GPU-OPTIMIZED MOLECULAR DYNAMICS SIMULATIONS

BY

TYSON J. LIPSCOMB

A Thesis Submitted to the Graduate Faculty of

WAKE FOREST UNIVERSITY GRADUATE SCHOOL OF ARTS AND SCIENCES

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

Computer Science

May 2012

Winston-Salem, North Carolina

Approved By:

Samuel S. Cho, Ph.D., Advisor

David John, Ph.D., Chair

V. Paúl Pauca, Ph.D.
Acknowledgments

I would like to thank my advisor, Dr. Samuel S. Cho, for the guidance and insight he has given me while conducting my graduate research and writing this thesis. He has a strong passion for his work and has been a great encouragement while I have worked with him. Without his input, this thesis would not have been possible.

I would also like to thank Dr. David John and Dr. V. Paül Pauca who were both very helpful and insightful committee members who both helped improve this thesis. They, along with the entire staff and faculty in the Computer Science department at Wake Forest University, have created a friendly environment that fosters learning and scholarship in Computer Science. Their help and support has helped deepen my understanding of Computer Science and has better equipped me to solve the problems that I will face throughout my career.

My parents Tracy and Bonnie Lipscomb, my sister Kelsey, and all of my friends have also been a constant source of support and encouragement and have helped motivate me throughout my time at Wake Forest University. They have all helped shape me into the person I am today, both inside and outside of my academic career.

Lastly, I would like to thank God for salvation through Christ Jesus, for giving my life purpose, and for creating a universe filled with wonder and awe that has been a constant source of amazement for those studying it and all it contains. Without the guidance of His Spirit, I would not be where I am today.
# Table of Contents

Acknowledgments ................................................................................................. ii  
Abstract .................................................................................................................. v  

Chapter 1 History of Supercomputing ................................................................. 1  
1.1 Overview of Supercomputing ........................................................................ 1  
1.2 Supercomputing Milestones ........................................................................... 2  
1.3 Supercomputing Accessibility ....................................................................... 5  

Chapter 2 GPU Technology .................................................................................. 6  
2.1 Computer Graphics Technology and Parallel Computations ......................... 7  
2.2 GPU Computing for General Purpose Programming ..................................... 8  
2.3 General Purpose GPU Programming Languages .......................................... 10  
   2.3.1 CUDA for General Purpose GPU Programming .................................. 11  
   2.3.2 CUDA Libraries .................................................................................. 15  
   2.3.3 GPU Programming Parallel Performance Advantages ......................... 17  
   2.3.4 GPU Programming Bottlenecks .......................................................... 18  

Chapter 3 Floating Point Arithmetic ................................................................. 20  
3.1 Overview of Floating Point Arithmetic ......................................................... 20  
3.2 Accuracy of Floating Point Arithmetic ......................................................... 22  
3.3 Floating Point Arithmetic and CUDA ........................................................... 24  
3.4 Performance of Floating Point Arithmetic on CUDA GPUs ....................... 25  

Chapter 4 Molecular Dynamics ........................................................................... 27  
4.1 Biomolecular Folding .................................................................................... 27  
4.2 Molecular Dynamics Algorithm .................................................................... 28  
4.3 Coarse-Grained MD Simulations ................................................................... 31  
4.4 Coarse-grained Simulation Models ............................................................... 32  
4.5 Verlet Neighbor List Algorithm ..................................................................... 33  
4.6 Molecular Dynamics and GPUs .................................................................... 34
Abstract

Tyson J. Lipscomb

Protein and RNA biomolecular folding and assembly problems have important applications because misfolding events are associated with diseases like Alzheimer’s and Parkinson’s. However, simulating biologically relevant sized biomolecules on timescales that correspond to biological functions is an extraordinary challenge due to computational bottlenecks that are mainly involved in force calculations. We briefly review the molecular dynamics algorithm and highlight the main bottlenecks, which involve the calculation of the forces that interact between its substituent particles. We then present novel GPU-specific performance optimization techniques for MD simulations, including 1) a new Verlet-type neighbor list algorithm that is readily implemented using the CUDPP library and 2) data type compression scheme, as well as standard GPU-optimization techniques such as parallel random number generator and floating point operation issues. These and other GPU performance optimizations were applied to coarse-grained MD simulations of the ribosome, a protein-RNA molecular machine for protein synthesis composed of 10,219 residues and nucleotides. We observe a size-dependent speedup of the simulation code with over 32x speedup over the CPU-optimized approach for the full ribosome when all optimizations are taken into account.
Chapter 1: History of Supercomputing

Since the creation of the electronic computer, computer engineers have been developing increasingly more powerful computers to perform advanced numerical calculations that give insight into various fields, ranging from mathematics to physics to chemistry to economics. This can entail complex computations for solving a mathematical problem, such as the multiplication of two matrices, or simulating as physical process, such as the orbit of planets around a star. These types of calculations can greatly vary in complexity and time required to execute, which are often related to factors such as processor speed and memory size. In general, the amount of time required to perform a calculation is proportional to the computational power of the computer performing the calculation, measured in the number of calculations a computer can perform in a given amount of time. Though a computer with a small amount of computing power can solve the same problems as a more powerful one, a larger amount of time will be required. Some computational tasks are so complex that attempting to solve them on even advanced computers can be very impractical due to the amount of time required to complete them. As computers become more powerful, a wider range of problems can be solved in a reasonable amount of time.

1.1 Overview of Supercomputing

The study of supercomputing has led to the development of many computers that have more computational power than would have been imagined possible several decades ago by taking advantage of the most powerful computing hardware available. Measuring the power of a computer is not trivial. Though two computers may use the same number of processors running at identical speeds, they may have varying per-
formance characteristics due to different processor architectures. A common measure used to reliably compare the power of two different computers is to calculate the number of floating point operations it can perform in one second, often abbreviated as “FLOPS” (floating point operations per second) and prefixed with the common metric prefixes kilo-, mega-, etc., when dealing with large values. Measuring a computer’s performance in FLOPS allows the power of two computers to be accurately compared without analyzing the individual components of each machine.

1.2 Supercomputing Milestones

The first general purpose Turing-complete electronic supercomputer ever built was ENIAC, constructed for the United States Army in 1946[58]. Like most supercomputers, ENIAC took up a large amount of space, covering a total of 1,800 square feet[9]. Once completed, ENIAC cost a total of roughly $500,000 in 1946[9] (equivalent to ~$6 million today) and could run at up to 300 FLOPS[9]. The Army used ENIAC to successfully perform calculations that would have been very time consuming or difficult to solve with the technology available in the mid-1940s. Two such successes were calculating artillery trajectory tables[9]—a task that would have taken a great deal of time and resources otherwise—and performing simulations related to the creation of the hydrogen bomb[29] that would have been nearly impossible without a computer simulation. Though massive and low-powered by today’s standards, ENIAC was nonetheless crucial in the development of computing technology by demonstrating how valuable an electronic computer could be in various situations.

In the decades following the introduction of ENIAC, computing technology increased drastically and moved from the realm of government agencies to the home. ENIAC and many other early supercomputers were constructed out of glass vacuum tubes which took up a great deal of space in comparison to modern silicon-based tran-
sistors. The transition from light bulb-sized vacuum tubes to microscopic transistors brought about by advances in lithography led to computers transitioning from the size of warehouses to the size of a handheld device. Technology began advancing at a rate that allowed the number of transistors that could fit into a given space to roughly double every two years, leading to exponential increases in memory sizes and processor performance, a trend described by Moore’s Law[32, 33]. As consumer computer hardware became more powerful and affordable, computer engineers began constructing powerful supercomputers from components consisting mostly of consumer-grade parts instead of custom-made hardware[30]. Supercomputers constructed in this manner often consisted of a large collection of individual computers networked together and running an operating system that would allow them to coordinate their computations and perform as if they were a single, unified machine. This approach was often much cheaper and easier to implement than other approaches using large amounts of specialized hardware. With the cost and difficulty of creating a powerful supercomputer decreased, many more high performance supercomputers were constructed.

One major milestone in the progression of supercomputing using off-the-shelf parts was ASCI Red. Completed in 1997 and utilizing a total of 9,298 Intel Pentium Pro processors[7], ASCI Red was the first computer capable of TeraFLOPS computing—performing one trillion floating point operations per second[7]. Though still roughly the size of ENIAC, ASCI Red achieved levels of performance that far exceeded those of its predecessor. The biggest difference between ENIAC and ASCI Red—aside from the enormous performance difference—was that ENIAC used a collection of components that were custom made, whereas many of the components of ASCI Red were already in use by many personal computers and could be obtained and used without enlisting a large team of electrical and computer engineers to design and build them.
Most supercomputers have been constructed in a similar manner to ASCI Red, relying on central processing units (CPUs) to perform the majority of their computations. However, recent advances in the power and programability of graphics processing units (GPUs) have outpaced those of CPUs, as shown in Figure 1.1. These advances have led many to begin using GPUs alongside of CPUs in certain “hybrid” applications that harness the strengths of both types of processors, such as China’s Tianhe-1A supercomputer which became the fastest computer in the world in late 2010[53, 54], relying on 7,168 GPUs working alongside 14,336 CPUs to reach a computing power of 2.507 PetaFLOPS ($2.507 \times 10^{15}$ FLOPS)[39]. The designers of many other supercomputers are similarly relying on GPUs to create supercomputers that are more capable and powerful than those based solely on CPUs[41].

