitude of the numbers and/or the computations done on the numbers. For this reason, the Tesla line of GPUs was designed to have much faster double-precision performance than the GeForce cards to accelerate applications such as linear algebra, numerical simulation, and quantum chemistry [7].

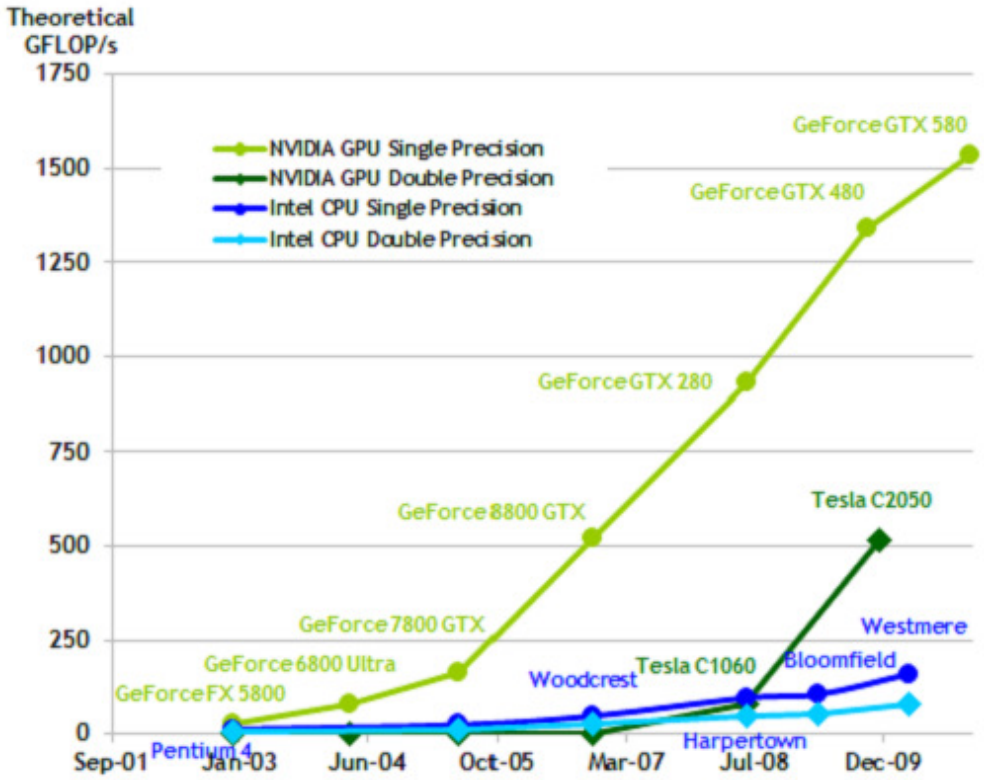


Figure 8: Floating-point operations per second (FLOPS) for CPUs and GPUs [7]

2.1.6 Measuring Performance

Kernels can be classified as one of two types: compute or bandwidth bound. Compute bound kernels have their performance limited by the number of FLOPS that can be executed in the device. On the other hand, bandwidth bound kernels have their performance limited by the available memory bandwidth on the device. Global memory bandwidth is usually the memory that is the limiting factor.

One way to quantify the effect of memory access efficiency in CUDA kernels is called the Compute to Global Memory Access (CGMA) ratio. This ratio is the number of floating-point operations per access to global memory in a particular region of a CUDA program. The throughput of floating-point operations is determined by two main factors: the speed of the ALUs and the rate at which data can be read from memory to feed the ALUs. The CGMA ratio, depicted in Equation 1, helps to determine if a kernel is compute or memory bound [4]. N_{FPO} is the number of floating point operations. N_{AGM} is the number of accesses to global memory. A higher ratio means better performance.

$$CGMA\ ratio = \frac{N_{FPO}}{\mathrm{N_{AGM}}} \qquad \text{Equation 1}$$

If the CUDA code is bandwidth limited, the maximum possible FLOPS can be found using Equation 2. Since Equation 2 depends only on a particular device's memory bandwidth, this calculation ignores all other overheads. GM_{BW} is the global memory bandwidth. D_{size} is the size of the datum (4 bytes for single-precision and 8 bytes for double-precision).

$$\max(FLOPS) = N_{FPO} * \frac{\mathrm{GM_{BW}}}{\mathrm{N_{AGM}} * \mathrm{D_{size}}} \qquad \text{Equation 2}$$

2.2. The Human Heart

The heart is composed of three layers: the epicardium, myocardium, and endocardium. The outside layer is called the epicardium. The middle layer is named the myocardium. This layer is where the cardiac muscle resides. The branching cardiac muscles are connected to each other by crisscrossing connective tissue fibers. These connective fibers do not carry electric current and therefore help determine the electrical pathways in the heart. The third and last layer is called the endocardium. It lines the valves and chambers of the heart [8].

The electrical System of the heart is governed by the coordinated flow of ions through the cardiac cells. When these ions move from cell to cell, a potential difference is created across the cell membrane. This potential difference is known as the cardiac TransMembrane Potential (TMP) or action potential. At rest the TMP is about -90 mV. During contraction it is about 30 mV [8] [9]. Action potentials are generated by noncontractile cardiac cells called autorhythmic cells. One such group of autorhythmic cells is the sinoatrial (SA) node or pacemaker. The SA node produces the dominating action potential and therefore paces the heart's contractions. The Purkinje fibers and the atrioventricular bundle also produce electrical signals that can pace the heart, but their slower rate cannot dominate the faster rate of the pacemaker unless it becomes dysfunctional [8].

The cardiac conduction System shown in Figure 9 distributes the action potentials from the SA node throughout the heart. The TMPs in the conduction System move at several meters per second compared to 0.3 to 0.5 m/s without the conduction System. The heart's ability to beat rhythmically is due in part to the conduction System. The conduction System allows the cardiac muscles to contract in unison. Otherwise, the cardiac muscles will contract at different times resulting in arrhythmias [8].

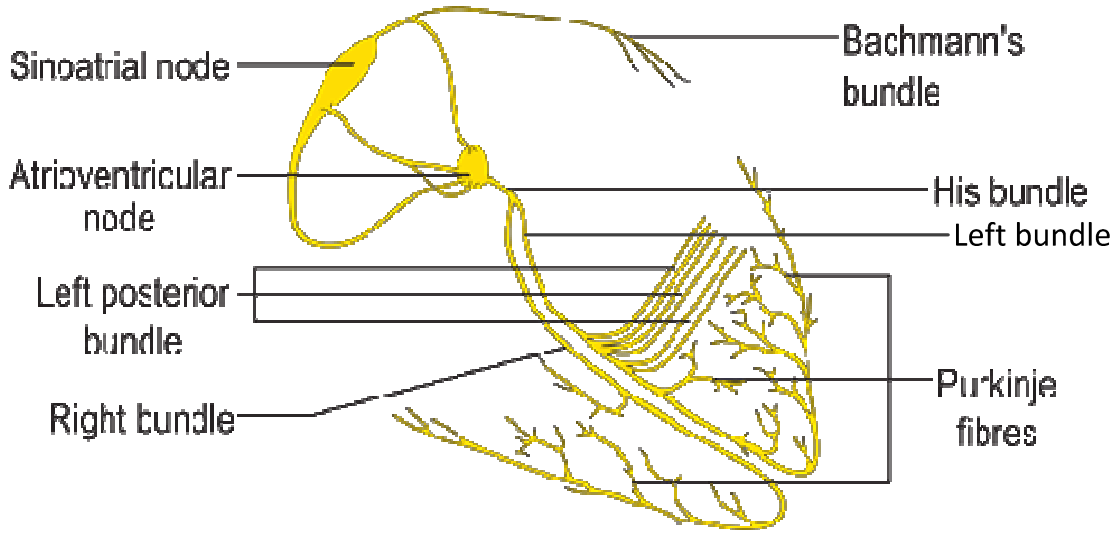


Figure 9: Conduction system of the heart [10]

2.3. Electrocardiography

The TMPs produced by the heart can be detected with an electrocardiograph on the body surface. This tool produces a graphic record of electrical activity called an ECG or EKG (ElectroCardioGram). The ECG represents the superposition of all the voltages produced by the heart [8].

Recording these voltages traditionally requires 12 leads to be placed throughout the body. Two are placed on limbs, and the other 10 are placed on the chest. These leads capture three types of waves: the P wave, QRS complex, and T wave. Together, these waves form an ECG. The waves of an ECG have distinguishable characteristics that help physicians diagnose heart abnormalities. However, the information in an ECG is limited. Therefore, ECGs can provide only limited diagnostic information [8].

2.4. The Inverse Problem of Electrocardiography (IPECG)

The framework for the NTEPI algorithm is based upon the IPECG. The IPECG aims to reproduce the heart's electrical activity from body surface potentials (BSPs) [11]. The BSPs are acquired using several hundred leads compared to the traditional ECG that uses 12 leads. An example apparatus for placing the electrodes on the body is illustrated in Figure 10. Previous work [11] has rendered action potentials on the surface of the myocardium. However, NTEPI has taken the research further by reproducing the electrical activity within the myocardium. This new information can significantly help diagnose problems that are within the myocardium rather than just on the surface [12].

Figure 10: A multielectrode vest for recording BSPs [11]

2.5. Noninvasive Transmural Electrophysiological Imaging (NETPI)

NTEPI uses two key pieces of information to solve the IPECG: personal BSP data and general electrophysiological activity in the myocardium. However, this information is plagued by uncertainties. To solve the IPECG accurately, the data uncertainties must be taken into account using a statistical perspective [12].

NTEPI was originally implemented in C/C++. The program relies heavily on Matlab for much of its computations, specifically for matrix inversion and Cholesky factorization. NTEPI is an iterative algorithm. The NTEPI iteration is composed of five main methods: *samGen*, *samProRKA*, *upd*, *updLinear*, and *updateV*. These methods are executed every iteration.

The *getCurrent* method updates two data structures for the iteration. *filteringLinear_step* is responsible for invoking the main methods of NTEPI and performing Cholesky decomposition on covariance matrix *Px*. The *samGen* method generates a set of sample vectors that form the matrix *samX* to initiate Monte Carlo simulation. The first step in the algorithm is the prediction step, and it takes place in the methods *samProRKA* and *upd*. No BSP data are used for this computation. The *samProRKA* method performs Monte Carlo simulations on a System of differential

equations. Next, the method *upd* performs probabilistic estimation algebra on large matrices and vectors. The next step in the algorithm is the update step, contained within the *updLinear* method. This step uses the UKF to take BSP data and the samples generated in the prediction step to update the samples. This update computation requires inverting a matrix. Lastly, *updateV* updates a vector *V*. *V* is one of the unknown variables related to electrical activity in NTEPI. Another vector that is related to electrical activity in the heart but is of more medical interest is *U*. Theoretically, whenever *U* is estimated *V* also needs to be estimated. To reduce the amount of computation in NTEPI, *V* is not estimated but updated by finding the mean of all the changed values of *V* during the current iteration.

2.6. Linear Algebra Libraries

Since NTEPI is an algorithm almost completely composed of linear algebraic operations, NTEPI can take advantage of several freely available libraries. This thesis uses two ubiquitous scientific linear algebra libraries: Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK). The BLAS library implements the basic matrix and vector operations. There are three levels of BLAS functions. Level 1 performs scalar, vector, and vector-vector operations. Level 2 performs matrix-vector operations. Lastly, level 3 performs matrix-matrix operations [13]. The LAPACK library uses the BLAS library to compute more complex operations like LU and Cholesky factorization [14].

There are several implementations of BLAS and LAPACK. Nvidia provides a free CUDA implementation of the BLAS library, named Compute Unified BLAS (CUBLAS) [15]. Another library called Matrix Algebra on GPU and Multicore Architecture (MAGMA) implements some of the LAPACK functions by leveraging both the GPU and CPU. MAGMA uses the CUBLAS and LAPACK (runs on the CPU) libraries in addition to its own custom CUDA kernels [16]. The AMD Core Math Library (ACML) is one example of an implementation of BLAS and LAPACK for the CPU. It is free, provides support for multiple threads, and is in a ready to use library file. The downside to this is ACML may not perform the best on Intel processors since it was designed for AMD processors [17]. However, since the time spent executing the

LAPACK routines is very minute compared to the total execution time of the entire algorithm, ACML was chosen. An alternative to ACML is Automatically Tuned Linear Algebra Software (ATLAS). ATLAS provides an implementation of BLAS and some of the LAPACK functions. ATLAS automatically tunes the functions to the architecture of the machine on which it is being compiled [18]. The downside to this alternative is that the library has to be compiled on each machine.

2.7. Data Storage

NTEPI stores matrices in column-major order, meaning, the elements in an entire column are consecutive in memory. For example in Figure 11, elements $A_{0,0}$ and $A_{0,1}$ are not consecutive in memory. They are four elements between them. Figure 11 illustrates the differences. Column-major storage is beneficial because popular linear algebra libraries use this storage technique. Some examples include the following:

- Matlab
- BLAS
- LAPACK
- CUBLAS
- MAGMA

The method elements of a matrix are stored in linear memory becomes very important for designing CUDA kernels that coalesce memory requests. Accessing the elements row-wise of a column-major matrix will result in uncoalesced memory accesses. Uncoalesced memory accesses cause poor performance.

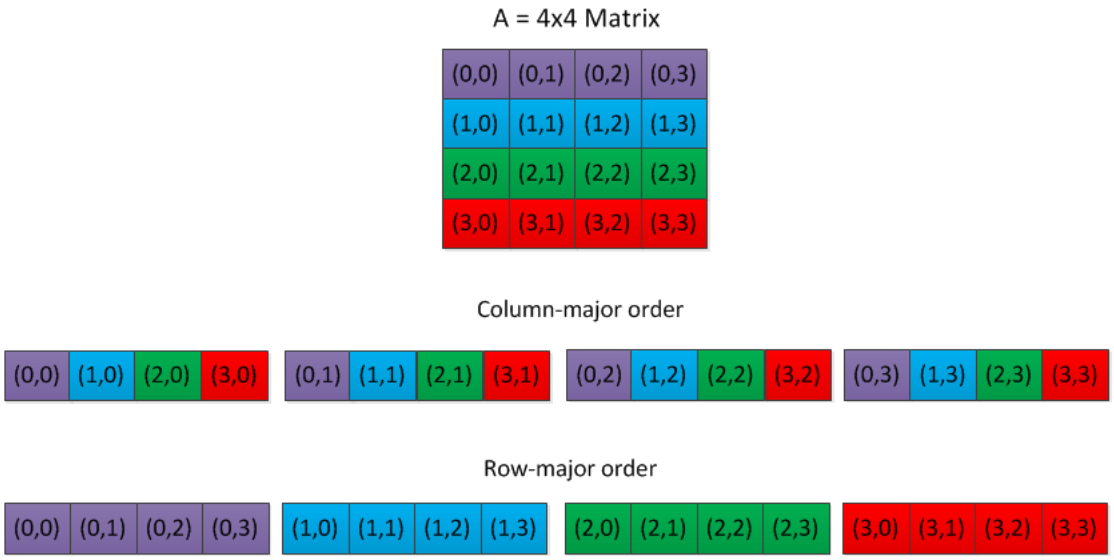


Figure 11: Column and row-major data storage

Chapter 3 Implementation

In this chapter, the hardware used in this thesis is described. Next, the profiling results of the baseline CPU version of NTEPI are presented. Based on these results, the functions with the highest execution times are accelerated with CUDA along with one additional matrix operation.

All tests conducted in this thesis were run on the Systems listed in Tables 2 and 3. Please note that in System C, only one of the GPUs was used in this thesis. The GTX 480 has the best specs except the Tesla C2070 has better double-precision performance and much more DRAM. The GTX 295 is an older card. This GPU supports double-precision but has only one double-precision floating point unit per SM unlike the others that have one in each CUDA core [1]. Although the GTX 295 has more SMs, each SM has only eight CUDA cores. The other GPUs have 32 CUDA cores per SM.

Table 2: The specifications of each host (CPU) used in this thesis

Host Specs	System A	System B	System C
CPU	Intel i7-2600	Intel i5	AMD Anthlon 64 X2 5600+
Cores	4	2	2
Hyper-Threading	Yes	Yes	No
Clock Frequency (GHz)	3.4	3.1	2.9
DRAM (GB)	16	8	4
Operating System	Windows 7 x64	Windows 7 x64	Windows 7 x64
ACML version (multiple processor version)	5.1.0	5.1.0	5.1.0
Matlab version	R2011b x64	R2011b x64	R2011a x64
Visual Studio	2010 Ultimate	2010 Ultimate	2010 Ultimate

Table 3: The devices' specifications in each system used in this thesis

Device Specs (per GPU)	System A	System B	System C
GPU	Tesla C2070	GTX 480	GTX 295 (2 GPUs on 1 card)
CCAP	2.0	2.0	1.3
SMs	14	15	30
CUDA cores	448	480	240
Graphics/Processor Clock (MHz)	575/1150	700/1401	576/1242
Memory Bandwidth (GB/s)	144	177.4	111.9
DRAM (GB)	6	1.5	0.875
Computing SDK version	4.1	4.1	4.1
Driver version	286.16	286.19	286.16

3.1. CPU Profiling

The algorithm was profiled with Microsoft Visual Studio 2010 instrumentation profiler on System C without CUDA. The instrumentation profiler inserts timing code to measure the time each function takes and each function call that is made by those functions [21]. Table 4 shows the results of the profiling. Elapsed inclusive time represents the total time spent executing the function including any functions called by the function. The results show the function *samProRKA* is responsible for the majority of the execution time. All of these functions perform matrix operations and thus should be able to take advantage of CUDA. Specifically, these functions perform the majority of the filtering algorithm. The algorithm takes 285.3 minutes to run (one contraction cycle) using a synthetic experiment. Out of this time, the five filter functions account for 98.06% of the total execution time of the algorithm. Therefore each function is accelerated with CUDA. A real experiment takes about 32 hours on System A to render one contraction cycle.

Table 4: Summary of instrumentation profiling using Microsoft Visual Studio 2010 without CUDA acceleration on a synthetic experiment

Function Name	Elapsed Inclusive Time %
Filter::samProRKA	79.63
Filter::upd	7.74
Filter::updLinear	7.71
Filter::updateV	1.53
Filter::samGen	1.45

Amdahl's law, shown in Equation 3, can be used to determine the maximum speedup obtainable from parallelizing serial code. *P* represents the "fraction of total serial execution time taken by the portion of code that can be parallelized," and *N* refers to the number of processors that will execute the parallel portion of the code [1]. The maximum speedup obtainable is given by the limit of Equation 3 as *N* goes to infinity, as shown in Equation 4.

$$speedup = \frac{1}{(1-P) + \frac{P}{N}} \qquad \text{Equation 3}$$

$$\lim_{N\to\infty} \frac{1}{(1-P) + \frac{P}{N}} = \frac{1}{(1-P)} \qquad \text{Equation 4}$$

Assuming that all 98.06% of the serial code can be parallelized, the maximum speedup using an infinite number of processors is 51.5. However, further analysis of the code reveals that all 98.06% of the code cannot be parallelized. Due to the nature of the algorithm, iterations are dependent on each other. Also, the matrix operations within individual iterations are mostly serial and therefore cannot be executed in parallel. Therefore, a speedup of 51.5 is not possible.

3.2. Implementing Cholesky Factorization

Cholesky factorization is executed prior to the function *samGen.* Although this operation does not represent a high execution time in NTEPI, it was accelerated with CUDA in an attempt to achieve the highest performance.

Cholesky factorization decomposes an $N \times N$ symmetric and positive-definite matrix into two separate matrices. A symmetric matrix is one in which its transpose equal to itself. A positive-definite matrix is a matrix that has all positive eigenvalues. These matrices appear frequently in numerical solutions of partial differential equations. There are two different ways to represent the same factorization: lower (shown in Equation 5) and upper (shown in Equation 6). Matrix L is a lower triangular matrix while matrix U is an upper triangular matrix. NTEPI uses matrix L [22].

$$A = LL^T \qquad \text{Equation 5}$$

$$A = U^T U \qquad \text{Equation 6}$$

Each implementation of the Cholesky factorization discussed in this thesis is based upon the MAGMA implementation, which was based on the LAPACK implementation. LAPACK has two implementations: one blocked and another unblocked. MAGMA has just a blocked implementation. Both implementations are iterative. The unblocked implementation calls level two BLAS [23]. The blocked implementation calls level three BLAS [24]. This implementation is faster for large matrices. It performs the Cholesky factorization block wise (where *nb* is the block size). During the iteration, a block along the matrix's diagonal is factorized by the unblocked Cholesky factorization routine. In addition, there are three calls to level three BLAS functions to aid in the computation of the other elements in the matrix. Table 5 summarizes the functions used in LAPACK's and MAGMA's blocked implementation.

Table 5: The level three BLAS functions that are called by the blocked version of Cholesky factorization and their corresponding matrix operations

Function	Matrix Operations
dsyrk	$C = C - AA^T$
dpotf2 or dpotrf	$A = \text{Cholesky}(\text{subMatrix(A)})$
dgemm	$C = C - AB^T$
dtrsm	Solves the following triangular System with multiple right-hand-sides: $XA^T = B$, where solution X overwrites the right-hand-side B on exit

There were a total of six implementations of Cholesky factorization developed for this thesis and each is analyzed below. Each implementation follows the operations in Table 5.

Hybrid Implementations

The first pair of blocked implementations (designated as version 0 and version 1) uses both the CUBLAS library and the ACML library. Version 0 is MAGMA's version with minor changes. Some changes are a result from using only a fragment of the MAGMA library since the entire library was not compiled. The changes are outlined below:

- The MAGMA version of cublasDgemm was not used.
- The MAGMA version of cublasDtrsm was not used.
- A block size, *nb*, can be passed into the function to override MAGMA's block size.
- There is an option to use LAPACK's blocked/unblocked Cholesky factorization function.
- Upper triangle of output is zeroed out upon exiting.

There are two operations in the iteration that can be executed in parallel: the call to ACML's Cholesky factorization and CUBLAS library's matrix multiplication kernel [22]. Both versions take advantage of this feature. However, version 0 does not overlap communication with computation. The reason is simple: implicit synchronization. MAGMA is launching an asynchronous memory copy from the device to the host in a

non-null stream using *page locked* memory. However, right after this operation a call to the CUBLAS library in the null stream is conditionally made. Any CUDA operation in the null stream synchronizes all CUDA operations. The CUBLAS call cannot start until the memory copy has finished. To avoid this flaw, version 1 was created. In this version, another stream is created to execute the call to the CUBLAS library. This additional stream allows the memory copy and the CUBLAS call to execute simultaneously. Each version was implemented twice, as shown in Table 6.

Table 6: The versions implemented on the CPU/GPU as hybrid Cholesky factorizations

Hybrid Implementations	**Uses LAPACK's unblocked Cholesky factorization routine to compute the factorization of each sub-matrix?**
version 0	Yes
version 1	Yes
version 0	No
version 1	No

CUDA only Implementations

The last two blocked implementations (designated as version 0 and version 1) were originally based on the LAPACK implementation. Further refinement came from MAGMA and [22]. The main difference between these last two implementations is that all the computation is done on the device. An unblocked kernel from [25] was used to compute the Cholesky factorization of a sub-matrix on the device rather than on the host. This avoids transferring data back and forth between the host and device. However, this kernel is limited to launching one block of threads. The order of the block must be less than or equal to 32 for CCAP 2.x since the maximum block size is 1024. Similar to hybrid version 0 mentioned above, CUDA version 0 does not overlap any computation but CUDA version 1 does. This overlap is done using multiple streams. These versions are listed in Table 7.

It is important to note that the original MAGMA implementation of Cholesky factorization used the blocked version of the LAPACK's Cholesky factorization routine.

In contrast, LAPACK's implementation of Cholesky factorization uses LAPACK's unblocked Cholesky factorization routine.

Table 7: The versions implemented with just CUDA

CUDA only Implementations	Uses an unblocked Cholesky factorization CUDA kernel to compute the factorization of each sub-matrix?
Version 0	Yes
Version 1	Yes

Results

In order to measure the performance of all four blocked implementations of Cholesky factorization, a test program was developed. Matlab was used to generate test matrices that were *N x N* symmetric and positive-definite. The Matlab function used was *gallery('gcdmat', nb)*, where *nb* was the order of the matrix. This test program tested how the following aspects affect performance: block size, matrix size, and the use of the blocked and unblocked LAPACK Cholesky factorization function. Each implementation was run 25 times per matrix size per block size from 10 to 128 except where block sizes would produce incorrect results, namely the CUDA only version functions. The best block size was chosen based upon the lowest average execution time. Figure 13 shows the best block sizes used for each matrix size and implementation. The minimum block size of 10 was chosen arbitrary. The maximum block size was chosen because MAGMA chose this block size for matrices with order less than or equal to 4256.

Overall, the performance of each implementation followed the same general pattern, shown in Figure 12. Speedups tended to be inversely proportional to the order of the matrix. Let's first examine the hybrid functions that used LAPACK's blocked implementation of Cholesky factorization. These implementations performed the worst because the blocked version was used rather than the unblocked version. This is evident in Figure 12 because the hybrid implementations that used the unblocked version of LAPACK Cholesky factorization had higher speedups. More precisely, these implementations performed the best. In particular hybrid version 1 using the unblocked Cholesky factorization routine performed the best overall because communication, host computation, and device computation were overlapped with one another when possible.

Interestingly, the two implementations that computed the factorization entirely in the device yielded the highest speedup. However, these implementations were factorizing matrices of order 50. These functions were the fastest in this scenario because there was no communication between the host and device. Factorizing the smaller matrices with hybrid versions resulted in communication that could not be fully overlapped with computation since the computation was less complex. Just like hybrid version 1 performed the best in comparison to hybrid version 0, CUDA version 1 performed the best in comparison to CUDA version 0 because operations were done in parallel using multiple streams when possible. In conclusion, the best implementation for NTEPI was hybrid version 1 since the matrix being factorized is order 2084. The only case where this function is not the best implementation is with very small matrices.

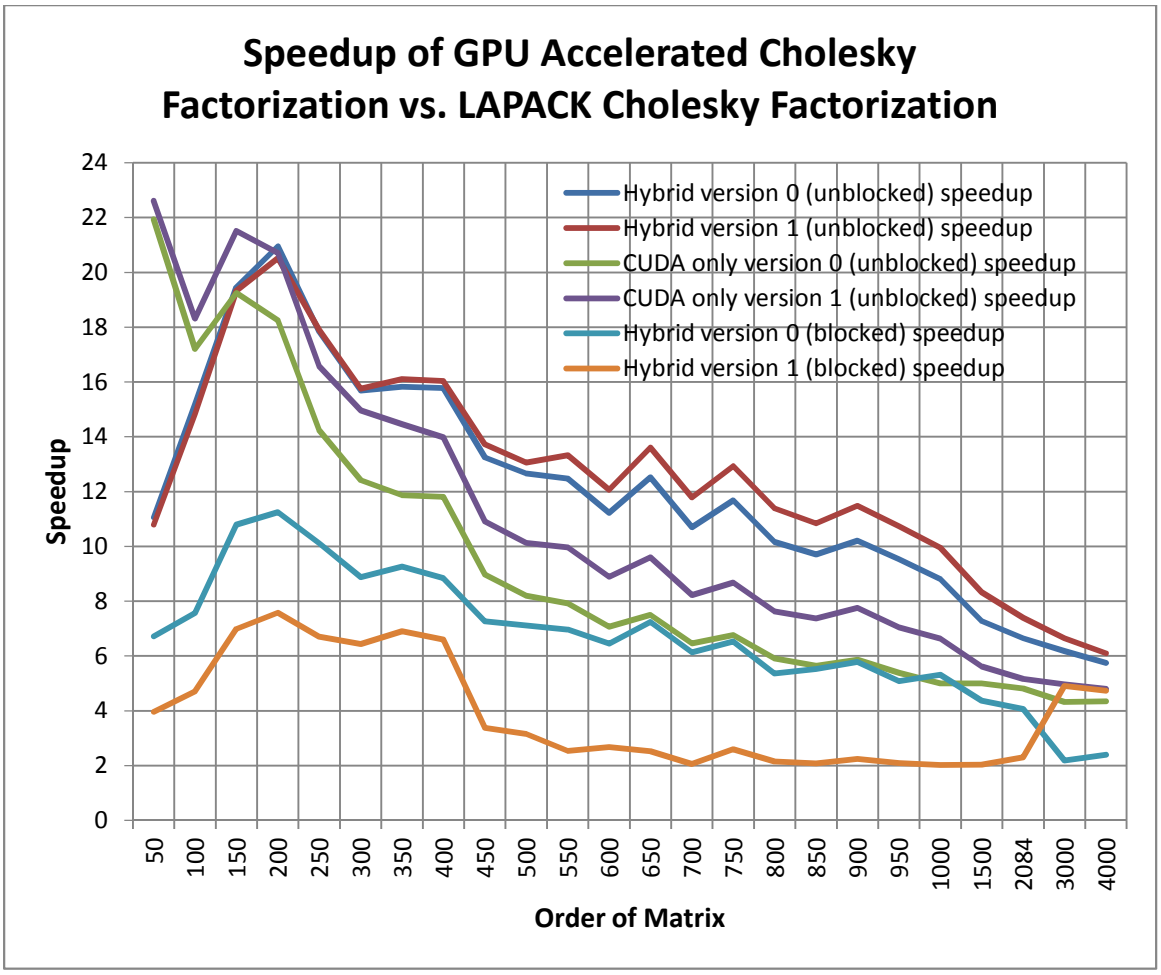


Figure 12: The speedup of all four blocked Cholesky Factorization algorithms implemented compared to ACML's blocked implementation. Each execution time was average over 25 runs on System A. The best block size, *nb*, was chosen by testing *nb* from 10 to 128. The block sizes used are shown in Figure 13.

In general, there were two groups of implementations of Cholesky factorization: one that chose large block sizes and another that chose small block sizes, shown in Figure 13. The implementations that chose large block sizes used LAPACK's unblocked Cholesky factorization function. The hybrid implementations that chose small block sizes used LAPACK's blocked Cholesky factorization function. Therefore, it can be deduced that larger block sizes with LAPACK's blocked Cholesky factorization function take longer. On the other hand, larger block sizes with LAPACK's unblocked Cholesky factorization function were faster than with smaller block sizes. Overall, MAGMA's logic for picking block sizes for matrices smaller than order 4256 was correct for their inefficient implementation.

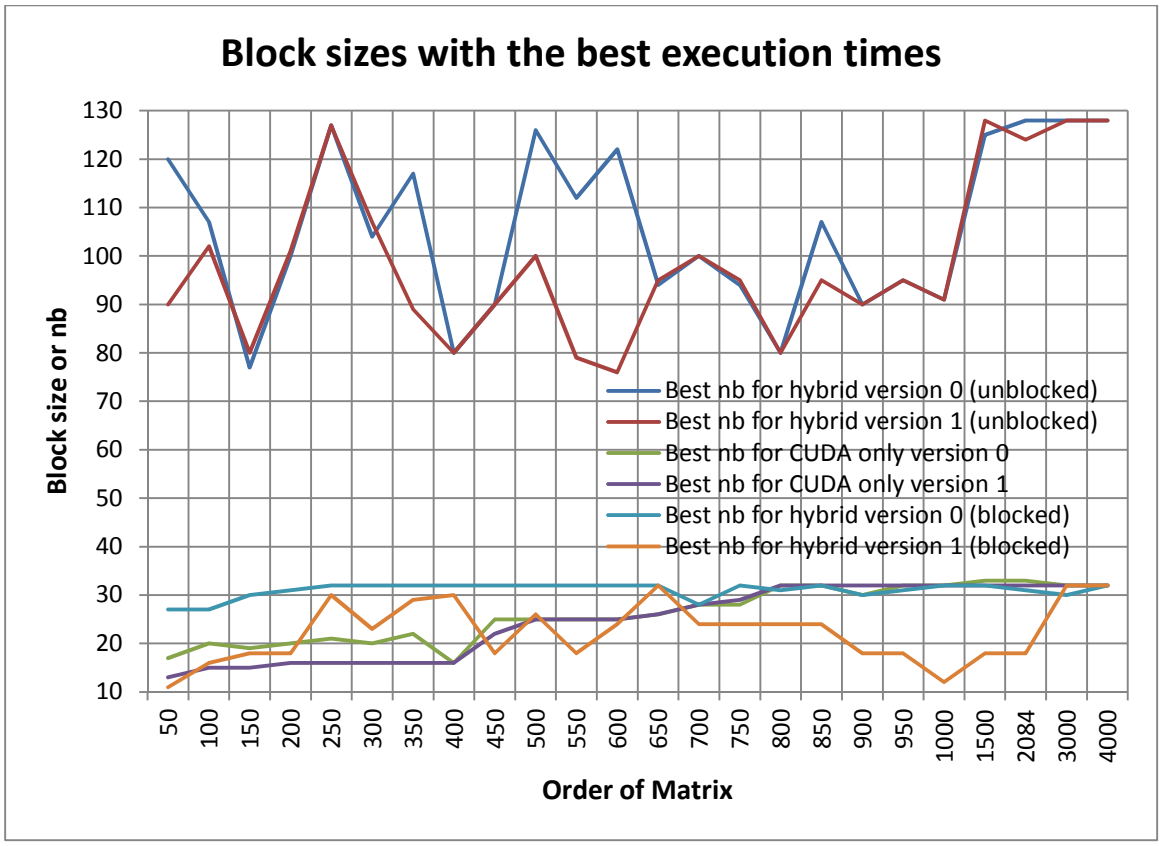


Figure 13: The hybrid versions used the best block size or *nb* out of the following range: 10 to 128. The CUDA only versions used the best block size out of the following range: 10 to 32. Beyond a block size of 32 the method would output incorrect results. The block sizes that are larger than the matrix were not used. Instead, the block size became equal to the order of the matrix.