![Comparison of CPU and GPU Performance](image)

Figure 1.1: Comparison of CPU and GPU Performance, as shown in [38]
1.3 Supercomputing Accessibility

Though supercomputers are very powerful machines, using them can often involve overcoming various barriers that are inherent in the way they are constructed. When writing a program for a massively parallel computer, the programmer may have to overcome various problems stemming from the way the computer is constructed. In a traditional single-thread application, the programmer does not need to worry about synchronizing a variety of threads or moving data between different computing nodes.

With supercomputers, however, the program must be designed in such a way as to fully utilize the distributed nature of the machine. Since supercomputers utilize a distributed network of different nodes, programs are written to be parallel to maximize the available computational resources and minimize idle nodes. Additionally, if a processor in one node needs to access a piece of data that resides on another node, care must be taken that a large amount of overhead is not introduced when the first node waits for the data to transfer.

Though commonly used parallel processing libraries such as MPI[34] and OpenMPI[42] exist to make distributed, parallel programs easier to write, gaining access to a state of the art supercomputer can still be difficult for some. Many of the organizations that maintain the world’s fastest supercomputers also charge a fee to use them for a period of time, which creates a financial obstacle that researchers must overcome in order to perform their work. In most cases, one must be a well-established researcher or scientist to gain access to these supercomputing facilities.

Therefore, supercomputing has a relatively high cost of entry, both in financial terms and in time spent developing code. If both of these barriers were decreased, supercomputing would be more widely accessible for many seeking to solve complex and computationally expensive problems.
Chapter 2: GPU Technology

In the 1980s, hardware companies began producing dedicated accelerator hardware components to handle the manipulation and display of 2D graphics in order to offload computationally complex graphical calculations from the CPU. These “graphics cards” differed from multipurpose CPUs by focusing solely on the generation of computer graphics, sacrificing computational flexibility for higher graphical performance. As computing continued to advance, programmers began creating computer programs that were able to manipulate 3D models for display on a standard monitor. These advances put even more emphasis on the need for dedicated graphics hardware and the introduction of 3D video games created a consumer-level market for powerful graphics cards.

Over time, the competition between different vendors led to large improvements in graphics cards that resulted in increased performance as well as decreased cost. Graphics cards began implementing more graphical functions in hardware and, with the introduction of the GeForce 256 in 1999, NVIDIA introduced the GPU to computing, defining a GPU as “a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second” [37]. The GeForce 256 was the first in a long line of GPUs that would be produced by companies such as NVIDIA and ATI (which later merged with the company AMD) that could process graphics much more quickly and efficiently than CPUs developed in the same time period.
2.1 Computer Graphics Technology and Parallel Computations

One of the aspects of computer graphics that greatly influenced the different design approaches taken by the developers of GPUs is the inherent parallel nature of displaying computer graphics. When creating a 3D image for display, a GPU performs a variety of computations on a set of graphics primitives that are highly data independent. A computer representation of a 3D scene is composed of a set of points known as vertices that comprise a variety of 2D primitives, most commonly triangles, that make up a collection of 3D figures[6]. To display this 3D scene, a GPU considers the set of all primitives as independent structures and computes various properties such as lighting and visibility, performing each operation independently of the various other primitives in the scene[30]. The overwhelming majority of these computations are floating point operations[30], so GPUs are optimized primarily for performing these types of computations.

Since a GPU will focus primarily on performing a relatively small set of operations on a specific set of data points, GPU makers focus on creating hardware that specializes in these tasks instead of a wide array of different tasks, as is the case with CPU designs. This places a limitation on the flexibility of GPUs, but allows them to perform their specialized tasks much more efficiently. Therefore, GPUs focus on the utilization of a large number of parallel processors dedicated to performing graphical computations, whereas CPUs focus more on performing a much smaller number of generic calculations very quickly[30]. Though CPUs have become more parallel in recent years, they still do not compare to the parallelism of current generation GPUs. The different approaches to computing used by CPUs and GPUs has led CPU manufacturers to generally favor processor speed over parallelism, whereas GPU manufacturers generally favor parallelism over processor speed (Table 2.1). Though a
single core of a CPU may be many times more powerful than a single GPU core, the large number of cores on a GPU makes them better suited to many types of highly parallel tasks.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel i7-950</th>
<th>AMD Opteron 6274</th>
<th>NVIDIA C2070</th>
<th>NVIDIA 580GTX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>CPU</td>
<td>CPU</td>
<td>GPU</td>
<td>GPU</td>
</tr>
<tr>
<td>Cores</td>
<td>4</td>
<td>16</td>
<td>448</td>
<td>512</td>
</tr>
<tr>
<td>Speed</td>
<td>3.06 GHz</td>
<td>2.2 GHz</td>
<td>1.15 GHz</td>
<td>1.54 GHz</td>
</tr>
</tbody>
</table>

Table 2.1: Processor cores and speed for several CPUs and GPUs

Traditionally, GPUs have relied on a collection of single-purpose components that each specialize in performing a different task in the process of generating an image. Together these units make up what is known as a “graphics pipeline”[30]. The task of processing graphical information can be neatly divided into a set of independent tasks and when an image is to be displayed, the data relating to the image is passed through this pipeline, processed, and finally displayed on the computer screen. By creating specialized, efficient hardware, GPU developers were able to create powerful single-purpose GPUs at the expense of flexibility.

2.2 GPU Computing for General Purpose Programming

As successive generations of GPUs continued to become more powerful, it became clear that the large amount of computational power in GPUs had great potential for general purpose computing. Many types of problems that are often solved on supercomputers have calculations that can be done independently using floating point operations. Thus, there is a strong motivation for writing special-purpose programs that could utilize the parallel nature of GPUs[11]. In order to use a GPU for general purpose calculations, however, a calculation must be translated into instructions that a GPU can process. This required rephrasing a computation in terms of vertices,
textures, and 3D primitives. Though not impossible, writing a non-graphical program to run on a GPU was much more difficult than writing an equivalent one to run on a CPU[30].

Though GPU manufacturers were not yet designing GPUs specifically for general purpose applications, the increasing complexity of graphics in video games were motivating them to create GPUs with more programmable processing units that would allow video game designers to write their own custom shaders—portions of code used to calculate different properties of a 3D scene. Various portions of the rendering pipeline were changed from fixed-function units with hardwired algorithms to more flexible units that could perform custom computations[30]. Game designers were now able to write custom shaders instead of being bound to the shader implementations that manufacturers built into their hardware. Though these improvements were implemented because of demand from game designers, they were also welcome additions for researchers wanting to use GPUs for a wider array of applications. The course of the industry was shifting from creating GPUs that could only perform a limited set of functions to producing GPUs with a larger degree of flexibility and programability.

The introduction of Microsoft’s DirectX 10 API broke away from traditional pipeline approaches to programming computer graphics and presented the programmer with a collection of unified shaders instead of a pipeline of several different types of special purpose shaders[10, 30]. One of the first GPUs released that took advantage of the DirectX 10 API was NVIDIA’s GeForce 8800 which was based on a unified architecture consisting of a large collection of unified shaders that could be programmed with a large degree of flexibility[45, 47]. Generating computer graphics was done by programming the unified shaders to perform the tasks that were previously performed by different successive pieces of the graphical pipeline. The difference was that instead of passing the data from one hardware component to another, the data was
cycled through the same unified shaders multiple times, with the shaders performing a different function on each pass. This new approach to graphics rendering focused GPU manufacturers on creating massively parallel, highly programmable hardware that was well suited to general purpose GPU computations. The graphics APIs were still the main way of interfacing with GPUs, so it was clear that new software would need to be designed to allow programmers to more easily tap into the computational power of GPUs without requiring them to rewrite their algorithms in terms of graphical computations.

2.3 General Purpose GPU Programming Languages

With the introduction of generic, unified GPU architectures, GPU manufacturers NVIDIA and AMD began working on providing programmers with the ability to program their GPUs to be used for a variety of purposes outside of graphical applications. Programming GPUs through graphics APIs was possible, but required programmers and algorithm designers to approach their problems in drastically different ways. Several software packages have been released whose aim is creating a general, programmer-friendly way to program GPUs for general purpose applications.

One of the first implementations of a general purpose GPU programming language was Brook, developed by researchers at Stanford University[11]. The goal of Brook was to provide programmers with an extension to the C programming language that would allow accessing GPUs for use in non-graphical computations. Though the hardware used by the Brook developers was not as well suited for general purpose computations as modern GPUs, programs written for GPUs using Brook were able to run up to eight times faster than similar code written for CPUs[11].

Impressed with the advances that the Brook project made in the area of general purpose GPU computing, NVIDIA hired Ian Buck, one of the lead researchers in the
Brook project, to begin development on their Compute Unified Device Architecture (CUDA) library[15], which allows programmers to write C/C++ applications that can execute portions of parallel code on NVIDIA GPUs. ATI similarly started their own initiative known as Stream[3, 59].