3.3. Implementing samGen in CUDA

The first main filter method in NTEPI is *samGen*. This method is composed of matrix-vector additions and subtractions. A matrix-vector operation is when a column vector is either added or subtracted in parallel to all the column vectors in a matrix, illustrated in Figure 14. In general, these matrices have thousands of elements. Therefore, CUDA can be leveraged to accelerate these highly data parallel operations. There are three operations carried out in this function. All operations write to matrix *samX*. First, column vector *meanX* is copied to the first column of matrix *samX*. Lastly, sub-matrix *y1* and *y2* are updated by matrix-vector operations. An overview of the operations carried out in this function is illustrated in Figure 15. These operations were implemented several ways.

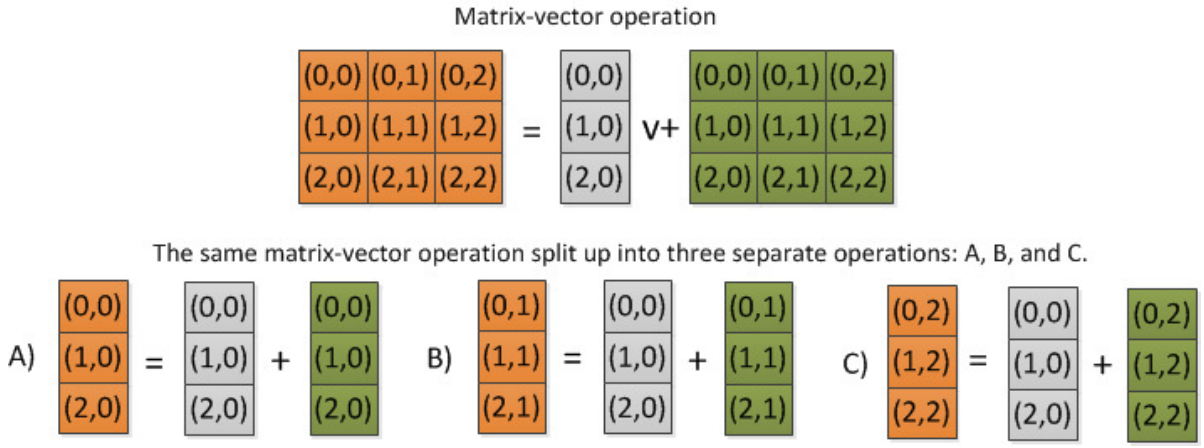


Figure 14: The definition of a matrix-vector operation. Here v+ is matrix-vector addition. Operations A, B, and C can be executed in parallel.

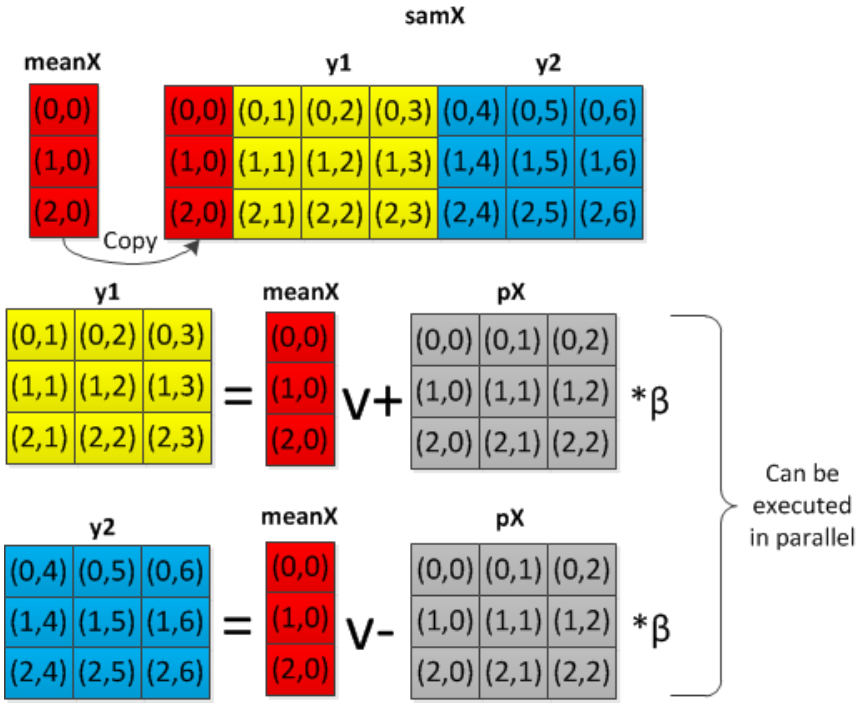


Figure 15: The linear algebra operations in *samGen* where β is a constant scalar and v+ and v- designate a vector-matrix add or subtract. Vector and matrix sizes do not represent actual sizes used.

The first optimization technique of combing multiple kernels together will be presented below.

3.3.1 Combining Multiple Kernels into One Kernel

One optimization strategy used was to combine multiple related kernels into one kernel. Take the following fictitious example in Table 8 below. Computing matrix *C* and *D* in Table 8 requires the following redundant operations:

- B is scaled by r twice
  - B is accessed twice from global memory
- D is accessed 4 times from global memory
- C is accessed 3 times from global memory

These redundant operations translate into a combined CGMA ratio of 5/9. Since one of the floating-point operations is redundant, the CGMA ratio is inflated. Remember a CGMA ratio describes the number of floating-point operations per access to global memory in a particular region of a CUDA program. This means for every 5 floating point operations there are 9 memory accesses.

Table 8: An example of three related kernels operating on matrices *B, C,* and *D*. *r* is a scalar.

Operation	Equivalent CUBLAS Library Kernel Calls	CGMA ratio
C = r*B + C	`CUBLASDaxpy(C.mdeviceData,B.mdeviceData,R,SIZE,1);`	2/3
D = r*B + D	`CUBLASDaxpy(D.mdeviceData,B.mdeviceData,R,SIZE,1);`	2/3
D = D - C	`CUBLASDaxpy(D.mdeviceData,C.mdeviceData,-1.0,SIZE,1);`	1/3

A better way to implement the above operations is to combine the three operations into one custom kernel call, shown in Figure 16. This strategy allows intermediate results to be saved to registers thus reducing floating-point operations and precious memory bandwidth. However, this strategy uses more registers. Using too many registers can reduce the number of active threads in the device. Fewer threads running concurrently in the device could hurt performance. However, executing with fewer threads might be mitigated because the device is being overall more efficiently used. Using a custom kernel rather than three CUBLAS library calls achieved a speedup of 1.81, shown in Table 9. The reason why this custom kernel is faster is outlined below:

- r*B is computed 50% fewer times
- D is accessed 50% fewer times from global memory
- C is accessed 33.3% fewer times from global memory
- CGMA ratio of 4/5

```
__global__ void myCustomKernel(sdfp *B, sdfp *C, sdfp *D, int size) {
	int tid = threadIdx.x + blockIdx.x * blockDim.x;

	if(tid < size) {
		sdfp regC = C[tid];//register C
		sdfp regBscaled = 1.2*B[tid];//scale and register B
		sdfp regD = D[tid];//register D

		regC = regBscaled + regC; //C = r*B + C
		regD = regBscaled + regD; //D = r*B + D
		regD = regD - regC;       //D = D - C

		C[tid] = regC;//write C back to global memory
		D[tid] = regD;//write D back to global memory
	}//if
}//myCustomKernel
```

Figure 16: The three operations in Table 8 implemented with a single CUDA kernel using a block size of 160

Table 9: Performance of custom kernel (Figure 16) vs. multiple related CUBLAS library calls (Table 8) on System A

Implementation	Time (ms)	Speedup
3 CUBLAS library calls	6.21	1.00
1 Custom written kernel	3.42	1.81

3.3.2 Implementations

The first implementation (version 0) of the *samGen* method calculated *y1* and *y2* (Figure 15) with three separate kernels. First, a copy of matrix *pX* was scaled by β with the CUBLAS library. A copy was created because matrix *pX* cannot be modified. Then *y1* and *y2* were each calculated with a custom kernel. This strategy needed to read the column vector *meanX* and the matrix *pX* from global memory twice. The CGMA ratio for the kernels calculating *y1* and *y2* is 1/3. A CGMA ratio of 1/3 means for every floating point operation there are three global memory access. The maximum FLOPS can be estimated using Equation 2. For example, on System A these kernels can achieve a maximum of 6 GFLOPS in double-precision, compared to the peak of 515 GFLOPS available on System A [26].

The second implementation (version 1) fixed the pitfalls from the previous one. Rather than using three separate kernels, a custom kernel was used to carry out all the operations in Figure 15 except the memory copy. This is an example of combing multiple related kernels into one kernel. This kernel had a CGMA ratio of ¾. For every three floating-point operations there were four accesses to global memory. For example, on System A the said kernel can achieve a maximum of 13.5 GFLOPS.

The third implementation (version 2) is based on the previous implementation. This implementation tries to accelerate the matrix-vector operations using shared memory. Remember that a matrix-vector operation, shown in Figure 14, adds or subtracts a column vector to every column in a matrix. As a result, each element in the column vector will be referenced as many times as there are columns in the matrix. Instead of reading the same values from global memory every time an addition or subtraction is performed, the values in the column vector can be stored in shared memory to be reused.

To implement this concept in CUDA, the matrix-vector operation must be transformed into a blocked algorithm, shown in Figure 17. Each block of threads will carry out a piece of the matrix-vector operation. Within this block of threads, a column of threads will store the necessary elements (denoted by "Block of column vector") in shared memory to be reused by all the threads in the block. Since only a column of threads is reading from global memory and writing to shared memory, there can be thread divergence if the column of threads is less than the warp size. Once the data are stored in shared memory, all the threads in the block carry out the matrix-vector addition or subtraction (Figure 14) by first reading data from shared memory and then from matrix *A*. In short, this algorithm reduces the number of accesses to global memory. The threads within a block are working together by sharing data.

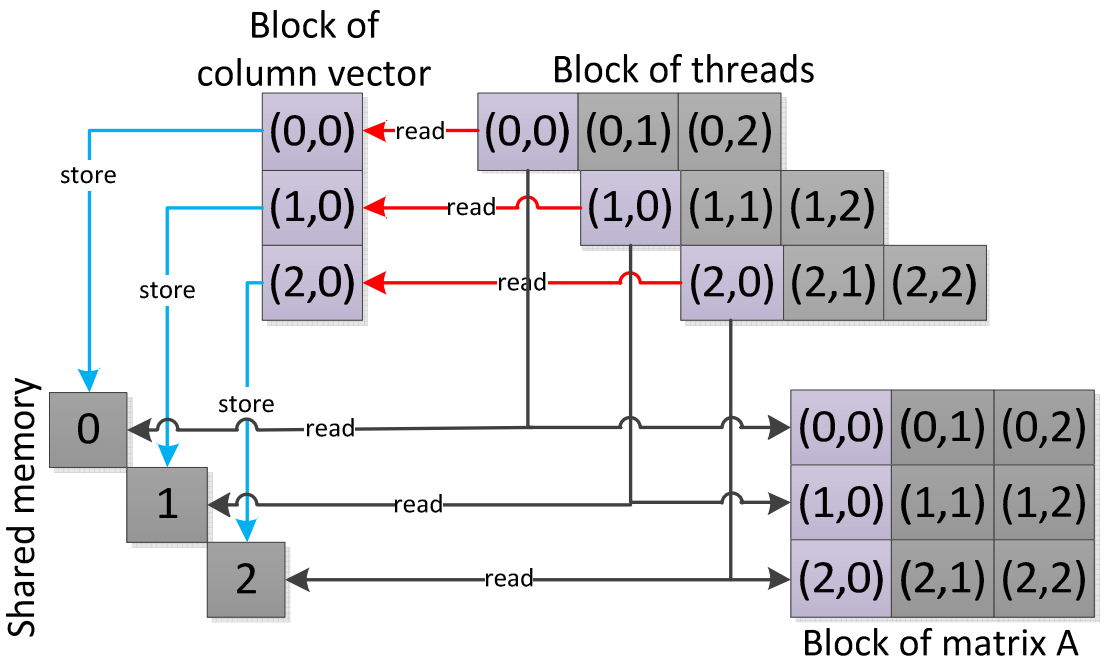


Figure 17: How version 2 of *samGen* uses shared memory to compute the matrix-vector operation

3.3.3 Results

Comparing the maximum attainable FLOPS, version 1 should be about 2.25 times faster than version 0. In reality, version 1 was about 3.17 times faster than version 0, as shown in Table 10. The use of shared memory in version 2 did not yield better performance than version 1. Despite poor performance, version 2 still computed the correct results. Possible reasons for the reduction in performance could be thread divergence and/or the computation of additional thread variables to implement the algorithm in CUDA.

Table 10: The results of each CUDA implementation of *samGen* on System A executed 25 times.

Implementation	Average Execution time (ms)	Speedup
CUDA version 0	5.16	24.35
CUDA version 1	1.63	77.17
CUDA version 2	1.73	72.68
CPU	125.64	1.00

3.4. Implementing samProRKA in CUDA

The second method implemented in NTEPI was *samProRKA*. This function is illustrated in Figure 18 below. Remember this function was responsible for 79.63% of NTEPI's total execution time, shown in Table 4. On entrance, copies of the matrices *samX* and *samV* are created. They are used for computing updated versions of *samX* and *samV*. Next, a series of matrix operations are encountered in a loop. Lastly, the function checks to see if any elements in the two matrices contain any invalid values. Invalid values are found by finding the element with the largest magnitude and comparing it to a maximum value. If none are detected, control returns to the function caller. However, if invalid values are detected, the function reverts to the original values of the matrices by using the copies mentioned above and exits to try a different step size. Over the course of optimizing NTEPI, three distinct C++ wrapper functions were created to implement this function in CUDA.

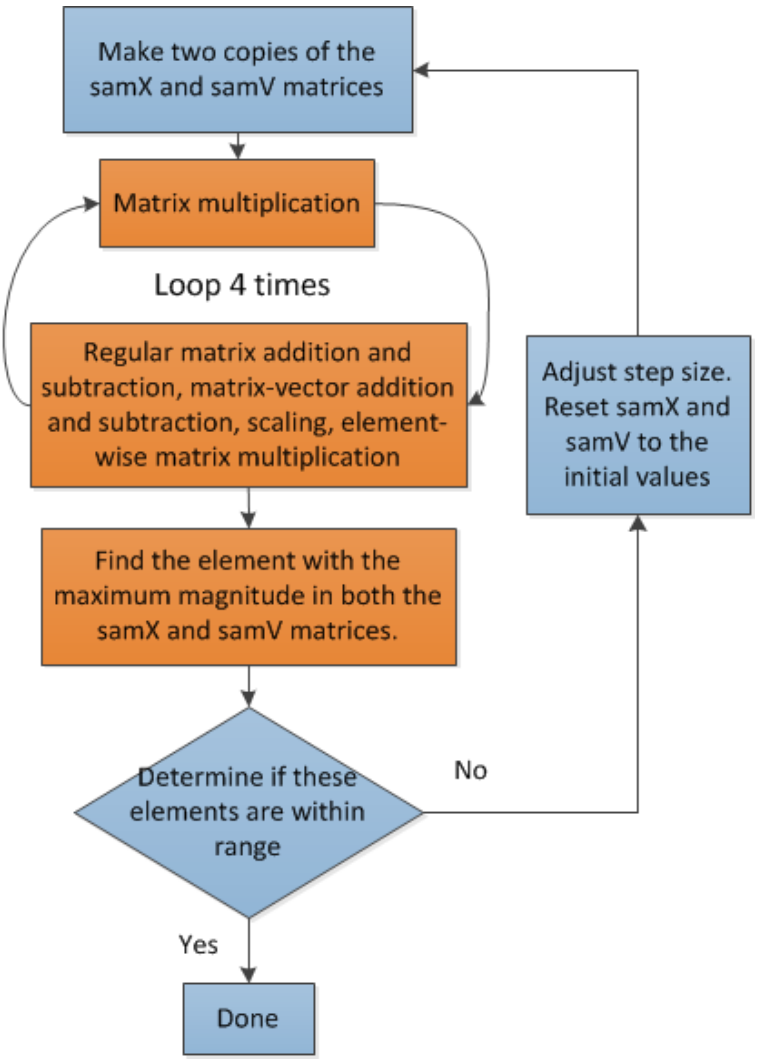


Figure 18: Summary of the operations conducted in *samProRKA*

The first version (version 0) of this function consisted of calling a kernel for each matrix operation. Each call to this version would call a total of 86 kernels. Also, streams were not used to overlap kernel execution. Both custom and CUBLAS library kernels were used to implement this function.

The second version (version 1) grouped related kernels together into multiple kernels. 86 kernel calls were reduced to 18 kernel calls, a 79.07% reduction in kernel calls. Streams were again not used to overlap kernel execution.

This last version (version 2) built upon the previous version by adding streams to overlap kernel execution on devices with CCAP 2.x. There are numerous instances where two kernels can be executed independently of each other. Streams allow these kernels to be executed in parallel in the device if resources are available.