Though CUDA is designed to run only on NVIDIA hardware and ATI’s Stream was originally focused on ATI hardware, the OpenCL project[23] aimed to create a programming language that would allow code to be run on hardware—both GPUs and CPUs—from different vendors. It is the most common framework for writing code that can be ran on multiple brands of hardware. The OpenCL language was first proposed by Apple and has been adopted by Intel, AMD and NVIDIA, ensuring that a wide array of hardware will be supported[22]. OpenCL is based on C/C++ but has several restrictions[24]. For this reason, a C/C++ program cannot be compiled by the OpenCL toolchain, which could limit the ease of adaptability of existing code.

For the remainder of the thesis, we will focus on CUDA, the GPU language we used for our study.

2.3.1 CUDA for General Purpose GPU Programming

NVIDIA’s CUDA library provides programmers with a collection of language constructs that allow parallel code to be written for NVIDIA GPUs without the need to learn many new programming methods. One of NVIDIA’s main goals in the creation of CUDA was to allow programmers with a background in common programming languages to be able to access NVIDIA GPUs with as much ease as possible. CUDA uses a heterogeneous computational environment, meaning that two types of processors, CPUs and GPUs, are used during program execution, allowing a programmer to utilize the strengths of both types of processors[38]. A CUDA program can therefore be thought of as a standard CPU program with library functions that offload certain
computations to the GPU.

One of the largest differences between writing a CUDA program and writing a standard C/C++ program is creating and launching “kernels”—the CUDA equivalent of a parallel function call. Calling a CUDA kernel is very similar to calling a C/C++ function, but when a kernel is called it executes in parallel on the GPU and returns control to the CPU immediately. The kernel call creates a collection of parallel subprocesses on the GPU called “threads”. Each thread performs one part of a larger computational task independently of all of the other threads running on the GPU.

The threads created by a kernel call are arranged on the GPU in a multilevel hierarchy that can be configured to fully utilize the hardware of any CUDA-capable GPU. All of the threads created by a kernel are arranged into different groups of one, two, or three dimensional arrays of threads known as “blocks” that will execute independently on different processors within a GPU. The blocks that are created by a kernel call can be thought of as modular subdivisions of a computation that are executed independently and in arbitrary order. All of the blocks within a kernel are arranged into a one or two dimensional collection of blocks called a “grid” (Figure 2.1).

On a given GPU there are only a finite number of blocks that can execute at the same time. If the GPU has a small number of computational cores, a small subset of the blocks will execute simultaneously and, when they are completed, a second set of blocks will execute. This process will continue until all of the blocks have finished execution. On the other hand, if a GPU has a large number of computational cores, it is possible that all of the blocks will be able to execute simultaneously and will be able to complete at about the same time. This method of arranging blocks allows a compiled program to be flexible enough to adapt itself to GPUs with various numbers
of processor cores automatically (Figure 2.2).

Memory distribution within a CUDA program is arranged in a similar hierarchical manner. At the highest level is grid-level memory consisting of general purpose read/write global memory, read-only constant memory, and read-only texture memory. Every thread and block in a program can access this level of memory and it is in many ways analogous to RAM in a CPU computing environment. Additionally, each block has access to shared memory that is many times faster than global memory. A block’s shared memory is available to every thread within a block and is meant to be used as a type of cache for operations that would otherwise require numerous redundant reads or writes to global memory or for operations from different threads that access the same portions of global memory. Shared memory is persistent for the entire lifetime of a block and is automatically released when a block finishes execution. At the lowest level, each CUDA thread possesses a relatively small amount of private memory that is only accessible by that thread. Not unlike L1 processor cache,
private thread memory is both the fastest and most limited type of memory available to CUDA hardware.

The execution model implemented by NVIDIA GPUs is a generalization of the well-known single instruction, multiple data (SIMD) model known as single instruction, multiple threads (SIMT). In SIMD computer hardware, a set of computational units, such as processor cores, access different pieces of data and perform the exact instructions on their respective pieces of data. SIMD architectures are well suited to massively parallel tasks, such as matrix multiplication, and simplify the process of synchronization by allowing parallel hardware components to automatically step through identical portions of code in synchrony and complete their actions simultaneously. A common approach used when writing SIMD programs is to assign each processor to a different portion of memory and then have each processor execute an
operation on that piece of data. For example, in a vector addition computation where vectors \( A \) and \( B \) were added and stored in a vector \( C \), each core \( i \) would add the elements \( A[i] \) and \( B[i] \) and store the result in \( C[i] \). Each of the \( N \) cores would read from contiguous locations in memory and likewise write to contiguous memory locations. Once the cores have finished executing their parallel instructions, the program will have all of its processors synchronized. By comparison, in a single instruction, single data (SISD) programming model, each thread would need to be synchronized at the end of the execution, adding computational overhead to the computations. To the programmer, the SIMT approach used by CUDA functions identically to a SIMD application, meaning that no further understanding is needed in order to utilize CUDA.

CUDA also implements barrier synchronization, a necessity in any parallel computing environment, by allowing a programmer to halt CPU execution and wait for a specified kernel call to complete at any point in the program. This allows a programmer to configure a program to either halt the CPU portion of an application entirely and wait for the GPU to finish its calculations or to continue executing CPU code while the GPU executes a kernel and synchronize with the GPU at a later time. Different applications will require different synchronization patterns and CUDA affords a great deal of flexibility to the programmer in this regard. Without a fully flexible barrier synchronization scheme in place, many applications could not be fully optimized for a heterogeneous computing environment.

### 2.3.2 CUDA Libraries

On top of existing CUDA functions for executing instructions on the GPU, there exist highly optimized CUDA libraries that have been created in order to provide programmers with easy access to functionality that is commonly needed when writing programs. These libraries provide efficiently implemented algorithms and allow
more development time to be focused on designing and implementing other parts of a program. Two popular optimized CUDA libraries that we used in our research were the CUDA Data Parallel Primitives (CUDPP) and the CUDA Random Number Generator (cuRAND) libraries, which we briefly describe here.

The CUDPP library[13] was originally created to test the implementation of several parallel algorithms on CUDA-compatible GPUs and has expanded to include a wide range of efficient parallel algorithms, including radix sort and parallel scan that are highly complex and would require a large amount of time to implement from scratch.

The cuRAND library[36], which is included with recent releases of the CUDA SDK, provides functionality to generate random numbers for use either on CPUs or GPUs and includes multiple algorithms for random number generation as well as a variety of different distribution types[35]. Without GPU-based random number generators (RNGs), programmers would be required to generate random numbers on a CPU and then transfer them to the GPU. This approach would be inefficient because it would either require a transfer to take place every time a kernel needed a set of random numbers or require storing a large number of random numbers in memory before they were needed by a CUDA kernel. A GPU RNG library, however, allows random numbers to be generated on demand without any kind of intervention from a CPU. Additionally, cuRAND can generate random numbers for use in CPU code many times faster than a CPU-based RNG. Implementing a GPU-based RNG would be a complex and error-prone task for many programmers, so cuRAND provides efficient functionality that would be out of reach for a large majority of programmers.
2.3.3 GPU Programming Parallel Performance Advantages

The primary advantage of using GPUs for general purpose computing is their inherently parallel nature. Traditional graphics processing approaches required that a minimal amount of overhead be introduced when processing data. Every graphics rendering scenario would necessarily be a parallel process, so designers of GPUs have always focused on ways to set up multiple concurrent operations with the lowest computational cost possible. The low overhead involved with performing parallel graphics processes also allows GPUs used in non-graphical applications to perform very well when a large number of parallel computations are needed. By comparison, CPUs often require much more overhead when invoking a large number of parallel processes[30]. Using GPUs instead of CPUs for parallel computations therefore reduces performance bottlenecks due to the overhead involved in setting up massively parallel computations.

Parallelism in GPUs is also much easier to manage than with CPUs. Thread scheduling and synchronization are nontrivial issues with CPUs, but GPUs are designed to handle these problems natively in hardware, reducing both the amount of time a programmer must spend performing these tasks and the number of clock cycles required to manage parallelism. With GPUs, individual threads are not managed by the programmer and a large number of identical threads are most commonly launched in parallel to operate on a collection of data with only a few function calls.

The native parallelism in GPUs allows them to scale without major architectural hardware changes. The computing power of a GPU can be doubled by simply adding twice as many computational cores and a small number of circuits to manage issuing commands to the new cores. By contrast, the addition of more cores to a CPU is often times a more involved process due to their more intricate, multipurpose nature. The relative ease with which GPU cores can be added has led to the development
of GPUs with hundreds of computational cores, allowing an equally large number of threads to be executed concurrently. Though the computational cores on a GPU are not as powerful as those on a CPU in terms of computational rates, the large number of cores more than makes up for the performance difference, allowing GPUs to have much higher computational throughput than CPUs.

2.3.4 GPU Programming Bottlenecks

Though GPUs are well suited to solving certain types of problems, writing efficient code for them is not without its difficulties. One of the largest bottlenecks that is introduced into any parallel code utilizing a GPU is the overhead involved in transferring data between the CPU and GPU and vise versa across a PCI Express bus. Once a program loads information into RAM, whether from the hard drive or from another input source, the CPU is able to access it very quickly without much concern about how long it takes to access this data. With GPUs, however, any data needed by a GPU during a computation must be transferred before it can be accessed by the GPU. During this transfer the GPU will not be able to work on the data and could sit idle for an extended period of time. Depending on the size and frequency of transfers needed in a computation, this can create a bottleneck that can significantly impact the performance of an algorithm.