3.4.1 Results

Table 11 below summarizes the findings for each version implemented in CUDA. Version 0 was about 21.20% slower than version 1; meaning grouping related kernels together helped to improve the speedup. Streams improved the performance in version 2 by 0.15%.

Table 11: The results of each CUDA implementation of *cudaSamProRKA* on System A executed 25 times each

Function	Average Execution time (ms)	Speedup
CPU	10054.00	1.00
CUDA version 0	630.52	15.95
CUDA version 1	496.73	20.24
CUDA version 2	496.08	20.27

The results above show the device must have very few resources available to execute kernels concurrently. In Figure 19, the kernel *scaleAddKernel* is executed twice; once for the *samX* matrix and once for the *samV* matrix. Each of these calls can be executed in parallel. Therefore, an instance of *scaleAddKernel* is asynchronously launched in stream 8 and another in stream 9. However, there is extremely little overlap. The reason for this behavior is simple. Looking at the kernel launch parameters shows that *scaleAddKernel* creates 33,939 blocks of 256 threads each. System A has 14 SMs, and only 6 blocks can be allocated to each SM at a given time for this block size. Out of 33,939 blocks that need to execute on the device, only 84 can be assigned to the SMs at a time. Another kernel will not run on the device until it has nearly finished. This is why there is a slight improvement. When the device is finishing a kernel's last few blocks, there will usually not be enough blocks to occupy fully the device's resources. Therefore, the device will launch the other kernel in the other stream to occupy the device fully, shown in Figure 20. Kernels are overlapped only when another kernel is underutilizing the device.

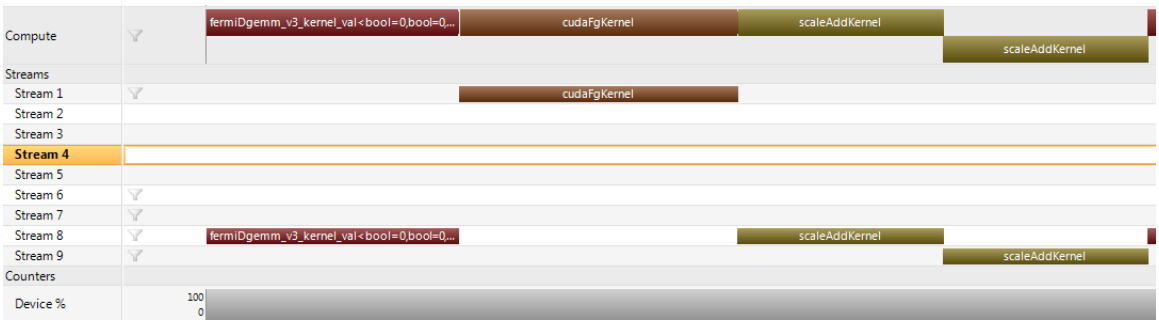


Figure 19: The profiling results of *cudaSamProRKA_v2* from Parallel Nsight 2.1 on System A. The Compute row summarizes how the kernels executed in the device. The Streams row shows what kernels executed in what stream. Lastly, device % is the percent of time a kernel executed on the device.

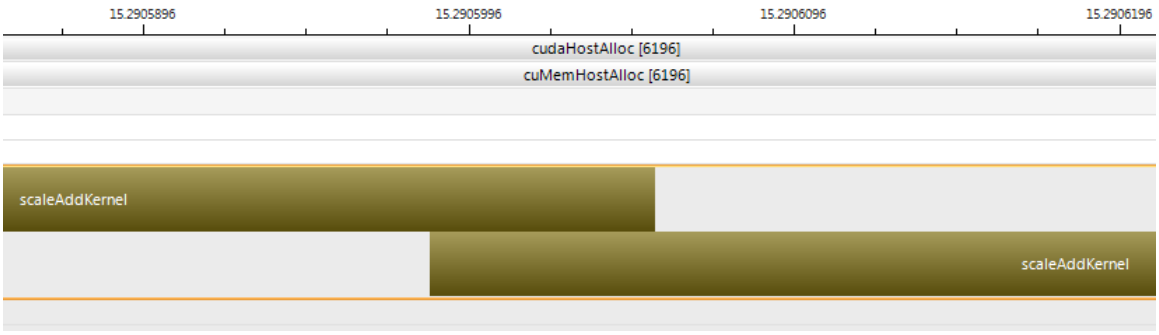


Figure 20: A zoomed in version of two kernels overlapping execution in Figure 19. The kernels overlap for about 600 ns. This represents about 0.0173% of the kernels average execution time.

In addition, the speedup increase due to grouping related kernels can now be fully explained. A speedup increase of 21.20% is quite significant. Excluding matrix multiplication from the computation reveals the significant speedup obtained by combing related kernels. Combing the related kernels netted a speedup of about 3.2, shown in Table 12.

Table 12: The speedup relative to CUDA version 0 on System A when matrix multiplication is removed is shown. Since each matrix multiplication averages 109 ms (Table 22) and is called 4 times, 4*109 ms was subtracted out from the execution times obtained in Table 11.

Function	Execution time (ms)	Speedup
CUDA version 0	194.52	1.00
CUDA version 1	60.73	3.20
CUDA version 2	60.08	3.24

3.5. Implementing Matrix Reduction with CUDA

In this thesis, four implementations of matrix reduction were explored. Matrix reduction is used in the filter function *up*. Each implementation was designed using a

C++ wrapper function around kernels calls. These four implementations can be split into two different strategies: one based upon the CUBLAS library and the other based upon parallel reduction. Each function can reduce any matrix but with varying degrees of performance.

3.5.1 Matrix Reduction via Parallel Reduction

Parallel reduction is a common data parallel primitive. Functionally, reduction sums all the elements of an array together to a single value. Reduction algorithms in the NVIDIA GPU Computing SDK 4.0 were used to form a modified reduction algorithm for this thesis. Reduction was one of the ways matrices in NTEPI were reduced into column vectors, as illustrated in Figure 21. It is similar to reducing a single vector except there are as many vectors as there are rows in the matrix. And the number of elements in each vector is determined by the number of columns in the matrix.

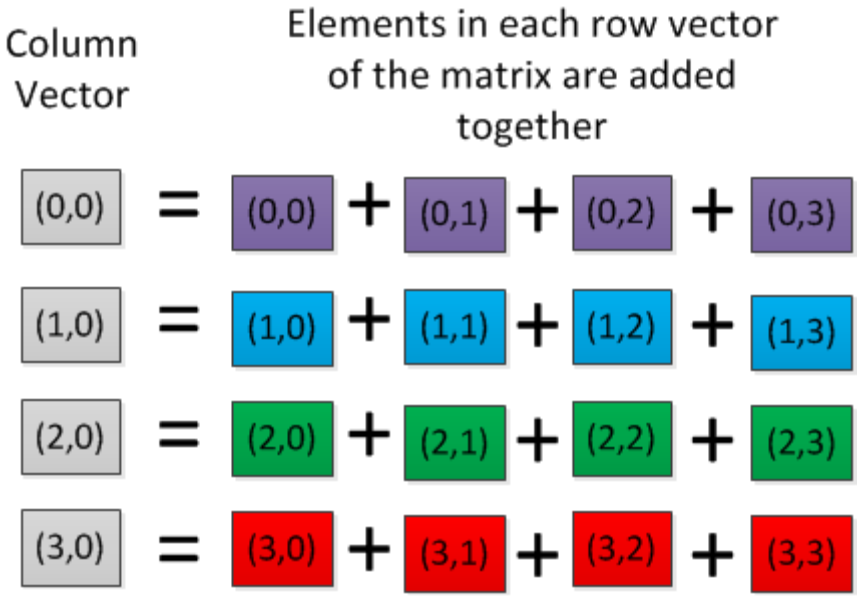


Figure 21: Matrix reduction in NTEPI

Nvidia's reduction algorithm uses a tree-based approach within each thread block (Figure 22). The idea is to use multiple blocks to reduce large arrays. Each block will compute a piece of the final summation. In order to compute the final summation, blocks would have to communicate their partial results with each other. However, there is no global synchronization between blocks implemented in CUDA. One solution is to use kernel launches as a global synchronization point. This can be done by breaking the

problem with multiple calls to a single reduction kernel, shown in Figure 22. The kernel launch parameters are the only changes needed [27].

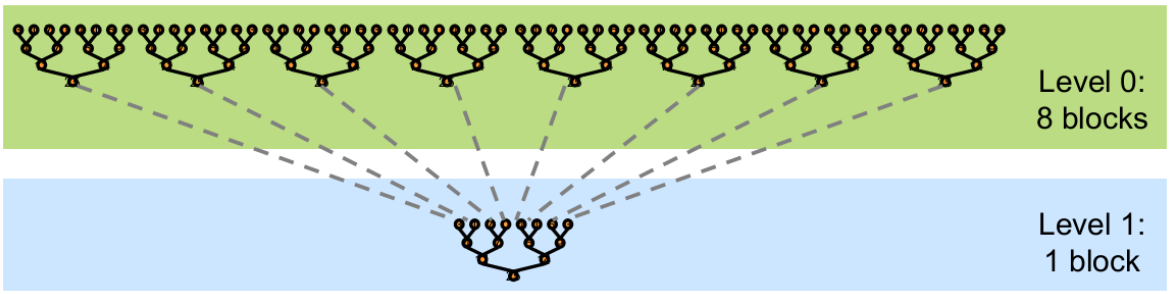


Figure 22: Parallel reduction using CUDA [27]

Nvidia implements several different reduction algorithms. Each successive implementation implements a different optimization technique to increase performance. The first strategy is common among all implementations. The strategy is to load elements from global memory into shared memory and do most of the reduction in shared memory. This strategy is done for two reasons: to coalesce memory reads form global memory and to conserve memory bandwidth accessing data more than once. If data were not loaded into shared memory then global memory accesses would not be coalesced during the reduction since not all threads access consecutive memory addresses. The next strategy is to remove shared memory bank conflicts by using thread ID based indexing. Since the reduction happens in shared memory, achieving the highest bandwidth possible is crucial. The next strategy is to remove idle threads. As the tree is traversed, more and more threads will become idle. It turns out having fewer threads do more work is more efficient [27]. This strategy reduces the overhead of calculating indexes and the number of idle threads which will use the calculated indexes only once. The last optimization is to unroll loops. Loops add additional overhead to each thread and should be avoided.

Nvidia's reduction implementation is limited to power of two sized arrays. This limitation in its algorithm poses a problem for NTEPI because none of the matrices have columns that are a power of two. There are two solutions to this problem: copy the incompatible matrix to a larger matrix that has a power of two column dimension or modify the kernel to accept arbitrary sizes.

Another disadvantage to Nvidia's reduction implementation is it reduces only one row vector. However, a matrix reduction can be split into several vector reductions. A

kernel call is needed for each vector reduction. Each of these kernel calls could be done concurrently on devices of CCAP 2.x. However, for older cards each kernel call would have to be done sequentially since concurrent kernel execution is not supported. One solution would be to create a kernel that reduced every row of a matrix at the same time. This solution seems likely to achieve the highest performance no matter what the CCAP of the GPU.

Before a vector can be reduced to a single number, something very important needs to be taken into consideration. Remember how matrices are stored in column-major order? This means the elements in a row do not have consecutive addresses. Accessing a row vector from a matrix in column-major order will cause un-coalesced memory accesses on the device. There are two directions one could take: transpose the matrix before performing the reduction or change the kernels to read the row vectors in an un-coalescing manner. The latter option will not be explored because accessing memory in an un-coalesced manner will significantly hurt performance much more so than taking a transpose.

The first implementations (version 0a and version 0b) used Nvidia's original reduction kernel. The first step is to transpose the input matrix if specified. Since Nvidia's reduction kernel must have row vectors with a power of two elements, the number of columns of the input matrix is checked if it is a power of two. If it is not then the matrix is copied into a larger matrix with a power of two dimension. Next, the matrix is reduced by calling several of Nvidia's reduction kernels, one for each row of the input matrix or one for each column of the transposed input matrix. The Nvidia's reduction kernel can be launched using streams (version 0b) with CCAPs 2.x or without streams (version 0a). Nonetheless, for each kernel call, a thread block will write a partial sum to a temporary matrix. This temporary matrix will have a row for each kernel call and a column for each thread block that Nvidia's reduction kernel had. In summary, each kernel call will write to a row of the temporary matrix. Lastly, the output column vector is computed by adding each column of the temporary matrix to the output column vector. This matrix reduction is summarized in Figure 23 below.

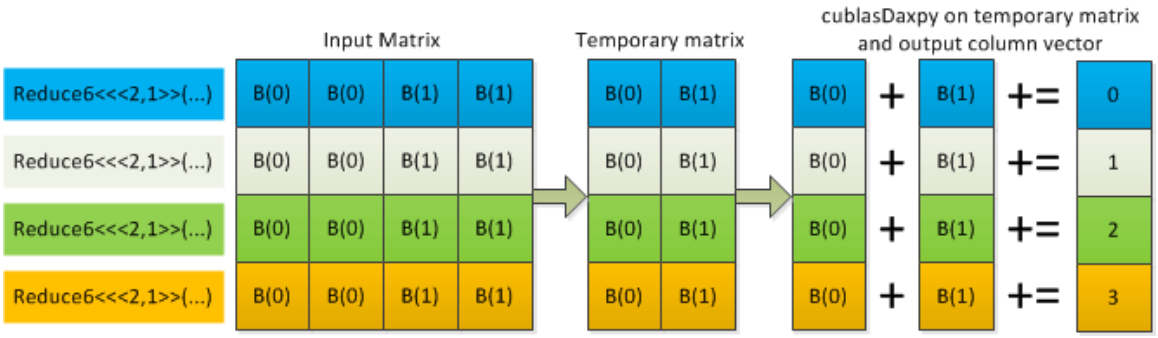


Figure 23: Simplified example of matrix reduction using CUDA version 0. Nvidia's reduction kernel is designated by *"reduce6"*. Each of these kernels spawns 2 blocks. Each block has one thread. Block indices are designated by "B(#)".

The next implementation (version 1) focused on modifying the *reduce6* kernel to reduce every row vector at the same time. Rather than calling a kernel for each row, one kernel was called to implement the first part of the reduction, from input matrix to temporary matrix. Other than that difference, reducing a matrix with this function is the same as reducing a matrix with version 0.

The last reduction based implementation (version 2) removed the need to resize the input matrix if the number of columns was not a power of two. The requirement was removed by modifying Nvidia's reduction kernel again to handle arbitrary row vector sizes. Other than that difference, this implementation is the same as version 1.

Failed Matrix Reduction Implementation

Another implementation was designed based on version 1 and version 2 but failed to function properly on all GPUs. The main difference was the removal of the CUBLAS library calls that performed the final reduction step. Instead, this reduction kernel performed the entire reduction in one kernel. Once the first thread-block of each row vector wrote its partial sum to global memory, the kernel would attempt to add all the partial sums (equal to the number of thread-blocks allocated to each row vector) for that vector. However, there is no guarantee that the other thread-blocks operating on this row vector have finished writing their partial sum to global memory. The implementation functioned on System C, but on System A the output was incorrect. In fact, the output was not-a-number (NAN). NAN was produced because a thread was writing to global memory when another thread was trying to access the same memory location. The main reason this implementation failed to work correctly was because thread-blocks cannot be

synchronized. Only threads within the same block can be synchronized. It just so happened to work on System C because all thread-blocks finished writing to memory, but this is not guaranteed to happen.

3.5.2 Matrix Reduction via CUBLAS Library

The second way to reduce a matrix, denoted by version 3, without using reduction is to use the CUBLAS library's matrix/vector addition kernel on each column vector of the matrix. Since elements are stored in column-major order all the elements in each column are stored in consecutive memory addresses. This means there is no need to transpose the matrix. Each element in the column is added to the corresponding element in the output column vector. Without double-precision floating-point atomic-adds, each column vector addition must be done serially since multiple vector additions are reading and writing to the same memory address.

3.5.3 Matrix Reduction Results

Two matrices, *A* and *B*, were used to test matrix reduction. The sizes of these matrices were based on an actual matrix that is reduced in NTEPI. This matrix has the following dimensions: 2084 x 4168. Matrix *A* was created to have a power of two number of columns (2084 x 4096) while matrix *B* was created to be very similar to matrix A but without a power of two number of columns (2084 x 4097). In fact, matrix *B*'s column dimension illustrates the worst case scenario for the matrix reduction functions that require power of two column dimensions. The next power of two size for 4,097 is 8,192. Having to increase the size of the matrix adds overhead to the matrix reduction. Rather than adding 4097 elements per row, 8192 elements need to be added per row. The extra elements are zeroed so they do not influence the output column vector. The matrix reductions were also done with and without a matrix transpose to reveal the overhead in performing the transpose. Remember, a transpose is necessary in obtaining the correct results for the matrix reduction implemented with parallel reduction.

The reduction based implementations had the kernel launch parameters (block and grid sizes) hand tweaked to achieve the best performance possible. The block sizes needed to be a power of two since the loop unrolling used block sizes that were powers-of-2.

Matrix reduction via the CUBLAS library or version 3 did not require the programmer to provide kernel launch parameters. The CUBLAS library automatically does this for the programmer.

From Table 5 Figure 13 and Figure 14, version 2 had the highest speedup of 3.58 (with input transposed) out of all three of the CUDA implementations and therefore is used in NTEPI. In addition, the size of the matrix has little effect on the performance of the function. Similarly, version 3 was not affected by the size of the matrix in these tests. In contrast, version 0a (without streams) had a 14.3 ms increase in execution time when reducing *A* compared to reducing *B* (with transpose). Version 0b performed similarly to version 1. This result is as expected since using streams is practically the same as version 1. However, the implementations that reduced every row at the same time were faster.

Table 13: The execution times of the five matrix reduction implementations on System A and a CPU implementation named cpuMxReduce. Each function was called 25 times and the times were divided by 25. Matrix *A* has dimensions 2084 x 4096. Matrix *B* has dimensions 2084 x 4097.

	Execution time Reducing A (ms)	Execution time Reducing B (ms)	Execution time Reducing A (no transpose) (ms)	Execution time Reducing B (no transpose) (ms)
CUDA version 0a	22.8	37.1	18.7	26.5
CUDA version 0b	10.3	26.4	6.0	15.9
CUDA version 1	8.0	24.5	3.4	13.8
CUDA version 2	5.7	6.1	1.7	2.1
CUDA version 3	12.3	12.3	12.3	12.3
CPU	20.4	20.2	26.9	26.3

Table 14: The speedups of the five matrix reduction implementations on System A with respect to the CPU implementation named cpuMxReduce. Speedups for a given column are compared to the cpuMxReduce function for that column. Speedups were derived from Table 13.

	Speedup of Reducing A	Speedup of Reducing B	Speedup of Reducing A (no transpose)	Speedup of Reducing B (no transpose)
CUDA version 0a	0.89	0.54	1.44	0.99
CUDA version 0b	1.98	0.77	4.48	1.65
CUDA version 1	2.55	0.82	7.91	1.91
CUDA version 2	3.58	3.31	15.82	12.52
CUDA version 3	1.66	1.64	2.19	2.14
CPU	1.00	1.00	1.00	1.00

Using streams in version 0b greatly improved performance because there is great overlap in kernel execution, shown in Figure 24. Notice the device percent utilization near the cursor. It begins to lower once kernels finish executing in streams 19 and 20 but rises when streams 6, 7, and 8 begin executing kernels again. When the device percentage utilization lowers, hardware on the device frees up. When the utilization reaches 100%, all the resources are being used in the device. In other words, kernels execute at the same time when there are enough resources available.

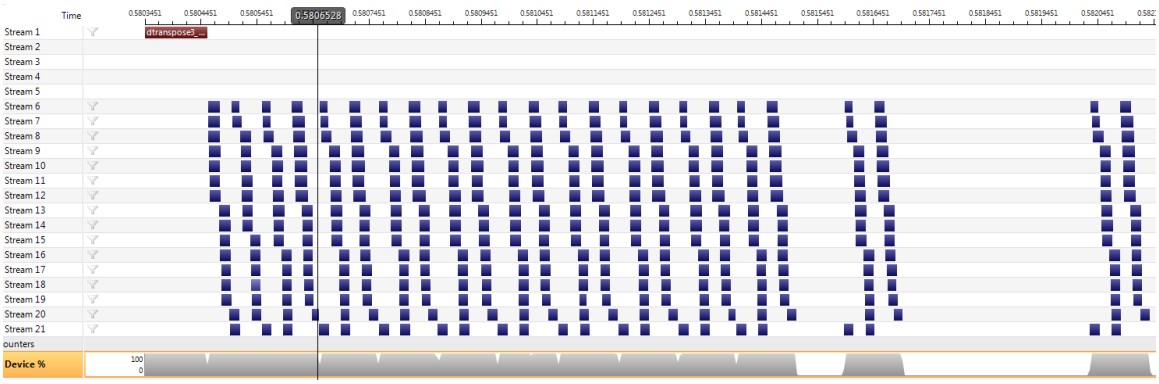


Figure 24: The kernel overlap when executing version 0b on matrix *A* with 16 streams on System A. The maximum concurrency reached is 7. Device % is the percent of time a kernel is executing on the device.

3.6. Implementing up in CUDA

The third function implemented in CUDA contained matrix scaling, matrix multiplication, matrix addition, matrix reduction and matrix-vector subtraction. All these operations are highly data parallel operations. Therefore, CUDA can be used to accelerate

these operations. These operations are summarized in Figure 25 below. First, columns in matrix *samX* are copied to the *meanX* column vector and the *sample* matrix. Next, these matrices that were just copied are scaled. Third, matrix reduction is carried out on the *sample* matrix and its results are written to the *meanX* column vector. Fourth, a matrix-vector operation updates matrix *samX*. The next two operations perform matrix multiplications using matrix *samX* as input. Lastly, matrix *pX* is updated by performing matrix addition on the results from the previous operations. Operations than can be executed in parallel are denoted by streams. Remember, executing operations in the null stream implicitly synchronizes all CUDA operations. A null stream operation cannot start until all previous operations have completed. The null stream is used to enforce data dependencies through implicit synchronization.

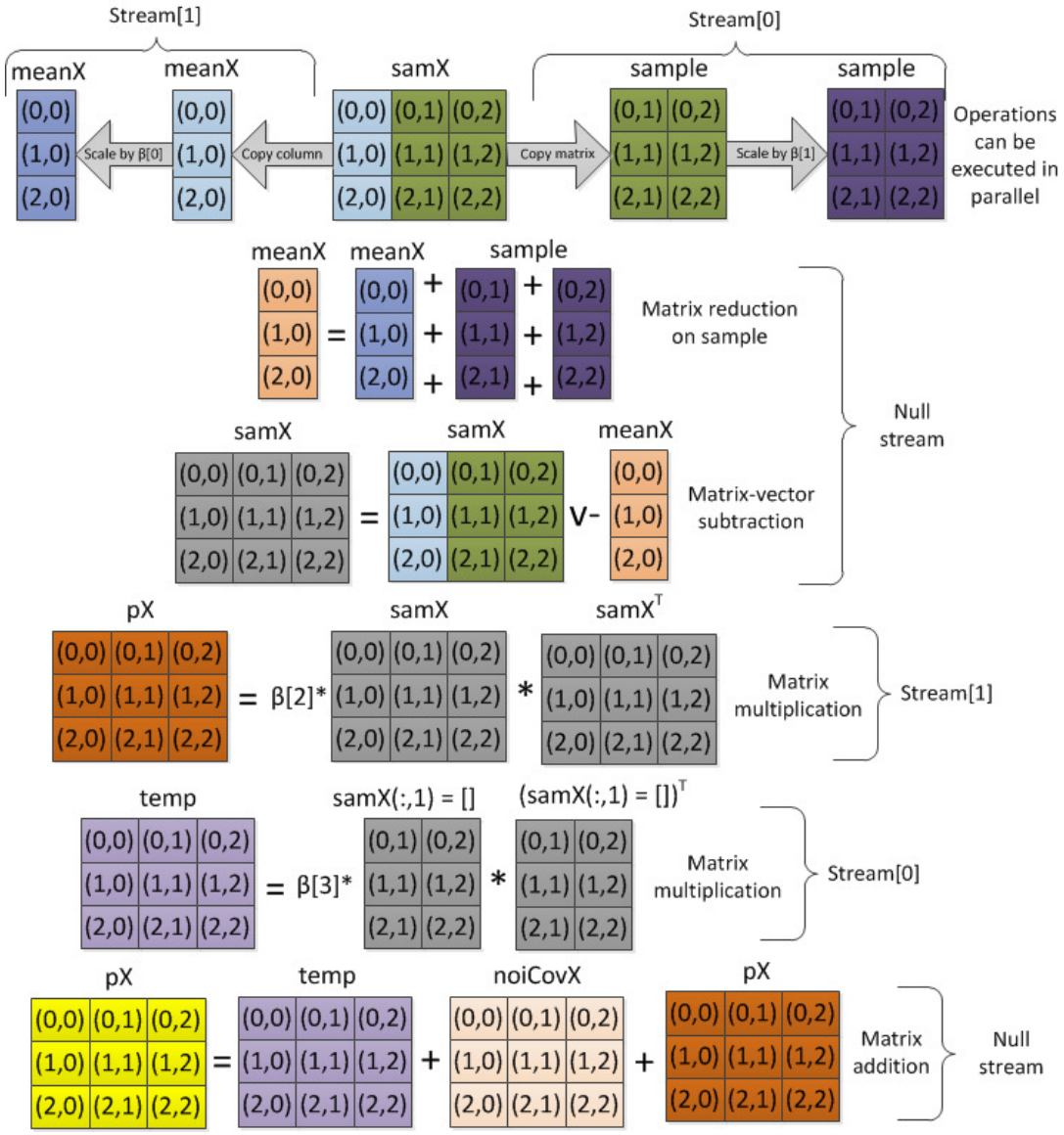


Figure 25: A summary of the operations carried out in *upd*. Please note matrix sizes are created for illustrative purposes only and do not reflect actual sizes used in any test case.

Versions 0a and 0b did not use multiple streams. The only difference between the two versions is the removal of copying the first column of matrix *samX* in version 0b.

The second version (version 1) used two streams, not counting the null stream. Concurrent execution of the first operations indicted on Figure 25 was attempted. That is, the left side was placed in one stream while the right side was placed in another stream. The next operations, matrix reduction and matrix-vector subtraction, were placed in the null stream. Doing so implicitly synchronizes the operations in all streams, meaning the matrix reduction and matrix-vector subtraction will not begin execution until all previous

work has been completed in all streams. This was intentionally done because there is a data dependency between the operations in the streams mentioned and in the matrix reduction and matrix-vector subtraction. Next, the two matrix multiplications can be executed in parallel. Therefore each is executed in a separate stream. Again, the last operation, matrix addition, is placed in the null stream to enforce the data dependency between the operations.

This last version (version 2) is the same as the previous version except the matrix addition was transformed from two CUBLAS kernels into one custom kernel. This change was made because matrix *pX* was read and written twice. With the custom kernel, matrix *pX* is only read and written once.

3.6.1 Results

The results in Table 15 show that only a minute improvement resulted from revisions. Using multiple streams did not significantly help performance. Streams did not provide much improvement because the kernels that were running in different streams could not be significantly overlapped. As suspected, allocating and copying less data within the device was faster. The greatest improvement came from improving the last operation in Figure 25, the matrix addition. In version 0 and version 1 this matrix addition is implemented with two CUBLAS kernel calls. In version 2, this matrix addition is implemented with one custom kernel. This kernel reduces the number of times matrix *pX* is accessed from global memory by 50%. Implementing this matrix addition with the custom kernel was 1.47 times faster than the implementation that called two CUBLAS kernels. The speedup and the reduction of memory accesses directly correlate with each other. The 50% reduction in memory accesses yields the 50% speedup obtained with the custom kernel. These results are shown in Table 16.

Table 15: The performance results of all implementations of *upd*. Each implementation was run 25 times on System A.

Implementation	Average Execution time (ms)	Speedup
CUDA version 0a	142.491	6.392
CUDA version 0b	142.413	6.395
CUDA version 1	142.119	6.408
CUDA version 2	141.452	6.439
CPU	910.760	1.000

Table 16: The performance of the two implementations carrying out the matrix addition in Figure 25. Each implementation was run 25 times on System A. The CUBLAS kernels used a block size of 384 while the custom kernel used a block size of 256.

Implementation	Execution time (µs)	Speedup
2 CUBLAS kernels	2,044.18	1.00
1 custom kernel	1,389.51	1.47

3.7. Implementing upLinear in CUDA

This function consists of matrix multiplication, matrix inversion, and matrix addition and subtraction. These operations are highly parallel with the exception of matrix inversion which is moderately parallel. Therefore, CUDA can be used to accelerate the operations, shown in Figure 26. The first two independent operations are executed first because they are used later. By calculating these operations together, streams can be leveraged to help execute these operations in parallel. Next, matrix *pxht* is used twice to help calculate two matrices. Third, this result is used to calculate another matrix. Lastly, matrix *hpx* that was calculated in the very beginning is used to calculate another matrix. These last two operations are independent of one another, and therefore they can be done in parallel. These operations are summarized in Figure 26. Three versions of *upLinear* were implemented and are analyzed below.

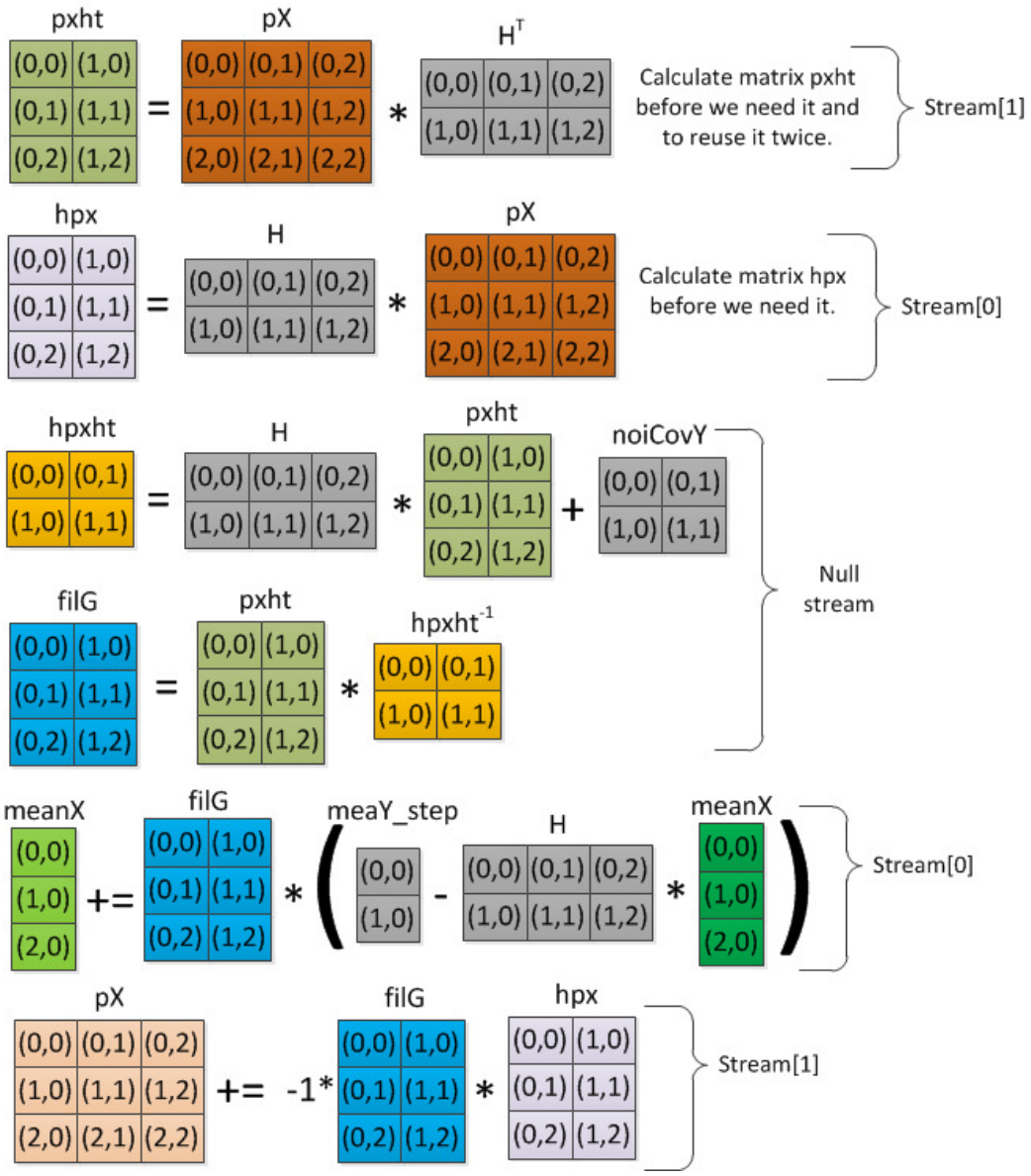


Figure 26: The operations carried out in *upLinear*. Please note matrix sizes are created for illustrative purposes only and do not reflect actual sizes used in any test case.

The overall design of version 0 followed Figure 26, except streams were not used. Since matrix *pxht* is used twice in this computation, it is calculated and saved for future use. Again, multiple streams were not used in this version. Lastly, matrix inversion was calculated by using an implementation of the LAPACK library called ACML (AMD Core Math Library). In order to use this library, data need to be transferred from device memory to host memory. Once the inverse is computed, the data need to be transferred back to the device to be used by other kernels.

The second implementation (version 1) improved first version by using two additional streams. They were used to execute the first and last two operations in parallel. Streams could not be used to overlap communication with computation since the results of the matrix inversion are needed to continue with the rest of the computation.

The last implementation (version 2) tried to improve the speed of matrix inversion. Instead of using a CPU library, the MAGMA CPU/GPU library was used. This library uses both the LAPACK and CUBLAS libraries to compute the inverse of a matrix.