Also, even though the parallel nature of GPUs makes them efficient at processing large amounts of data in parallel, serial portions of the algorithms will not benefit from implementation on a GPU. Most modern GPUs feature hundreds of computational cores, so if a computation utilizes only a single core in a serial computation, all but one of the GPU cores will sit idle, leading to a very low utilization of resources. Tasks that run in parallel, by comparison, can utilize the entire GPU if written correctly, but a high degree of interdependence between threads can drastically decrease the
performance of a program if it relies on that portion of the algorithm for a large number of operations or a single operation that takes up a large portion of the execution time.
Chapter 3: Floating Point Arithmetic

When performing numerical computations on a computer, there are two ways to represent the numbers involved in the calculations. One approach is to treat each number as an integer value, meaning that it will be treated as a whole number with no decimal or fractional portion. Alternatively, if data that cannot be accurately represented by integers is being processed, a floating point representation, i.e., a representation using a whole number and a decimal point, can be used.

Integer and floating point values allow programmers to choose between either accuracy or precision when performing calculations. When using integer values, numbers can be represented exactly, but the range of numbers that can be represented is limited to only whole numbers. Alternatively, floating point numbers are able to represent a wider range of values, but this often comes at the cost of reduced accuracy[50]. Even though floating point numbers can represent a larger range of numbers, it is not possible to represent any given numerical value using a floating point number that takes up only a finite amount of space.

3.1 Overview of Floating Point Arithmetic

The IEEE 754 standard[49] defines a method for storing numbers in binary floating point format using three pieces of information: a sign $s$, a fractional portion $f$, and an exponent $e$, each represented by a series of bits. The fractional portion is used to represent a binary fractional number $1.f$ and the exponent is used to represent a biased exponent term $2^{(e-b)}$, where $b$ is used to allow representation of exponents with a value less than 0. The value of the number stored in this format is defined as

$$(-1)^s \times 1.f \times 2^{(e-b)}$$
A single precision and double precision format are defined by the IEEE 754 standard for representing floating point numbers on computers that correspond to 32- and 64-bit values, respectively. Single precision numbers are only half the size of double precision numbers, but they are more limited in the range of numbers that can be represented and the precision with which those numbers can be represented. The single precision format uses 8 bits for the exponent with a bias term of 127 and a total of 23 bits for the fractional portion of the number. Including the leading 1 in the 1.f fraction, single precision numbers have a total of 24 bits to represent significant digits and can scale this value by any power of 2 in the range $2^{-126}$ to $2^{127}$. (Exponents of -127 and 128 are reserved for use in representing special values such as 0, -Inf and +Inf.) The double precision format, by comparison, uses 11 bits for the exponent with a bias term of 1023 (with exponents of -1023 and 1024 reserved for special values) and 52 bits for the fractional portion of the number. This allows for over twice as many significant digits as the single precision format and allows much larger and much smaller numbers to be represented, on the scale of $2^{-1022}$ and $2^{1023}$.

The different properties of single and double precision IEEE 754 numbers allow programmers to choose which format is best for a given application based on required precision, hardware support and the amount of available storage or memory. If an application requires a high degree of precision, the double precision format will likely be the best choice to use. However, if an application does not require a high degree of precision and will be running on hardware that does not natively support double precision arithmetic, the performance of the program may be adversely affected by using double precision data types.
3.2 Accuracy of Floating Point Arithmetic

When storing and manipulating numbers on a computer, the finite number of bits that can be allocated to each number leads to the necessity of rounding numbers when they cannot be adequately represented by a floating point number using a finite number of bits. The IEEE 754 defines several different rounding modes, but suggests that the “round to nearest” mode be used. As the name implies, this rounding mode produces results that are rounded so that they are closest to the true value of a calculation. Although these results will be the most accurate possible using limited precision numbers, it is still important to be mindful of the fact that a small amount of accuracy will still be lost in some calculations. Though the loss of accuracy may go unnoticed in most every-day applications, when performing scientific computations that involve a large number of very precise floating point calculations, it is possible that these small inaccuracies can add up in a way that could render the calculation completely inaccurate[50].

Further complicating the issue is the fact that when performing a series of floating point calculations, the order of operations can be very important to the outcome of the entire calculation and can lead to different results depending on the order of operations—even if these operations are associative in mathematics[57]. For example, in mathematics, the law of associativity states that \((A + B) + C = A + (B + C)\). However, this property does not always hold when performing floating point operations on computers. This is primarily due to the rounding operations that take place between each mathematical operation.

When performing floating point calculations, most modern processors use higher precision registers that included guard bits—extra bits for representing numbers of higher precision—that allow mathematical operations to be performed and represented with more precision[50]. When the result is converted back to a floating point
number for storage, however, these extra bits must be discarded, at which point a rounding operation takes place. In the operation \((A + B) + C\), the result of \((A + B)\) will be rounded before it is added to \(C\), sacrificing some accuracy in the process. Likewise, when calculating \(A + (B + C)\), the result of \((B + C)\) will be rounded before it is added to \(A\). To demonstrate this, observe the following example, given in [57], where \(\text{rn}(x)\) represents the rounding operation performed after each step using single precision numbers:

\[
A = 2^1 \times 1.0000000000000000000001
\]
\[
B = 2^0 \times 1.0000000000000000000001
\]
\[
C = 2^3 \times 1.0000000000000000000001
\]

\[
A + B = 2^1 \times 1.100000000000000000000110000... \\
\text{rn}(A + B) = 2^1 \times 1.10000000000000000000010
\]

\[
B + C = 2^3 \times 1.001000000000000000000100100... \\
\text{rn}(B + C) = 2^3 \times 1.0010000000000000000001
\]

\[
A + B + C = 2^3 \times 1.0110000000000000000000101100... \\
\text{rn}(\text{rn}(A + B) + C) = 2^3 \times 1.01100000000000000000010 \\
\text{rn}(A + \text{rn}(B + C)) = 2^3 \times 1.0110000000000000000001
\]

Though the results given by \(\text{rn}(\text{rn}(A + B) + C)\) and \(\text{rn}(A + \text{rn}(B + C))\) are rounded properly and considered “correct” by the IEEE 754 standard, they are two different numerical values. Situations like this one can arise many times throughout a mathematically intensive computation and can cause results to differ slightly from calculation to calculation depending on whether or not the operations are performed in an identical manner each time.
3.3 Floating Point Arithmetic and CUDA

When writing a program to run on a GPU, the mathematical operations that are performed will be done in an environment that is much different from that of the CPU, which requires programmers to take care to ensure that operations produce accurate results. Though the current generation of NVIDIA GPUs are IEEE 754 compliant with both single and double precision floating point numbers, previous generations of hardware were not. The features that are supported by a GPU can be determined by its Compute Capability version, consisting of both a major and a minor revision number[38]. GPUs that are Compute Capability 1.2 or lower do not support IEEE 754 double precision numbers and are not fully IEEE 754-compliant with some single precision floating point operations. However, Compute Capability 1.3 brought the introduction of IEEE 754-compliant double precision support, but did not improve the single precision support of Compute Capability 1.2 and lower devices. Full IEEE 754 support with both single and double precision was introduced in Compute Capability 2.0 devices and has been implemented in all devices since these were introduced[38].

Even though the current generation of CUDA-capable GPUs support IEEE 754 floating point arithmetic, a program written for a CUDA GPU will often produce results that differ from the ones generated by the CPU in an equivalent calculation. It is also likely that numerical results produced by one GPU program run will be different than those of the same program during a different run. The main reason for this discrepancy is the way that kernels are executed in CUDA. When a grid of blocks are created, the order of their execution is undefined and will therefore vary between successive program executions. If different blocks perform calculations on overlapping portions of data, the order of those operations will therefore be undefined and can lead to slightly different results across multiple runs. This will often
lead to differences between the results of a CPU-based calculation and a GPU-based one under certain circumstances. In a single threaded CPU program, the order in which different operations will take place are well defined, which will in general cause its output to be identical after every run. The GPU-based calculations are not guaranteed to execute in any particular order, which will often lead to small variations in the mathematical output.

Additionally, current generation CUDA GPUs support the use of fused multiply-add (FMA) operations, which perform a multiply operation on two floating point numbers and then add the result to a third floating point number without performing a rounding operation between each operation. The result is that the calculation will often times be more accurate than if the two operations were done with a rounding operation between each step[57]. There are currently no x86 CPUs that support FMA operations, so comparing the results of computations done in GPUs and CPUs could be complicated further.

### 3.4 Performance of Floating Point Arithmetic on CUDA GPUs

Although the current generation of CUDA GPUs are fully IEEE 754-compliant, there are still some performance issues that should be noted. The Fermi CUDA architecture can perform double precision calculations at only half the rate at which it can perform single precision calculations[40]. Furthermore, the consumer-grade Fermi GeForce GPUs are limited to even less throughput double precision throughput.

For applications that require accuracy greater than what can be given by single precision arithmetic, it is necessary that double precision numbers are used. Neglecting to do so can lead to differences in the accuracy of the final results of a calculation that would render the computation invalid. The use of double precision numbers in
a GPU computing environment will degrade performance, but this may be necessary for some applications to produce accurate results.

Furthermore, the amount of performance that is lost may not necessarily be identical to the performance loss when performing a double precision operation. In an application that has more integer operations than double precision operations, for example, the performance difference between applications using single and double precision floating point numbers may be much smaller. The accuracy and performance tradeoff must therefore be assessed for each application and will vary from case to case.
Chapter 4: Molecular Dynamics

One important application of general purpose GPUs is the study of protein and RNA folding using molecular dynamics (MD) simulations. Within every living cell there is a complex arrangement of molecular machines known as biomolecules that are each tasked with performing important functions related to the function of the cell[1]. They include enzyme catalysis, structural support, transport, and regulation, just to name a few[20]. All cellular processes occur because of biomolecular functions, and many of these functions are determined by the way that individual proteins or RNA molecules form, assemble, and manipulate their structures[5, 18].

We focus on two classes of biomolecules called proteins and RNA (ribonucleic acids). Proteins are composed of 20 naturally occurring types of amino acids that are attached in a linear fashion. On average, a protein is composed of about 400 amino acids, and they can interact with other proteins to form complex structures. Similarly, RNA are composed of 4 naturally occurring types of nucleic acids that are also attached linearly, and they can interact with other RNA or proteins to form complexes. Biomolecular complexes assemble into biomolecular machines such as flagella that allow the cell to move, kinesins that transport cargo across cells, or ribosomes that synthesize proteins.

4.1 Biomolecular Folding

Under normal circumstances, biomolecules will self-assemble in solution into a specific structure to properly perform their cellular functions[5, 18]. However, if a biomolecule does not fold properly, a misfolding event results that can be detrimental to both the cell and the larger organism. It is now understood that many diseases such as
Alzheimer’s and Parkinson’s are caused by problems that occur during the folding processes[17]. It is therefore crucial that the folding process of different biomolecules be understood if these diseases are to be more effectively combatted.

Every biomolecule that undergoes the folding process has both a folded and unfolded state and the folding process consists of many possible pathways by which the biomolecule transitions from one state to another[16, 19, 21, 60]. We note that the study of biomolecular folding can be contrasted with the field of biomolecular structure prediction, which involves studying one state of a biomolecule and using statistical methods to predict what its structure will be after it has transitioned to its folded state[31]. This technique has been very successful in determining the folded structure of biomolecules after they have undergone the folding process, but fails to give any information about the structure of the biomolecule at intermediate points between its folded and unfolded state.

4.2 Molecular Dynamics Algorithm

Experimental techniques can be used to study both the folded and unfolded states of various biomolecules, but are currently unable to directly observe the intermediate states during the folding process. The resolution of the timescales of experiments are inaccessible to folding events, which are on the microsecond timescale for the fastest folding proteins[19]. However, by utilizing molecular dynamics simulations, the entire folding process can be studied by simulating the behavior of a biomolecule as it folds or unfolds and recording the position of each part of the system at predetermined intervals of about one picosecond (10^{-12} seconds)[19]. By collecting a series of sets of positions of beads in a system which correspond to a structure, a trajectory of the folding process can be observed from the beginning to the end.

Molecular dynamics is the study of systems of atomic or molecular structures and
the way they interact and behave in the physical world. The system being simulated can be thought of as a unified description of the individual structures within it. The physical description of the biomolecule in MD simulations is determined by an energy potential. The most basic description of a biomolecule must be an energy potential that includes the short-range connectivity of the individual components through a bond energy term and long-range interactions of the spherical components through attractive and repulsive energy terms. More sophisticated descriptions can include electrostatic charge interactions, solvation interactions, etc. We will describe in detail the Self-Organized Polymer (SOP) model energy potential[27, 43] in the next section. Regardless, the physical description of biomolecules essentially become spherical nodes that are connected by edges that correspond to interactions. The total structure (i.e., the collection of nodes and edges) within a molecular dynamics system can be thought of as an abstraction of a much smaller collection of physical entities (i.e., nodes) that acts as a single, indivisible unit within the system. When conducting a molecular dynamics study of a system, each of the structures within it are often treated as spherical physical beads that make up a larger system.

Once the energy potential is determined, one must determine the rules for moving the biomolecule over time. The molecular dynamics algorithm is as follows: given a set of initial positions and velocities, the gradient of the energy is used to compute the forces acting on each bead, which is then used to compute a new set of positions and velocities by solving Newton’s equations of motion ($\vec{F} = m\vec{a}$) after a time interval $\Delta t$. The process is repeated until a series of sets of positions and velocities (snapshots) results in a trajectory[2].

Since the equations of motion cannot be integrated analytically, many algorithms have been developed to numerically integrate the equations of motion by discretizing time $\Delta t$ and applying a finite difference integration scheme. In our study, we use
the well-known Langevin equation for a generalized $\vec{F} = m\ddot{\vec{r}} = -\zeta \vec{v} + \vec{F}_c + \vec{Γ}$ where $\vec{F}_c = -\frac{\partial \vec{v}}{\partial \vec{r}}$ is the conformational force that is the negative gradient of the potential energy $\vec{v}$ with respect to $\vec{r}$. $\zeta$ is the friction coefficient and $\vec{Γ}$ is the random force.

When the Langevin equation is numerically integrated using the velocity form of the Verlet algorithm, the position of a bead is given by $\vec{r}(t) + \Delta t \vec{v}(t) + \frac{\Delta^2 t}{2} \vec{F}(t)$ where $m$ is the mass of a bead.

Similarly, the velocity after $\Delta t$ is given by

$$V(t + \Delta t) = \left(1 - \frac{\Delta t \zeta}{2m}\right) \left(1 - \frac{\Delta t \zeta}{2m} + \left(\frac{\Delta t \zeta}{2m}\right)^2\right) V(t)$$

$$+ \frac{\Delta t}{2m} \left(1 - \frac{\Delta t \zeta}{2m} + \left(\frac{\Delta t \zeta}{2m}\right)^2\right)$$

The molecular dynamics program first starts with an initial set of coordinates ($\vec{r}$) and a random set of velocities ($\vec{v}$). The above algorithm is repeated until a certain number of timesteps is completed, the ending the simulation.

The length of each timestep will depend on the desired accuracy of the program and will directly impact the amount of time the simulation will take. Molecular dynamics seeks to simulate a continuous process through discrete steps, so some minute details will always be lost, regardless of how accurate the simulation may be. Though this may seem like a major drawback, the discrete nature of computers dictates that any simulation of a continuous process will always be an approximation and not an exact representation. When simulating the trajectory of particles in molecular dynamics, the movement of the beads is determined by calculating the trajectory of each bead in a straight line over the course of one timestep based on the current state of the model. The trajectory of the beads will therefore be a set of straight line movements, each corresponding to one of the beads in the system. As is the case with approximating any continuous function through discrete means,
as the number of discrete sampling points increases, so does the accuracy of the approximation. A simulation requiring a high degree of accuracy would therefore necessitate the use of a much smaller timestep, whereas a simulation that required relatively less accuracy could use a much larger timestep.

4.3 Coarse-Grained MD Simulations

When performing a molecular dynamics simulation there are a wide range of levels of precision that can be used depending on the desired degree of accuracy. If a detailed model is desired, it is possible to model a biomolecule at the quantum level, simulating the behavior of each individual electron within the molecular structure. Unfortunately it demands a great deal of computational resources to simulate. These high demands greatly limit the size of systems and the length of time that can be simulated and therefore limit the size of a quantum-mechanical simulation to only about 100 atoms and the time scale to the order of pico- or femtoseconds[26, 43].

An obvious solution to the computational problems introduced by quantum-mechanical molecular dynamics simulations would be to simulate a system at the atomistic level, representing each atom with a bead. This greatly increases the size of molecules that can be simulated compared to quantum level simulations. Atomistic resolution folding simulations are, however, restricted to very small (about 50 amino acids long), fast-folding (less than a microsecond) proteins at atomistic detail (see [56], for example), even though biologically relevant biomolecules are much larger and can fold much more slowly. By comparison, a single protein chain is, on average, 400 amino acids long and takes considerably longer than a millisecond to fold. To increase the timescales, some researchers use coarse-grained simulations that still accurately capture folding and binding mechanisms of biologically relevant protein and RNA biomolecules[12, 46]. For these classes of simulations, groups of atoms are typi-
cally represented as a bead or group of beads such that the degrees of freedom that are considered to be negligible in the overall folding mechanism are excluded, thereby reducing the total number of simulated particles so that the simulations become more feasible to compute.

4.4 Coarse-grained Simulation Models

One effective coarse-grained simulation model is the Self Organized Polymer (SOP) model, wherein each residue or nucleotide is represented by a single bead that is centered on the amino acid or nucleotide (the C\textsubscript{α} or C2' position) for proteins or RNA, respectively, thereby reducing the total number of simulated particles\cite{26, 43}. Recent studies have demonstrated that MD simulation of biomolecules using the SOP model reproduces experimentally measured mechanical properties of biomolecules with remarkable accuracy\cite{25, 61, 62}.