3.7.1 Results

Table 17 shows all three CUDA implementations achieved relatively high performance compared to the CPU implementation. However, from Table 17 the best implementation was version 0. This function did not use additional streams. The version that did use multiple streams (version 1) executed 1.51% slower than the implementation without streams (version 0). The degradation in performance can be attributed to the overhead in allocating, using, and de-allocating streams. Apparently, if not enough computation can be overlapped, the overhead in using streams will hurt performance. Lastly, version 2 was the slowest because of matrix inversion and the use of streams. The MAGMA version of matrix inversion was 39.29% slower than the LAPACK version. The MAGMA version is slower because the matrix size is not large enough to occupy the device fully.

Table 17: The results of each implementation of *upLinear*. Each implementation was run 25 times on System A.

Implementation	Average Execution time (ms)	Speedup
CUDA version 0	11.579	38.951
CUDA version 1	11.756	38.365
CUDA version 2	12.268	36.761
CPU	451.000	1.000

In order to determine when to use MAGMA's implementation of matrix inversion, a test was created. This test had MAGMA and ACML each calculate the inverse of a matrix. Each implementation took the inverse of 25 different nonsingular

matrices with the same size, and they were run for several sizes. The execution times were then divided by 25 to find the average execution time for a given matrix size for each implementation. Matlab was used to generate a random nonsingular matrix. A nonsingular matrix was generated by creating a random orthogonal matrix. This matrix is special because it has an inverse (and the inverse is equal to its transpose). The result of this test is shown in Figure 27 below. It can be concluded that the MAGMA library should be used to invert a matrix that is greater than 112x112. As the matrix size increases, the MAGMA library's speedup also increases. The curve in Figure 27 is not smooth because both algorithms use a blocked algorithm to compute the inverse of a matrix. If the matrix cannot be divided evenly by the chosen block size then performance should degrade.

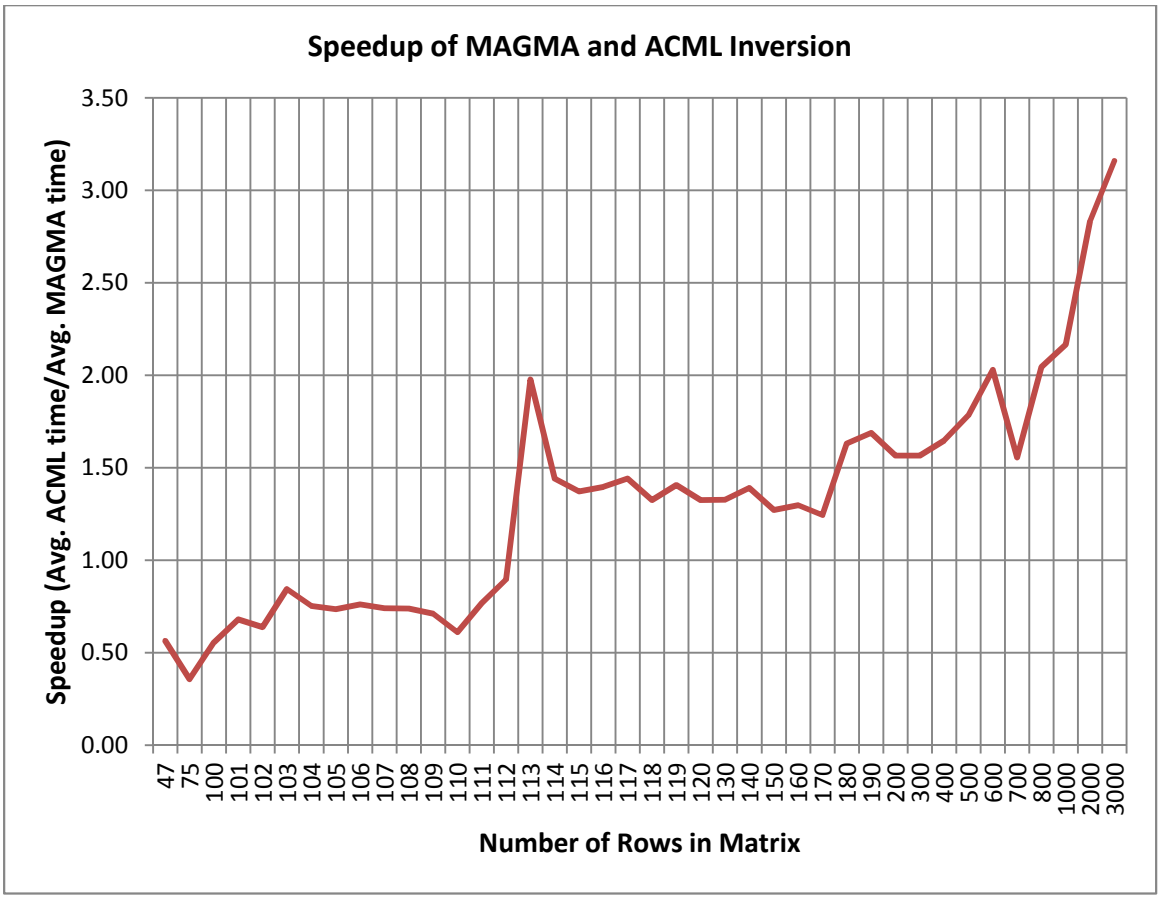


Figure 27: The speedup of MAGMA's inversion compared to ACML's inversion. Each execution time was average over 25 runs on System A.

Much of the execution time spent in the CUDA implementation of u*pLinear* is spent executing matrix multiplication. This is similar to *SamProRKA*. According to Table 18, about 80% of the time is spent on matrix multiplication.

Table 18: The percentage of time matrix multiplication (CUBLAS library) takes in the CUDA implementations in Table 17. Each implementation was run 1 time on System A.

Implementation (only matrix multiplication kernels)	Execution time (ms)	Percentage of time matrix multiplication takes
CUDA version 0	10.092	83.657
CUDA version 1	9.727	82.816
CUDA version 2	9.737	68.035

3.8. Implementing updateV in CUDA

This last method consists of several memory copies, matrix scaling, and matrix reduction. The matrix operations are highly parallel and therefore are accelerated with CUDA. The memory copies are not; however, since the matrices already reside on the device, CUDA is used. Transferring the data back to the host would only add additional overhead to NTEPI. Three versions of the CUDA implementation are presented here for analysis. Figure 28 below gives an overview of the operations carried out in this function. First, the first column of matrix *samV* is backed up. The next set of operations can be done in parallel, denoted by stream 0 and stream 1. Here, matrix *samV* is scaled except for the first column. The next operation copies the first column of matrix *samV* to column vector *meanV*. Next, this column vector is scaled. The nest operation is conducted in the null stream. Therefore, the previous operations denoted by stream 0 and stream 1 must finish before the matrix reduction starts. The next three operations can all be done in parallel since they are denoted by streams. *samV*'s columns except the last column become copies of the *meanV* column vector. Next, *samV*'s last column is replaced by the original data prior to any modifications. Lastly, the *meanX* column vector is copied to one of the columns in another matrix. The target column for the copy depends on an index variable within NTEPI.

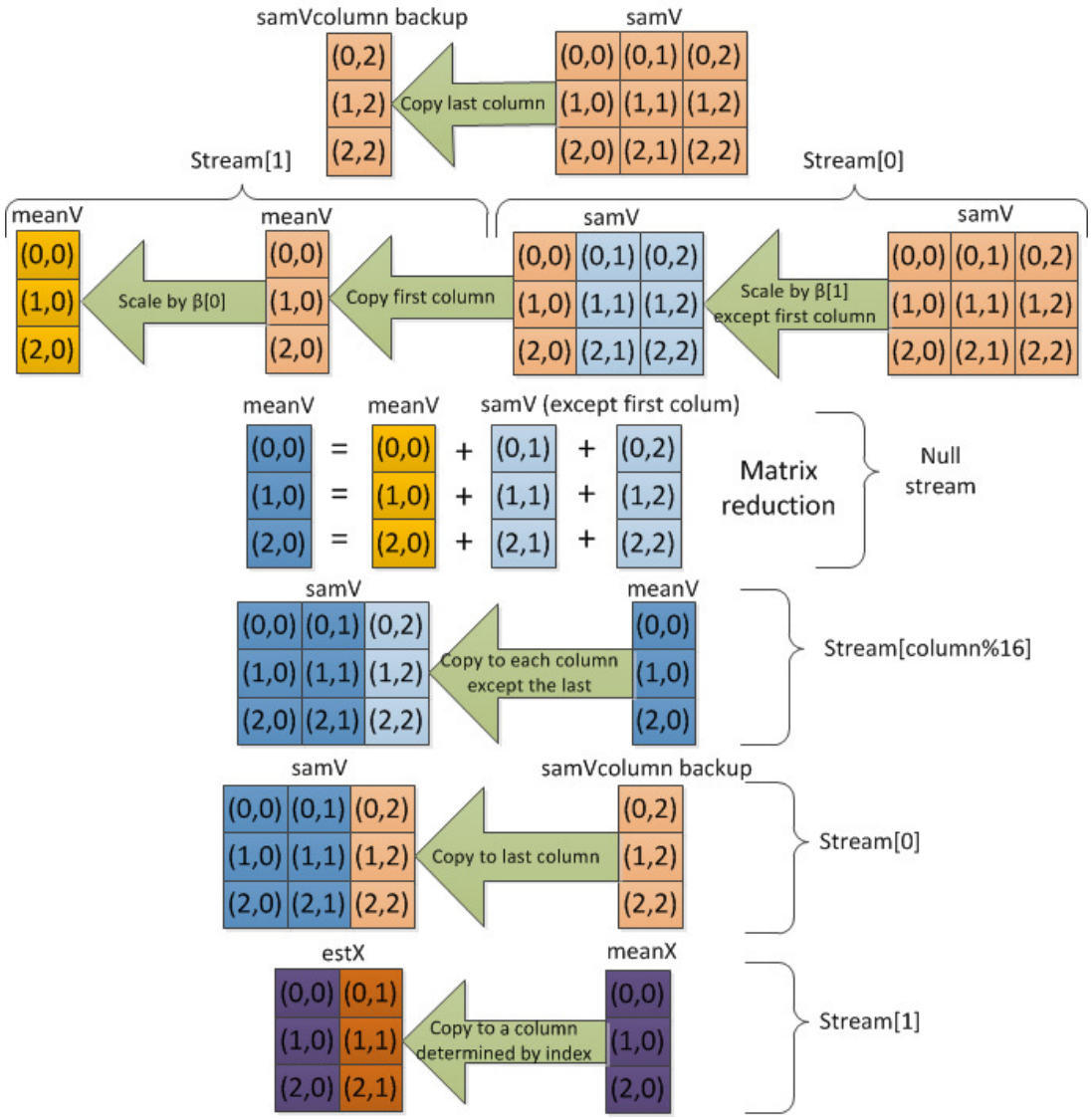


Figure 28: The operations in *updateV*. Please note matrix sizes are created for illustrative purposes only and do not reflect actual sizes used in any test case.

The first implementation (version 0) was slightly different than the operations shown in Figure 28. The main difference was instead of copying the last column of *samV*, the entire matrix was copied. It was noticed later that copying the entire matrix was not needed. Also, multiple streams were not used here.

The second implementation (version 1) improved by just copying a column of *samV*. Besides that change, nothing else was modified.

The last implementation (version 2) tried to use multiple streams. From Figure 28, the second operation and the last three operations used streams. The second of these four

operations was copying matrix *meanV* to each column of *samV*. Since there are as many memory copies as columns minus 1, streams could potentially improve performance by overlapping memory copies. These memory copies can potentially be overlapped because each copy is unrelated to the rest. However, each memory copy reads the same source. This could lead to bank conflicts when reading global memory. Next, the last two operations are unrelated memory copies and therefore can be done in parallel with streams.

3.8.1 Results

The CUDA implementations in Table 19 showed a meager speedup compared to the CPU implementation. The main reason for this subpar speedup is about 60% of the time is spent copying memory within the device, shown in Table 20. Memory bound operations on the device do not provide significant speedups.

Version 0 was the slowest because all of *samV* was copied unlike in the later revisions of the function. Version 1 and version 2 achieved similar speedups but version 2 was slightly faster. Version 2 was faster because the matrix scaling was executed in different streams. However, the fourth operation where matrix *meanV* was copied to the columns of matrix *samV* did not improve in performance using streams. The performance was about equal to the version that did not use multiple streams. To find out why, version 2 was profiled with Parallel Nsight in an attempt to understand why this operation was not faster. The memory copies at first glance looked to be executing concurrently, shown in Figure 29. However, upon further investigation Nsight revealed the data transfer rates were not consistent. Some transfers were fast and others were about 50% slower. However, all 16 streams were used. On the other hand, for version 1 the data transfers were all consistently faster than version 2's. The device did overlap memory copies but the reduction in transfer rates outweighed the achieved concurrency. The result was no net gain in performance.

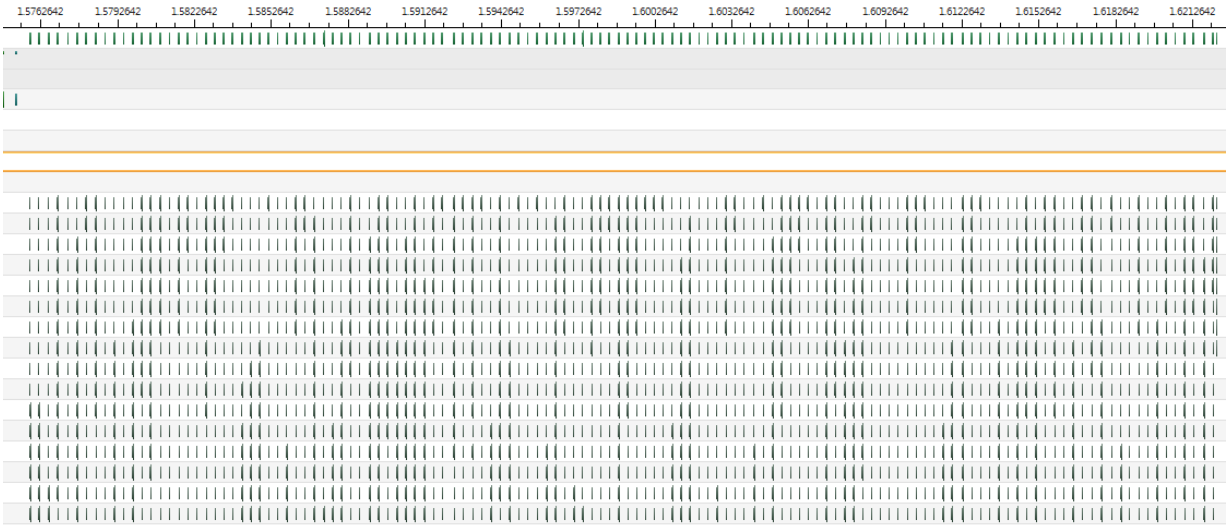


Figure 29: A screen capture from Parallel Nsight on System A. It shows the memory copies in the fourth operation of Figure 28 in version 2 using 16 streams.

Table 19: The results of each implementation of *updateV*. Each implementation was run 25 times on System A.

Implementation	Average Execution time (ms)	Speedup
CUDA version 0	24.08	4.17
CUDA version 1	21.35	4.70
CUDA version 2	21.20	4.74
CPU	100.40	1.00

Table 20: The results of each implementation of *updateV* with only memory copy operations. Each implementation was run 25 times on System A.

Implementation	Average Execution time (includes only memory copies) (ms)	Percentage of total execution time
CUDA version 0	11.78	56.40
CUDA version 1	11.10	58.17
CUDA version 2	11.08	57.96

3.9. Verifying CUDA Implementation

Throughout the development of this thesis, a method to verify the accelerated version provided the correct output was very important because a faster execution time means nothing if the output is incorrect. The output from the CPU version of NTEPI was considered to be correct. This output is referred to as the golden output.

The first method in verifying correct execution was examining the output from a complete run of a test case. Upon completion, the program would write its output to a binary file. This file would be compared against the golden output using a Matlab script. This method would only reveal if there was a problem or not. Obviously, another strategy needed to be developed to give the exact location of the bug. Also, this method was time consuming.

The next method was printing each matrix that was modified. Comparing these with the golden outputs would reveal where the problem was located. A simple file comparator was used to determine if the matrices were the same.

Another strategy used toward the end of development was to create separate projects to test each of the main NTEPI methods. Matrices were filled with random data. The CPU function would compute, and the results would then be saved to compare against the CUDA functions. This method was far more automated in determining correct execution. There was one problem with this method. Since matrices were filled with random data, matrix inversion would not work because the matrix was not invertible. A simple work around was developed: comment out the matrix inversion. This verification strategy found one bug in the code when it was thought the CUDA implementation was perfect. A read only matrix was being modified. This bug was quickly fixed.

3.10. GPU Profiling Results

Selecting the best GPU architecture for NTEPI is based on profiling results of running three NTEPI iterations on three GPUs. The three GPUs, shown in Table 3, represent the current high end, midrange, and low end CUDA GPUs. Rather than simply giving the best GPU out of the three GPUs analyzed, a set of hardware specifications was created that favors the NTEPI algorithm. These specifications will help future researchers chose the best (in terms of price and performance) GPU accelerator.