In the SOP model, the energy that describes the biomolecule, and hence dictates how it moves in time, is as follows:

\[
V(\vec{r}) = V_{FENE} + V_{SSA} + V_{ATT} + V_{VDW} + V_{REP}
\]

\[
= - \sum_{i=1}^{N-1} \frac{k}{2} R_0^2 \log \left[ 1 - \left( \frac{r_{i,i+1} - r_0^i}{R_0^i} \right)^2 \right] + \sum_{i=1}^{N-2} \epsilon_l \left( \frac{\sigma}{r_{i,i+2}} \right)^6
\]

\[
+ \sum_{i=1}^{N-3} \sum_{j=i+3}^{N} \epsilon_h \left[ \left( \frac{r_{i,j}^0}{r_{ij}} \right)^{12} - 2 \left( \frac{r_{i,j}^0}{r_{ij}} \right)^6 \right] \Delta_{i,j}
\]

\[
+ \sum_{i=1}^{N-3} \sum_{j=i+3}^{N} \epsilon_l \left( \frac{\sigma}{r_{i,j}} \right)^6 (1 - \Delta_{i,j})
\]
The first term is the finite extensible nonlinear elastic (FENE) potential that connects each bead to its successive bead in a linear chain. The parameters are: \(k = 20 \text{kal} / (\text{mol} \cdot \AA^2)\), \(R_0 = 0.2 \text{nm}\), \(r_{i,i+1}^0\) is the distance between neighboring beads in the folded structure, and \(r_{i,i+1}\) is the actual distance between neighboring beads at a given time \(t\).

The second term, a soft-sphere angle potential, is applied to all pairs of beads \(i\) and \(i + 2\) to ensure that the chains do not cross.

The third term is the Lennard-Jones potential that describes van der Waals native interactions, which is used to stabilize the folded structure. For each bead pair \(i\) and \(j\), such that \(|i - j| > 2\), a native pair is defined as having a distance less than 8\(\AA\) in the folded structure. If beads \(i\) and \(j\) are a native pair, \(\Delta_{i,j} = 1\), otherwise \(\Delta_{i,j} = 0\). The \(r_{i,j}^0\) term is the actual distance between native pairs at a given time \(t\).

Finally, the fourth term is a Lennard-Jones type repulsive term for van der Walls interactions between all pairs of beads that are non-native, and \(\sigma\), the radii of the interacting particles, is chosen to be 3.8\(\AA\), 5.4\(\AA\), or 7.0\(\AA\) depending on whether the pair involves protein-protein, protein-RNA, or RNA-RNA interactions, respectively.

It is important to note that the interactions in the van der Waals energies (and thus forces) scales as \(O(N^2)\), which can be avoided using a truncation scheme such as a neighbor list algorithm, which we describe below.

### 4.5 Verlet Neighbor List Algorithm

The evaluation of some forces can be simplified by noting that as two particles or beads are far away from each other, the interaction between them is negligible and effectively zero. Therefore, a cutoff radius can be introduced into the computation of the interactions that determines whether it is computed or not.

In the Verlet Neighbor List Algorithm, instead of calculating the force between
every possible pair of van der Waals interactions, a subset neighbor list is constructed
with particles within a “skin” layer radius of \( r_l \). This list is updated every \( n \) timesteps,
and only those interactions between beads less than the cutoff radius, \( r_c \), are computed. These computed interactions are members of a further subset pair list. The
values of \( r_c \) and \( r_l \) are chosen as \( 2.5\sigma \) and \( 3.2\sigma \), respectively. With the neighbor
list algorithm, the computations of the interactions become \( O(Nr_c^3) \approx O(N) \), which
becomes far more computationally tractable\[55\].

4.6 Molecular Dynamics and GPUs

Recently, several studies developed and demonstrated that GPU-optimized MD sim-
ulations, including the empirical force field MD simulation software NAMD\[51, 52\]
and the general purpose particle dynamics simulation software suites HOOMD\[44\]
and LAMMPS\[4\], can significantly increase performance. The nature of molecular
dynamics simulations is such that many large portions of a simulation are highly in-
dependent and can therefore be coded to fully utilize GPUs. At each timestep in a
simulation, a variety of forces must be calculated for each bead. Since the forces on
each bead can be calculated independently of each other at every timestep, individ-
ual threads can be assigned to each force that acts on each bead. For example, if
five forces are being simulated on one hundred beads, a total of five hundred threads
would be used. Each one of the threads can then perform the necessary calculations
in parallel, greatly reducing the computation time of a simulation. Updating the
positions and velocities are likewise highly parallel operations.

While calculating forces acting on each bead and updating their positions and
velocities are by themselves parallel tasks, the entire operation is an ordered, serial
process. At the beginning of each timestep the pair list is calculated, along with the
neighbor list if sufficient time has passed since the last update. Positions are then
updated based on the current distribution of forces. Once this has taken place, the forces acting on the beads based on their new positions must be calculated. Finally, the velocities of each bead are updated based on the forces present in the current timestep. Between each of these steps the entire process must be synchronized in order to perform accurate computations. A molecular dynamics simulation can therefore be thought of as a set of highly parallel tasks that must be performed in a specific order.
Chapter 5: Optimization Strategies for Molecular Dynamics Simulations on GPUs

When optimizing a CPU-based program or algorithm for use on a GPU, there are many different factors that must be taken into account. Though the type of calculations that CPUs and GPUs perform are fundamentally the same, the ways that they go about performing these calculations can vary due to differences in hardware implementations. The parallel hardware of GPUs and their different memory hierarchies and access patterns place constraints on the programmer that are often not necessary to address when writing a CPU-based program.

5.1 Memory Bottleneck Reductions

One of the biggest performance bottlenecks of GPUs is the latency that is introduced when data is transferred between the CPU and the GPU. When writing CPU-based code, the latency of transfers between main host memory and the CPU is rarely an issue. The bus connections between main memory and the CPU are designed to have the highest amount of bandwidth possible and modern compilers and CPUs use many techniques to hide this latency by performing other calculations while waiting for data to be transferred[38].

However, since GPUs are not directly wired to main memory, accessing data residing there on demand is not a viable option due to the latency issues that would be introduced. The current generation of GPUs use the PCI Express 2.0 interconnect, which has a maximum theoretical one-directional throughput of 8.0 GB/s[48]. By comparison, the Quick Path Interconnect used by the x58 Intel chipset allows the CPU to access RAM directly with a total throughput of 25.6 GB/s[28]. The difference
in bandwidth is further exacerbated by the latency that is introduced by transferring data over the PCI Express interface. When a CPU accesses data stored in RAM, there is very little delay that is involved due to the way that it is directly wired to main memory. However, when a GPU accesses main memory through the PCI Express interface, the data must be transferred through the CPU and the PCI Express bus, introducing delay that is very problematic for processing the necessary data in a timely manner. For this reason it is common practice to transfer a large block of data from main memory to the GPU’s onboard memory before performing a set of calculations on it to avoid accessing main memory through relatively slow interconnects. Though this method is many times faster than directly accessing main memory on demand, it is still slow enough to be a bottleneck in many applications.

When developing MD programs for CUDA, one of the most important optimizations is to arrange the code in such a way that memory transfers are kept to a bare minimum. Instead of designing the program to transfer data to the GPU before a calculation takes place and transfer it back to the CPU once the calculation has completed, the program can be written to transfer all of the data that would be needed throughout the course of a simulation to the GPU at the start of the simulation and only transfer data back to the CPU when output needs to be generated.

Generally, for many GPU-based calculations, this type of optimization is not necessary due to the fact that they perform one large, time-intensive calculation and only require a single data transfer from the CPU to the GPU before the calculation takes place and a single transfer from the GPU to the CPU after the calculation completes. When running MD simulations, however, identical calculations must be performed at each simulated timestep and transferring data between the GPU and CPU before and after each calculation would only serve to waste time and would not bring any added benefits to the simulation.
Also, although many GPU-based computations take advantage of precaching data to block-level shared memory that can be accessed much faster than the global GPU memory, the type of independent calculations that are performed in an MD simulation make this kind of optimization unnecessary. In calculations where different threads access overlapping locations in memory, such as calculating dot products in matrix multiplication, the data that is accessed by multiple threads can be cached to faster shared memory so only a single memory transfer is needed instead of one transfer for every thread that accesses the data. However, each piece of data that is needed to calculate the properties of a bead in a molecular dynamics simulation is generally associated with a single bead and is not needed to perform calculations about other beads in the simulation. Since these values are not accessed by multiple threads, caching them before they are needed would provide no performance benefit and could actually decrease performance due to the overhead involved with storing and accessing data in local memory.

5.2 Parallel Sort and Scan Neighbor List Algorithm

Though many portions of molecular dynamics simulation algorithms lend themselves to parallelization, the algorithm used to calculate the neighbor and pair lists cannot be modified to run in parallel, at least in the form originally introduced by Verlet[55]. In that algorithm, during the course of a normal simulation, the neighbor list is updated once every ten thousand timesteps, which is a relatively low frequency, but the pair list must be updated at every timestep.

Calling serial code once every timestep to update the pair list causes a severe performance issue, greatly affecting the execution time of a simulation. The performance decreases further because a large amount of data must be copied from the GPU to
the CPU before the neighbor list or pair list is calculated, and the data representing the neighbor list or pair list must be copied from the CPU to the GPU once the serial calculation has taken place. The combined performance drawbacks of performing a serial calculation on the CPU and the expensive memory transfers create a severe performance bottleneck. Therefore, performing an equivalent parallel calculation on the GPU would eliminate the bottleneck.

5.2.1 CPU-Based Neighbor List Algorithm

The original Verlet neighbor list algorithm that is optimized for CPUs works by serially iterating through an array of all the interactions between pairs of beads in the simulation unit called the Master List. When an iteration at position $i$ is found to contain beads that are within a certain cutoff distance of each other, they are copied to the Neighbor List. The index in the Neighbor List array to which the Master List entry will be copied is determined by how many previous entries have been added to the Neighbor List. For example, if $j$ entries have already been added to the Neighbor List when the $i$th entry of the Member List is to be moved to the Neighbor List, the $i^{th}$ entry of the Master List is copied to the $j + 1^{st}$ entry of the Neighbor List.

```c
void copyDataToCPU() // Slow process
j = 0; // Number of entries in the Neighbor List
for (i = 0; i < MasterListSize; i++)
    distance = getDistanceBetweenBeads(MasterList[i]);
    if (distance < CutoffDistance)
        NeighborList[j] = MasterList[i];
    j++;
void copyDataToGPU(); // Slow process
```

Figure 5.1: Verlet neighbor list algorithm

If the action taken by each iteration of the for loop was independent of the others, the algorithm would be readily parallelizable, as each action could be assigned to
its own independent, parallel process. However, since the value of the $j$ variable could vary at each iteration, there is no way to know \textit{a priori} which location of the Neighbor List the values of the Master List could be copied to without calculating each of the previous values. Therefore, it is obvious that this version of the neighbor list algorithm is inherently serial since the location of the copy that takes place in each iteration of the for loop is dependent on the results of each of the previous iterations.

5.2.2 Naïve Parallel Neighbor List Algorithm

There is, however, a modification that can be made to partially parallelize a portion of this algorithm. At each iteration, the distance between the two beads represented by the $i^{th}$ entry in the Master List must be computed, and each of these $i$ calculations can be done in parallel. Although this will not entirely solve the problem of parallelizing the neighbor list, computing the distance between two beads is a computationally involved process, so a parallel computation will increase the efficiency of the algorithm. This approach essentially divides the algorithm into a parallel section and a serial section that will execute on the GPU and the CPU, respectively.

In order to merge these two sections together, a new array known as the Member List is introduced which is of equal length to the Master List. When the parallel section of the algorithm determines that the pair of beads represented by the $i^{th}$ entry of the Master List are within the cutoff distance of each other, the corresponding entry of the Member List is set to “true.” Alternatively, if the pair of beads are not within the cutoff distance, the corresponding entry in the Member List is set to “false.”

The second, serial portion of the algorithm then iterates through the Member list and when an entry at position $i$ is found to be “true,” the $i^{th}$ entry of the Master List is copied to the Neighbor List.
ParallelGetDistanceBetweenBeads(Master_List, Member_List);
copySomeDataToCPU(); // Slow processes
j = 0; // Number of entries in the Neighbor List
for (i = 0; i < MasterListSize; i++)
  if (Member_List[i] == TRUE)
    Neighbor_List[j] = Master_List[i];
  j++;
copySomeDataToGPU(); // Slow process

Figure 5.2: Naïve GPU neighbor list algorithm

This naïve GPU-based neighbor list algorithm provides a significant performance boost over the original CPU-based algorithm, but is not ideal. There are still two memory transfers that must take place as well as a serial, CPU-based calculation. Eliminating these two bottlenecks would allow the performance of the MD simulation to increase significantly.

5.2.3 Fully Parallel Neighbor List Algorithm

To fully parallelize the algorithm to run on a GPU, we developed a novel algorithm that utilizes the key-value sort and parallel scan functionalities of the CUDPP Library. This approach calculates a Neighbor List that is equivalent to the one calculated by the serial neighbor list algorithm and accomplishes the task entirely in parallel on the GPU, circumventing the need to transfer data between the CPU and GPU and perform serial calculations on the CPU.

The first step of this parallel method is to perform a key-value sort on the data. This type of sort involves two arrays of equal length known as the “key” array and the “value” array which are sorted based on the numerical values of the entries of the key array so that the value array ends up being sorted in an identical way to the key array. A key-value sort is very useful when one set of data needs to be sorted based on properties that are not recorded in the data itself. For this algorithm, the Member
List is used as the key array and the Master List is used as the value array.

Creating the Neighbor List from the Master List can be viewed as this type of operation. When the Master List is treated as the set of values in a key-value sort, assigning numerical values to the corresponding entries of the Member List allows it to be used as the set of keys. Since the Member List needs to only hold a value of “true” or “false,” these values can be represented with a one or a zero\(^1\). These values can be sorted in a way such that the “true” values are moved to the top of the Member List and the “false” values are moved to the bottom of the list. The associated values of the Master List will be moved in an identical fashion, meaning that the entries of the Master List that are part of the Neighbor List will occupy the first entries in the Master List, as shown in Figure 5.3a.

Though the entries that need to be copied to the Neighbor List are guaranteed to be at the top of the Master List after the key-value sort takes place, the exact number of entries that need to be copied to the Neighbor List will still need to be determined. Fortunately, the parallel scan algorithm implemented by CUDPP allows all of the values stored in an array to be quickly summed up in parallel. Since the values present in the Member List consist of only ones and zeros, a parallel scan will produce the total number of “true” values present in the array, denoted by \(\text{num}_{NL}\), indicating how many entries of the Master List are within the neighbor list cutoff distance, as shown in Figure 5.3b. Finally, once the arrays are sorted and scanned, the first \(\text{num}_{NL}\) entries of the Master List will be copied to the Neighbor List, as shown in Figure 5.3c.

Therefore, using the parallel key-value sort and scan functionality of CUDPP produces a Neighbor List that that is equivalent to the one created by the serial

\(^1\)The CUDPP 1.1.1 library only supports ascending order sorts, so “true” values must be represented as zeros to have them sorted to the top of the Master List. However, the CUDPP 2.0 library supports ascending and descending sorts and can use either zeros or ones. Though the final version of our code uses CUDPP 1.1.1, we will represent “true” values as ones and “false” values as zeros to avoid confusion.
CPU algorithm without requiring the GPU to pause its calculations and sit idle while the data is transferred to and from the CPU and serial calculations are performed on the CPU. This removes a severe performance bottleneck and allows the program to perform much more efficiently and allows for the entirety of the computationally expensive portions of code to run on the GPU, greatly increasing the efficiency of the program and allowing simulations to be completed much more quickly.

An identical approach is taken to generate the Pair List from the entries in the Neighbor List. Instead of using the entries of the Member List to generate the Neighbor List, the entries of the Neighbor List are used to generate the Pair List. Both the Neighbor and Pair Lists are created based on the distance between two beads in an interaction; the only difference is the list that is used as the source and the cutoff distance between the two beads.
5.3 Array Reuse

The original implementation of the neighbor list algorithm called for three arrays to store data relating to the various beads and interactions that were part of the Master List, Neighbor List, and Pair List. The Master List array was treated as an array of all of the interactions in the biomolecule and was never altered. When the Neighbor List was generated, the interactions within the Neighbor List cutoff distance were copied from the Master List array to the Neighbor List array. Similarly, the interactions present in the Pair List were copied from the Neighbor List array to the Pair List array. The maximum size of the Neighbor and Pair List arrays cannot be known before a simulation is performed, so these arrays were always the same size as the Master List, denoted by $num_{ML}$.

While representing the Master, Neighbor and Pair Lists as three separate arrays is suitable, another approach can be taken which will allow the same information to be contained in a single array instead of three separate arrays. Creating the Neighbor List is equivalent to creating a subset of interactions contained in the Master List. Creating the Pair List is likewise equivalent to creating a subset of interactions that are present in the Neighbor List. It is therefore possible to view the Neighbor and Pair Lists as subsets instead of separate lists such that $PL \subset NL \subset PL$ holds, where $PL$, $NL$, and $ML$ represent the Pair List, Neighbor List and Master List, respectively. When creating the Neighbor List, the sort and scan algorithm discussed above does an in-place sort to arrange the interactions in the Master List that are present in the Neighbor List such that they occupy the first $num_{NL}$ entries, where $num_{NL}$ is the number of interactions in the Neighbor List. Transferring these entries to a Neighbor List array serves to transfer the $i^{th}$ entry of the Master List to the $i^{th}$ entry of the Neighbor List for the first $num_{NL}$ entries, so that the entries in the Neighbor List array are identical to the first $num_{NL}$ entries of the Master List. Therefore, instead
of transferring these values to the Neighbor List, they can be kept in their sorted location in the Master List, as shown in Figure 5.4. When an interaction needs to be read, the program can therefore read from an index in the Master List instead of the Neighbor List, obviating the need for a second array to store the Neighbor List values. During the course of a simulation, the Master List will only need to be accessed to generate the Neighbor List, so the order of the values stored in the list does not matter.