3.10.1 NTEPI Performance Bottleneck

The bottleneck in NTEPI is matrix multiplication in the *samProRKA* function for all three Systems. Performance results are shown in Figure 30 (chart formed from data in Table 22, Table 23, and Table 24). System A has the best performance because of its

superior double-precision performance even though System B has better hardware specifications than System A's, shown in Table 3. System C is about five times slower than System A. System C is slower because each SM has only 1 double-precision floating-point unit. System A and System B both have a double-precision floating-point unit per CUDA core. So, there are 32 in each SM [1]. In short, System A has the highest performance since matrix multiplication is compute bound and takes advantage of the superior double-precision support.

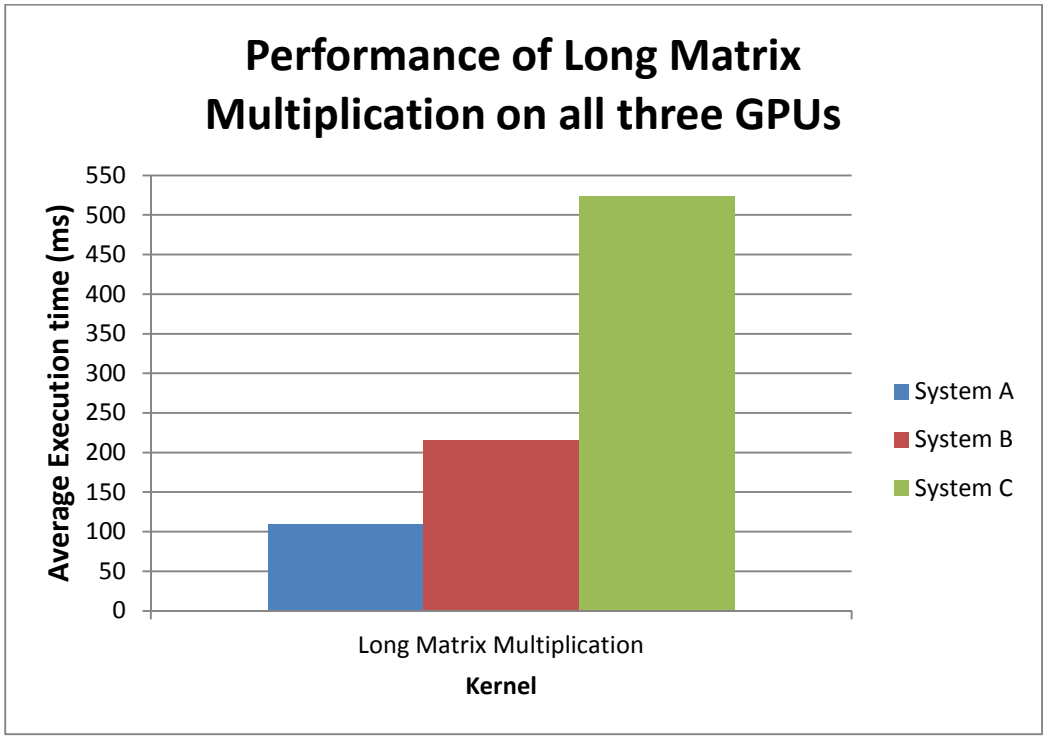


Figure 30: The performance of matrix multiplication in the *samProRKA* function in each System. Tests were conducted using three iterations.

3.10.2 Comparing performance of other top kernels

Table 22, Table 23, and Table 24 show the top 10 kernels in terms of highest device utilization time for System A, B, and C, respectively. Out of those kernels, five kernels common to all three tables were analyzed in Figure 31. Each kernel's performance in this figure is bound by the GPU's memory bandwidth. Comparing the performance of the kernels on Systems A and B shows System B was faster every time.

There are a few reasons for these results. Looking at Table 3 reveals GPU B has superior specifications in every regard except its double-precision performance. More specifically, GPU B has 23.2% more memory bandwidth, 7.14% more CUDA cores, 21.7% higher graphics clock, and 21.8% higher processor clock than GPU A. These differences translate to the percentage increases in Table 21. In conclusion, memory bound kernels perform better on GPU B than GPU A. The performance of GPU C is the worst due to inferior double-precision support, memory bandwidth, number of CUDA cores, and the available number of registers per SM. This last item caused fewer threads to execute in parallel. Having fewer threads that execute in parallel significantly hurts performance.

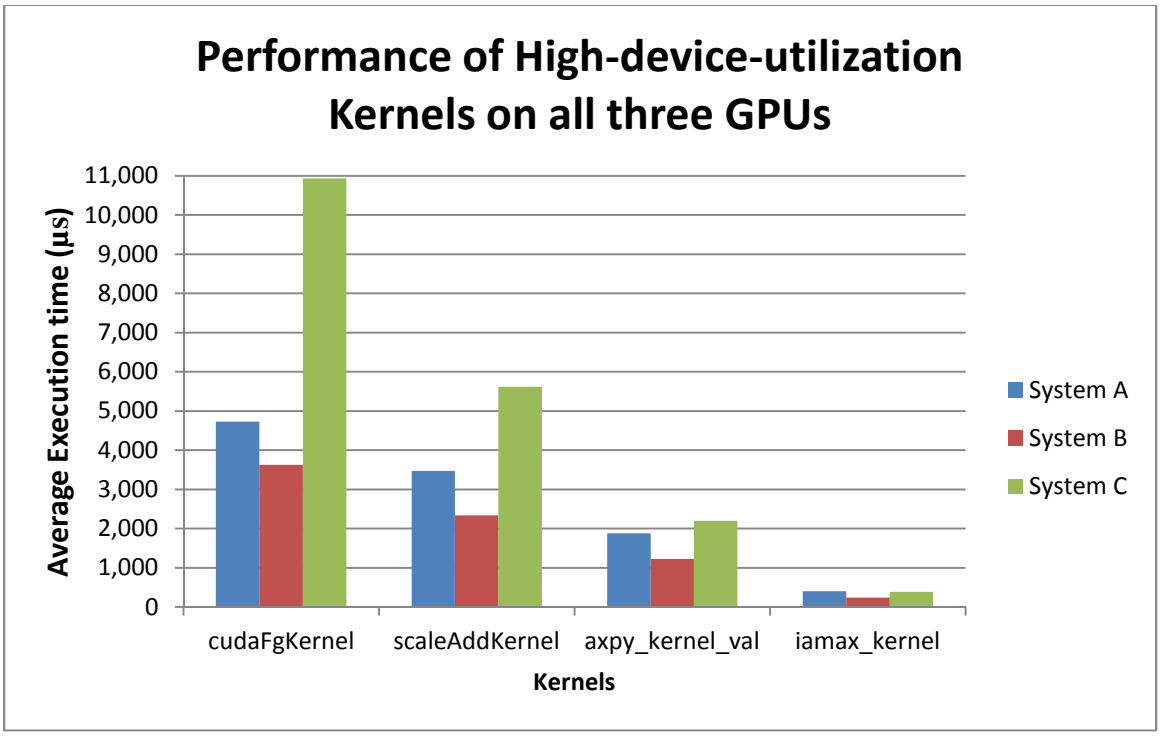


Figure 31: The performance of common-high-device-utilization kernels in each GPU. *cudaFgKernel* and *scaleAddKernel* carry out addition, subtraction, element multiplication, and scaling operations on matrices. *Axpy_kernel_val* adds/subtracts matrices. *Iamax_kernel* finds the maximum magnitude of an element in a matrix/vector.

Table 21: The percent increase in performance of GPU B over GPU A.

Kernel	Percent increase
cudaFgKernel	30.37
scaleAddKernel	48.40
axpy_kernel_val	53.48
iamax_kernel	69.06

Table 22: The top 10 kernels that have the highest device % in NTEPI (3 iterations) on System A. The execution time for three iterations was 51.57 seconds.

Kernel Name	Count	Device %	Total Device Time (ms)	Minimum time (µs)	Average time (µs)	Maximum time (µs)
fermiDgemm_v3_kernel_val	409	86.60	44744.46	150.20	109,399.66	115715.56
scaleAddKernel	600	4.00	2080.81	3453.90	3,468.01	3478.99
cudaFgKernel	400	3.70	1890.66	4718.12	4,726.65	4735.45
axpy_kernel_val	224	0.80	421.18	2.754	1,880.25	2,116.30
fermiDgemm_v2_kernel_va	6	0.70	345.97	632.45	57,661.16	114,714.99
iamax_kernel	400	0.30	160.50	5.543	401.26	807.268
gemm_kernel2x2_val	18	0.10	56.74	5.35	3,152.46	9,004.75
fermiDgemm_v3_kernel_val	45	0.10	33.91	259.46	753.61	1,173.05
trsm_right_kernel_val	48	0.10	31.01	243.38	645.98	1,168.69
dtranspose3_32	6	0.00	21.37	3,561.08	3,562.17	3,563.76

Table 23: The top 10 kernels that have the highest device % in the NTEPI (3 iterations) on System B. The execution time for three iterations was 93.46 seconds.

Kernel Name	Count	Device %	Total Device Time (ms)	Minimum time (µs)	Average time (µs)	Maximum time (µs)
fermiDgemm_v3_kernel_val	409	94.20	88026.91	213.66	215224.71	221002.67
cudaFgKernel	400	1.60	1450.27	3618.98	3625.67	3634.71
scaleAddKernel	600	1.50	1402.18	2329.31	2336.97	2340.62
fermiDgemm_v2_kernel_val	6	0.70	645.17	884.26	107528.26	214197.81
axpy_kernel_val	224	0.30	274.41	2.25	1225.06	1383.98
iamax_kernel	400	0.10	94.94	4.88	237.35	477.85
fermiDgemm_v3_kernel_val	45	0.10	55.55	405.49	1234.49	2168.26
gemm_kernel2x2_val	18	0.10	50.84	3.84	2824.35	8029.31
fermiDsyrk_v2_kernel_val	30	0.00	32.62	627.09	1087.35	1547.44
fermiDgemm_v3_kernel_val	6	0.00	28.47	712.33	4745.35	8801.73

Table 24: The top 10 kernels that have the highest device % in the NTEPI (3 iterations) on System C. The execution time for three iterations was 224.256 seconds.

Kernel Name	Count	Device %	Total Device Time (ms)	Minimum time (µs)	Average time (µs)	Maximum time (µs)
gen_dgemmNN_val	403	94.1	211,065.69	0	523,736.19	528,510.73
cudaFgKernel	400	2	4,372.34	10,603.78	10,930.86	10,937.74
scaleAddKernel	600	1.5	3,370.11	5,400.68	5,616.86	5,628.39
gen_kmul4_dgemmNT2_val	39	0.8	1,699.29	0	43,571.49	521,803.64
axpy_kernel_val	224	0.2	491.63	0	2,194.76	2,998.45
iamax_kernel	400	0.1	152.14	8.8	380.347	770.278
trsm_right_kernel_val	48	0	95.17	778.47	1,982.65	3,690.62
dtranspose3_32	6	0	53.11	0	8,851.02	10,630.25
gemm_kernel1x1_val	12	0	30.25	0	2,520.82	13,694.88
syherk_kernel_val	45	0	28.26	80.67	628.06	1,114.28

This chapter analyzed implementing each of the five NTEPI filter functions was implemented with CUDA. Each of these functions was optimized to provide the best performance on CUDA enabled GPUs. Using streams to overlap kernels did not provide much improvement due to the large matrices used in NTEPI. To verify the CUDA implementations were correct, the output of each of the functions was compared against the baseline CPU version. The performance of the kernels was thought to be highest in System A compared to Systems B and C. However, it was found that only matrix multiplication was faster on System A. System B executed bandwidth bound kernels faster. Although the ALUs executed slower, the ALUs were fed faster. System A executed NTEPI the fastest since matrix multiplication was executed the fastest on this system and matrix multiplication comprised an overwhelming majority of the execution time.

Chapter 4 NTEPI Acceleration Strategies

This chapter will discuss the overall strategies used to accelerate NTEPI with CUDA. The last two strategies described here use the work from Chapter 3. Single-precision and double-precision is also discussed. Lastly, the input files to NTEPI will be briefly described for reference purposes.

4.1. First Acceleration Strategy

The first strategy in parallelizing NTEPI was to accelerate a single operation in the matrix class with CUDA: matrix addition (A = A + B). This operation can be executed on the GPU using the CUBLAS library. Each time this matrix addition is encountered, device memory needs to be allocated (not included in the time), data has to be transferred to the device (matrices *A* and *B*) and then matrix *A* needs to be transferred back to the host when the device has completed execution, and device memory must be de-allocated (not included in the time). The timing results are shown in Table 25. It is evident that data transfers can significantly hurt GPU performance for this strategy. Factoring in the data transfers, matrix addition using the CUBLAS library achieved a speedup of 0.248 compared to a non-optimized CPU implementation. Therefore, avoiding data transfers between the host and device will usually result in much higher performance.

Table 25: Timings of inefficient matrix addition (A = A + B) using double-precision on System C where matrices have dimensions 2084 x 4169. Addition was carried out 10 times, which includes computation and communication.

Average host to device transfer time (ms)	Average device to host transfer time (ms)	Average device execution time (ms)	Total device execution time (computation and communication)(ms)	Average CPU execution time (ms)
141.80	61.40	2.58	205.78	51.11

4.2. Second Acceleration Strategy

The second attempt in accelerating NTEPI was to accelerate the computation outside of the matrix class and within the five main methods of NTEPI's computation. A discussion of the parallelization of these five functions is in Chapter 3. This strategy

reduced communication further because data transferred to the device was used by several kernels rather than one kernel before being transferred back to the host. However, all the large data structures still remained on the host. This required all necessary data to be transferred to the device before one of the main NTEPI functions was called. Any data then modified on the device need to be transferred back to the host after the function call. In short, the device never keeps any data stored in its memory between NTEPI's five main function calls.

To aid in development, a matrix class for CUDA was developed. This class handled all CUDA memory allocation and de-allocation. It provided debugging functions and wrapper functions to the CUBLAS library and MAGMA library. The class was designed to interface with the host matrix library to make allocating and copying host memory to the device easier.

Performance Results

From Table 26, System A performs the best out of the three Systems. This System is about 66.26% faster than System B. Despite System B having better hardware specifications except double-precision performance, System A was faster due to its superior double-precision support. A remaining question is how much more speedup can be obtained by removing even more communication between the host and device?

Table 26: The performance results of running NTEPI using the second acceleration strategy

	CPU time (minutes)	CPU/GPU time (minutes)	Speedup
System A	1931.12	147.82	13.06
System B	X	245.82	7.86
System C	X	609.73	3.17

4.3. Third Acceleration Strategy

This design was chosen to have the host handle mainly two parts of NTEPI: scalar arithmetic and control flow operations. The device would handle operations on vectors and matrices. Before the NTEPI iteration begins, device memory is allocated and host data are transferred to the device. Once the host has reached the main loop of the

program, only a few transfers and allocations occur within the main loop. This methodology lowers the overhead associated with memory allocations, de-allocations, and transfers between the host and device.

The design, shown in Figure 32, begins with the host loading data from files and placing the data into device memory. From this point, all the data structures reside in device memory. The host then enters the main loop of NTEPI. From this point on the host mainly instructs the device what to compute. Upon completion of the main loop, the host instructs the device to transfer the results back to the host. Once finished, both the host and device de-allocate memory. In order to keep all of these data structures within the device updated, CUDA calls are needed outside of these main functions. Most of these CUDA operations are simple memory sets and memory copies. These operations contribute very little execution time to NTEPI, and therefore are not illustrated below.

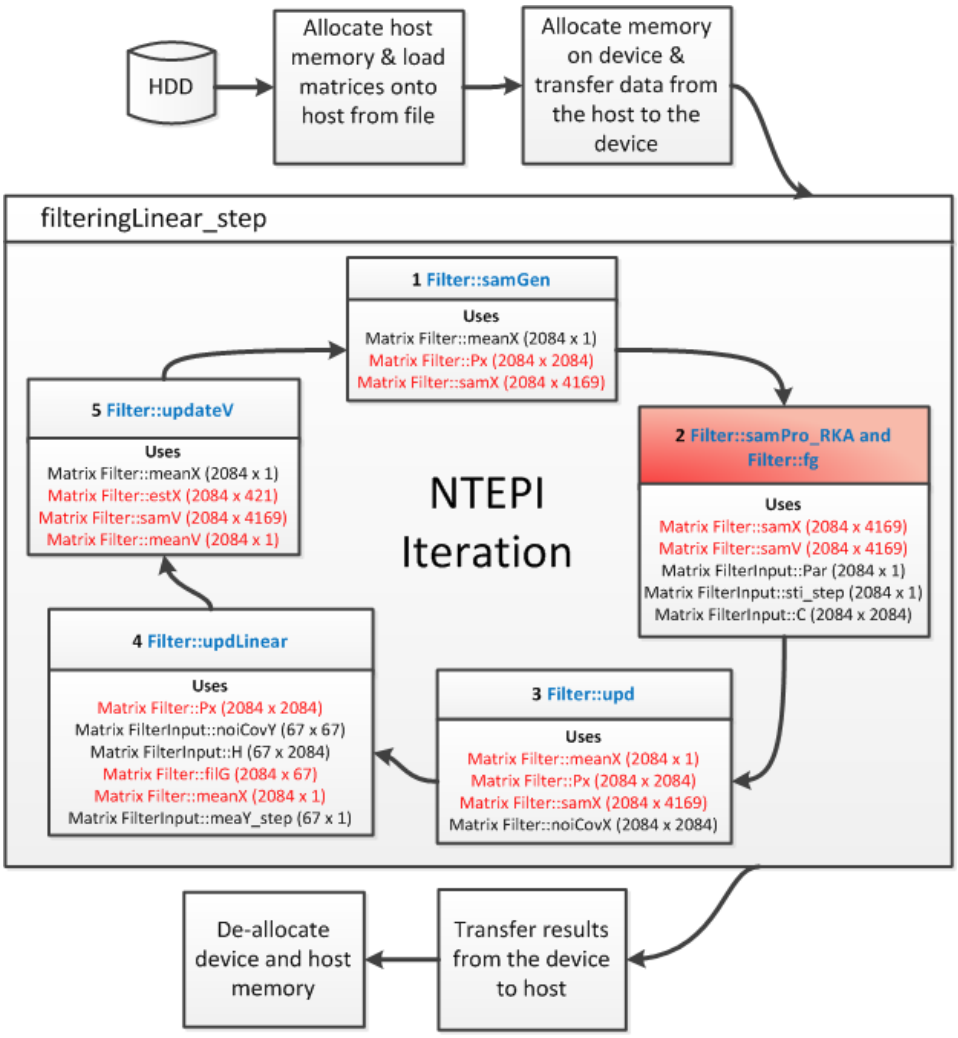


Figure 32: Overall parallel architecture of NTEPI. Function is shaded in red since it is still the bottleneck in the GPU version.

Comparing the results in Table 27 to Table 26, strategy three on Systems A and B increased performance by 25.42% and 14.0%, respectively. System C's performance increased by 17.4%. These increases in performance are directly caused by the removal of communication between the host and device.

Table 27: Performance results of running NTEPI using the third acceleration strategy.

	CPU time (minutes)	CPU/GPU time (minutes)	Speedup Compared to System A CPU execution time
System A	1931.12	117.88	16.38
System B	X	215.64	8.96
System C	X	519.38	3.72

This design could potentially consume more memory than a GPU has available. Current test cases use about 506.07 MB of device memory.

4.4. Implementing NTEPI in Both Single-Precision and Double-Precision

NTEPI was designed to execute entirely in either single-precision or double-precision arithmetic. This design was accomplished by using *typedefs* and compiler *if* statements. Changing a single *#define* would cause NTEPI to use either single-precision or double-precision arithmetic on both the host and device.

A solution needed to be found to convert the double-precision input files to single-precision when executing in single-precision mode. The first attempt was to read the double-precision input and convert the data to single-precision right after the file was read. This strategy was hindered by bugs. The second attempt was to remove the data conversion from NTEPI and into a secondary program. This strategy proved much easier and was prone to fewer bugs.

In the beginning of implementing the designs in this thesis, double-precision arithmetic was used. This precision was adopted because the baseline CPU version did. NTEPI outputs a single binary file upon completion of a test case. To compare the outputs of NTEPI, a Matlab script was written. Equation 7 shows the Matlab code used to compute the errors represented in Table 28 and Table 29. The single-precision and double-precision CUDA implementations were each compared against the single and double-precision CPU implementation, Table 28 and Table 29 respectively. Table 28 illustrates both CUDA implementations differ greatly from the single-precision CPU

implementation. Table 29 reveals the CUDA implementation using double-precision matches the CPU implementation using double-precision, with a maximum relative error of 8.39e-5. Furthermore, the same table shows the CUDA implementation using single-precision has a maximum relative error of 201.21. Therefore single-precision cannot be used for NTEPI.

The CUDA implementations appear to compute more accurate output than the CPU implementations. This observation can be derived from the results in Table 28 and Table 29. The CPU version using single-precision varied greatly from the CUDA version using single-precision. In contrast, the CPU version using double-precision varied moderately from the CUDA version using single-precision. Table 30 supports this theory because the CPU implementation using single-precision also has a large error with respect to the CPU implementation using double-precision.

```
   relError = (abs(dataRef - data2))./abs(dataRef);
maxRelError = max(max(relError))
minRelError = min(min(relError))
```

Equation 7

Table 28: Comparison of NTEPI's output using single and double-precision implemented with CUDA with the CPU version of NTEPI using single-precision. The synthetic test case on System A was used.

	Maximum Difference	Minimum Relative Error	Maximum Relative Error
CUDA Double-Precision	1.1829	6.6881e-011	3.6204e+005
CUDA Single-Precision	1.1830	0	3.6203e+005

Table 29: Comparison of NTEPI's output using single and double-precision implemented with CUDA with the CPU version of NTEPI using double-precision. The synthetic test case on System A was used.

	Maximum Difference	Minimum Relative Error	Maximum Relative Error
CUDA Double-Precision	6.92e-009	0	8.39e-005
CUDA Single-Precision	0.11	7.54e-011	201.21

Table 30: Comparison of the CPU version of NTEPI's output using single with the CPU version of NTEPI using double-precision. The synthetic test case on System A was used.

	Maximum Difference	Minimum Relative Error	Maximum Relative Error
CPU Single-Precision	1.1829	6.6878e-011	2.9851e+006

4.5. Input Binary Files Used by NTEPI

All of NTEPI's data structures originate from binary files shown in Table 31.

Table 31: Lists and describes the binary input files that NTEPI uses where *n* is the number of nodes representing the heart, *m* is the number of data points on the body surface, and *t* is the number of steps.

.bin file	Loaded into the following C variable(s)	Data type	Matrix dimensions	Description
Trans_state	C	double	n x n	Describes the diffusion process for the epicardial potential propagation from node to node
Trans	H	double	m x n	Describes the relationship between the electric potential distribution in the heart and on the body surface
Parameter_R2	Par	double	n x 1	Used to help determine which areas of the heart are damaged
time_inverse	time_inverse	double	t x 1	Coarse grained time data for ECG samples
time_P	time	double	t x 1	Fine grained time data for generating epicardial potentials
Measurement_noisy	meaY	double	m x t	Input ECG data. Each column represents the potential distribution at each time point. Each row represents the ECG trace of one node on the body surface
Noise_mea	noiseY & noiCovY	double	m x 1	Noise covariance value for each node on the body surface
Init_P	Px	double	n x n	Initial covariance values
Noise	noiCovX	double	n x n	Noise

4.6. Best GPU Architecture for NTEPI

There are two directions for choosing the best GPU architecture for NTEPI: the highest possible performance and cost effective high performance. Guidelines for choosing the best CUDA enabled GPU are presented below.

Highest Possible Performance

For the highest possible performance, one would want a GPU that combines the architectures of GPU A and GPU B. A GPU with high memory bandwidth will accelerate bandwidth bound kernels and superior double-precision support will accelerate kernels that are compute bound like matrix multiplication. To increase further the speed of matrix multiplication and all other kernels, the number of SMs should be increased since the more SMs there are the more threads can execute in parallel.

Cost Effective High Performance

To get the best performance on a budget, GPU B works very well. CCAP 2.x and above will work best due to the need for double-precision support. CCAP below 2.x will not provide high performance in double-precision.

Chapter 5 Conclusions

NTEPI took about 32 hours on a high end processor to render one contraction cycle of the heart. This processing delay is not acceptable in a clinical setting. Profiling on the CPU found that about 98% of the execution time was contained within NTEPI's five filter functions. Since highly data parallel matrix operations are executed in these functions, CUDA can be used to accelerate these functions. NTEPI was sped up by a factor of 16.38 on System A with CUDA. To obtain this speedup, every NTEPI matrix and vector was placed on the device to avoid communication between the host and device. By placing all major data structures on the device all operations could be computed using the device no matter how insignificant an operation contributed to the total execution time. Therefore, communication between the host and device did not significantly contribute to the execution time. Moreover, kernels were condensed into a single custom kernel to conserve precious memory bandwidth, streams were used to overlap communication with computation and to overlap kernel execution as much as possible, and the MAGMA library was analyzed, enhanced, and leveraged in NTEPI. The major bottleneck in the CUDA implementation was large matrix multiplication in the samProRKA function. Matrix multiplication was implemented with the CUBLAS library by Nvidia. About 90% of the total execution time is within the samProRKA function executing large matrix multiplication. Until matrix multiplication can be sped up, there cannot be any significant improvement in performance for the CUDA implementation of NTEPI. Figure 33, shows that even when *samProRKA* was sped up by about 20.3, it still constitutes the majority of the execution time in NTEPI. Lastly, the best GPU to accelerate NTEPI without a concern for the cost would be one that combines the features of the Tesla C2070's double-precision performance and the increased memory bandwidth, higher clocks, and high number of SMs of the GTX 480. However, the cheapest GPU for the best performance would be a GPU similar to the GTX 480.

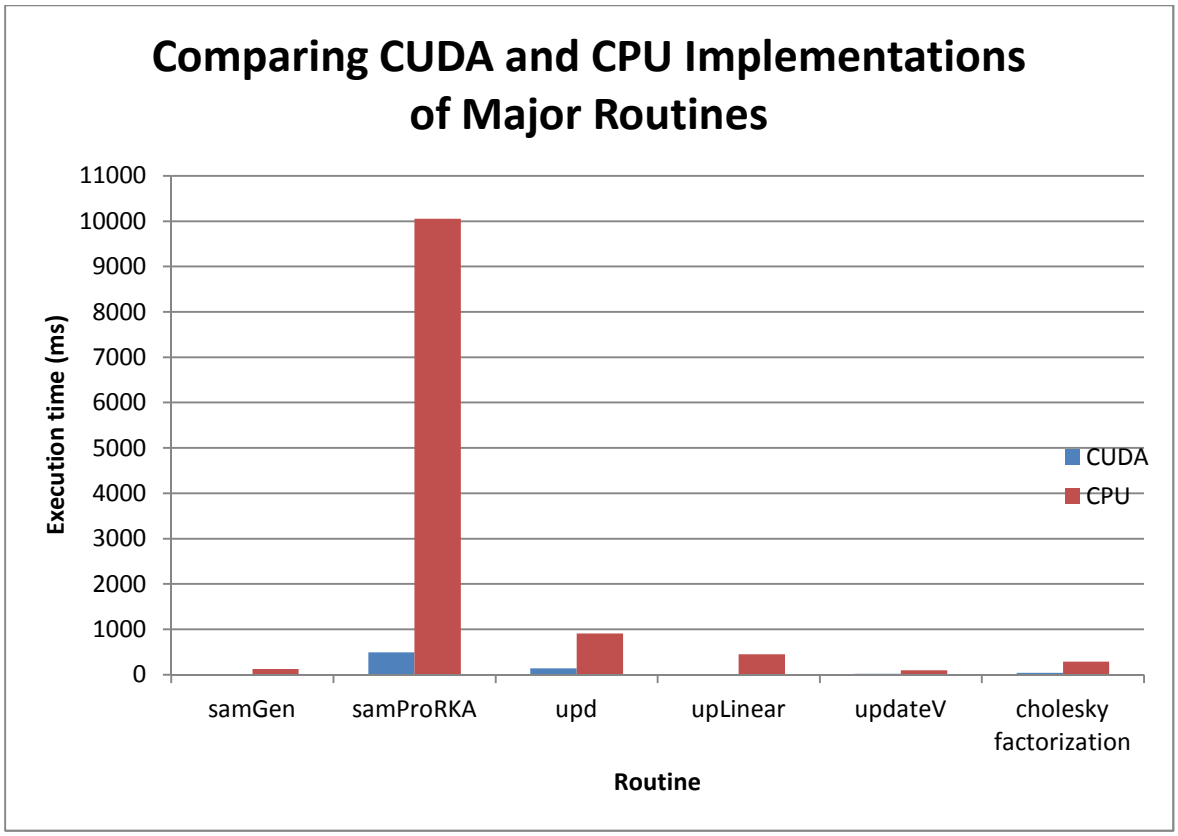


Figure 33: Comparison of the execution times of CUDA accelerated routines with their equivalent CPU implementations on System A.

The work completed in this thesis can be continued in a number of ways. First, multiple GPUs can be used rather than one. However, communicating data between the GPUs might be an issue. Also rather than using GPUs as the means of accelerating NTEPI, a different technology can be investigated. For instance, a computer cluster can be used with MPI (Message Passing Interface) and Scalable LAPACK (ScaLAPACK). ScaLAPACK is an enhanced version of LAPACK for use on parallel distributed memory machines [28]. Another possible direction would be a Field Programmable Gate Array (FPGA) implementation. The function *samProRKA* can be implemented in hardware. This function would be perfect for hardware because there are only two matrices that need to be copied to and from the FPGA during execution. In addition, this function is responsible for about 90% of the execution time of the CUDA accelerated NTEPI. Perhaps a solution to gain the highest performance would combine all implementations into a heterogeneous architecture. The best architecture would be used for each operation.

Bibliography

[1] "Nvidia CUDA C Programming Guide," 5 6 2011. [Online]. Available: http://www.nvidia.com. [Accessed 1 6 2011].

[2] J. Sanders and E. Kandrot, CUDA BY EXAMPLE: An Introduction to General Purpose GPU Programming, Upper Saddle River: Addison-Wesley, 2011.

[3] Khronos Group, "OpenCL Reference Pages," 2011. [Online]. Available: http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/. [Accessed 12 4 2011].

[4] D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Amsterdam: Elsevier Inc, 2010.

[5] Nvidia, "NVIDIA's Next Generation CUDA Compute Architecture," 2009.

[6] S. Rennich, "CUDA Concurrency & Streams," [Online]. Available: http://developer.nvidia.com/gpu-computing-webinars. [Accessed 5 3 2012].

[7] Nvidia Corporation, "CUDA C Best Practices Guide," 2011. [Online]. Available: http://www.nvidia.com.

[8] E. N. Marieb and K. Hoehn, "The Cardiovascular System: The Heart," in *Human Anatomy & Physiology*, San Francisco, Pearson Benjamin Cummings, 2007, pp. 677-712.

[9] L. Wang, *Personalized Noninvasive Imaging of Volumetric Cardiac Electrophysiology,* Rochester, NY, 2009.

[10] "Heart," 2012. [Online]. Available: http://en.wikipedia.org/wiki/File:ConductionsystemoftheheartwithouttheHeart.png. [Accessed 3 1 2012].

[11] Y. Rudy and J. E. Burnes, "Noninvasive Electrocardiographic Imaging," *Annal of Noninvasive Electrocardiology,* vol. 4, pp. 340-359, 1999.

L. Wang, H. Zhang, K. Wong, H. Liu and P. Shi, " Physiological-Model-

[12] Constrained Noninvasive Reconstruction of Volumetric Myocardial Potentials," *Biomedical Engineering,* pp. 296-315, 2010.

[13] "BLAS Frequently Asked Questions (FAQ)," [Online]. Available: http://www.netlib.org/blas/faq.html. [Accessed 8 3 2012].

[14] "LAPACK - Linear Algebra PACKage," [Online]. Available: http://www.netlib.org/lapack/. [Accessed 9 3 2012].

[15] Nvidia Corporation, "CUDA Toolkit 4.1: CUBLAS Library," 2011. [Online]. Available: http://developer.nvidia.com/cuda-toolkit-41.

[16] "MAGMA," 12 4 2012. [Online]. Available: http://icl.cs.utk.edu/magma/.

[17] AMD, "AMD Core Math Library (ACML)," 2011. [Online]. Available: http://developer.amd.com/libraries/acml/pages/default.aspx. [Accessed 2011].

[18] "ATLAS FAQ," [Online]. Available: http://math-atlas.sourceforge.net/faq.html. [Accessed 2011].

[19] Microsoft, "Understanding Profiling Methods," [Online]. Available: http://msdn.microsoft.com/en-us/library/dd264994.aspx. [Accessed 2011].

[20] H. Ltaief, S. Tomov, R. Nath, P. Du and J. Dongarra, "A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators," *LAPACK Working Note #223.*

[21] "LAPACK 3.4.1: dpotf2.f," [Online]. Available: http://www.netlib.org/lapack/explore-html/dd/d6b/dpotf2_8f_source.html. [Accessed 3 2012].

[22] "LAPACK: SUBROUTINE DPOTRF," [Online]. Available: http://www.netlib.org/lapack/double/dpotrf.f. [Accessed 2 2012].

[23] S. Henry, *Parallelizing Cholesky's decomposition algorithm,* unpublished, 2009.

[24] Nvidia, *Tesla C2050/C2070 GPU Computing Processor: Supercomputing at 1/10th the cost,* 2010.

M. Harris, "Optimizing Parallel Reduction in CUDA," [Online]. Available:

[25] http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf. [Accessed 2012].

[26] "ScaLAPACK — Scalable Linear Algebra PACKage," [Online]. Available: http://www.netlib.org/scalapack/ . [Accessed 4 2012].