Similarly, the Pair List can be treated as a subset of the Neighbor List interactions. In order to calculate which interactions are present in the Pair List, the sort and scan operations will be performed on the combination Master/Neighbor List array, moving the Pair List interactions to the top of the array. The first \( \text{num}_\text{PL} \) entries of the sorted array will therefore be the interactions that are part of the Pair List, where \( \text{num}_\text{PL} \) is the number of Pair List interactions, as shown in Figure 5.5a. Only

![Figure 5.4: Reusing the Master List to store the Neighbor List](image)

![Figure 5.5: Reusing the Master List to store the Neighbor and Pair Lists.](image)
the first $num_{NL}$ entries will be sorted, meaning that the entries that are present in the Neighbor List will remain in the first $num_{NL}$ entries in the array. As a result of these operations, the first $num_{PL}$ entries of the combined Master/Neighbor/Pair List array will be the Pair List interactions and the first $num_{NL}$ entries will be the Neighbor List interactions, as Figure 5.5b shows. Because the interactions of the Pair List are merely the interactions in the Neighbor List that are within a radius $r_c$ of each other, we are guaranteed that $num_{PL} \leq num_{NL} \leq num_{ML}$.

This algorithm has a twofold benefit to MD simulations. First, it serves to reduce the amount of memory transfers that take place by removing the need to copy values from the Master List to the Neighbor List and from the Neighbor List to the Pair list, thereby increasing performance. Second, it reduces the overall memory footprint of the application by combining three of the largest arrays in the simulation into a single array. If a simulation is to be run on a GPU with a limited amount of memory, reusing arrays can make the difference between being able to fit the simulation for the data into the GPU’s memory and preventing the simulation from running on the GPU.

### 5.4 Current CUDPP Library Limitations

Unfortunately, array reuse was not fully implemented due to some limitations in the CUDPP library. Versions 1.1.1 and 2.0 of the CUDPP library were both used in the course of this study. Attempting to perform a key value sort on an array with more than approximately 32 million values would cause the CUDPP 1.1.1 library to produce an error, so array reuse could not be used when simulating biomolecules above a certain size. The 70s ribosome has over 52 million Master List interactions, which required multiple sorting passes to be done on the Master List before a Neighbor List could be created when using CUDPP 1.1.1. Although the CUDPP version 2.0
library has removed this limitation, the release notes warn that the performance of the key value sort algorithm may decrease with smaller sized arrays, but the issue will be addressed in a future release[14]. An algorithm was developed to address the limitations of the CUDPP 1.1.1 library’s sorting functionality and the performance results were compared with those of the CUDPP 2.0 library’s fully functional sorting technique with array reuse. Though the simulation model that utilized CUDPP 1.1.1 used array reuse with only the Neighbor and Pair Lists instead of all three lists and required additional overhead with the workaround algorithm, it still performed better than the CUDPP 2.0-based simulation model with array reuse, as we will discuss in Chapter 6.

When a Master List with more than 32 million elements needs to be sorted using the CUDPP 1.1.1 library, the array must be split into separate sub-arrays whose sizes are less than or equal to 32 million elements. For arrays with more than 32 million but less than 64 million entries, the array can be split into two parts, sorted individually, and then combined in a way so that the entries that should be added to the Neighbor List occupy the topmost entries in the array. For arrays with between 64 million and 96 million values, the arrays must be split into three pieces, and so on. The 70s ribosome has roughly 52 million entries in its largest array, so only one split is used in the MD simulation implemented for this thesis. The algorithm for sorting the Master List in this manner in order to create the Neighbor List is outlined below.

The first step of the algorithm is to copy the first 32 million values from the Master List to the Neighbor List (Figure 5.6a). Next, the remaining Master List entries are copied to a temporary array called Neighbor List Temp whose size is denoted by $num_{Temp}$ (Figure 5.6b). The last $num_{Temp}$ entries of the Member List are likewise copied to a temporary array Member List Temp (Figure 5.6c). The first 32 million entries in the Neighbor List and Member List are then sorted and
scanned (Figure 5.6d) in the manner outlined above. The sum produced by the scan operation, $num_{NL1}$, indicates how many of these entries are within the neighbor list cutoff distance. These entries will occupy the first $num_{NL1}$ entries of the Neighbor List array.

Next, the Neighbor List Temp and Member List Temp arrays are sorted and scanned (Figure 5.6e) in the same way. The sum produced by the scan operation performed on Neighbor List Temp, $num_{NL2}$, is the number of entries in the Neighbor List Temp array that are within the neighbor list cutoff distance. The first $num_{NL2}$ entries of Neighbor List Temp are then copied to the Neighbor List array (Figure 5.6f), starting at the $num_{NL1} + 1^{st}$ position of the Neighbor List. The total number of neighbor list interactions is therefore $num_{NL} = num_{NL1} + num_{NL2}$ and the first $num_{NL}$ elements of the Neighbor List array are the interactions within the neighbor list cutoff distance.

Originally an algorithm was developed to perform a similar set of operations while reusing the Master List array for both the Neighbor and Pair Lists, but the results produced by it were inaccurate.

For this reason the above algorithm was developed and the Master List is not able to be reused for storing the Neighbor and Pair Lists as this would result in some interactions being overwritten and thereby lost. However, when calculating the Pair List, the Neighbor List array is reused by sorting the Pair List values to the top of the Neighbor List array.
(a) The first 32 million values from the Master List are copied to the Neighbor List
(b) The remaining Master List entries are copied to the Neighbor List Temp array
(c) The remaining Member List entries are copied to the Member List Temp array
(d) The first 32 million entries in the Neighbor List and Member List are sorted and scanned
(e) The Neighbor List Temp and Member List Temp arrays are sorted and scanned
(f) The first $num_{NL2}$ entries of Neighbor List Temp are copied to the Neighbor List array

Figure 5.6: CUDPP 1.1.1 Neighbor List sort and scan algorithm
5.5 Type Compression

Loading all of the necessary data into device memory at the start of a simulation reduces one memory bottleneck, but in the course of each calculation information must be transferred between the global device memory and either the local memory or registers of an individual compute core. Though these transfers are much faster than those between the CPU and the GPU, there are a large number of these transfers that must take place for every calculation and the total time required to transfer the data needed by the compute cores from global device memory can quickly add up.

The most straightforward way to cut down on the amount of data that is transferred in these situations, thereby minimizing latency, is to use the smallest data types possible. When using integral data types, the minimum size of each variable is dictated by the maximum value that it could be required to hold in a simulation. For example, if 8 bits were allocated to assign a unique number to each bead in a simulation, there would be a maximum of $2^8 = 256$ beads that could be used in a simulation. Using smaller data types therefore limits the size of the structure that can be simulated, but in practice this does not create any problems so long as reasonable sizes are chosen.

Data types in C/C++, CUDA, and most other programming languages come in only a limited number of sizes, however, so it is entirely possible that when trying to find an optimal size for a data type given a maximum structure size, the maximum values given by the built-in data types may be either too small to store the information necessary, or much larger than is necessary, leading to wasted space in the form of bits that will never be used.

One variable that can have its size reduced in the SOP model is the variable identifying the index numbers of the beads that are represented in an interaction stored in the Neighbor List and Pair List. The largest biomolecule studied for this thesis is the
70s ribosome, which has a total of 10,219 beads, so the variable used to represent the index of one of these beads must be able to represent at least 10,219 different values. The formula that gives the minimum number of bits required to represent a given number \( x \) is \( \lceil \log_2(x) \rceil \), so the minimum number of bits needed to assign a number to each of the beads in the 70s ribosome is therefore \( \lceil \log_2(10,219) \rceil = 14 \). Using 14 bits allows a maximum of \( 2^{14} = 16,384 \) beads to be represented in this way.

Unfortunately, there are no 14-bit data types in CUDA or C/C++. The ushort (unsigned short integer) data type occupies 16 bits of memory, which is enough to represent \( 2^{16} = 65,536 \) different values. This is much more than would be necessary for the 70s ribosome, but leaves 2 bits that will always be unused. Using the ushort data type instead of the 32-bit int data type still reduces the amount of time taken while transferring data and the amount of space needed to store that data by half.

In addition to the variable identifying each bead in a Neighbor List or Pair List entry, the SOP model uses a variable to represent the type of interaction the entry is. There are a total of three different types of interactions, meaning that no more than 2 bits would be necessary to represent these values. The smallest data type in CUDA and C/C++ is the uchar (unsigned character) which is an 8-bit data type capable of representing \( 2^8 = 256 \) different values. Though this is more than enough unique values, a total of 6 bits will always be unused, leading to a large waste of storage space and memory transfer time, but this is still much better than using a 32-bit int data type.

Since the identifying number for a bead in the 70s ribosome could be represented by using 14 bits and the type of interaction could be represented by using only 2 bits, these two variables could be combined into a single 16-bit ushort value to completely eliminate the wasted space when using a ushort and a uchar to represent each of them individually. This reduces the total memory requirement for each entry from 24 bits
(a) A total of 128 bits of memory are required when using 4 32-bit ints to represent two interaction indices and two interaction types. Only 14 bits of the two index values will ever be used with the 70s ribosome and only 2 bits of the type values will be used. A total of 96 will therefore always be unused.

(b) A total of 48 bits are required when using two 16-bit ushorts and two 8-bit uchars to represent two interaction indices and two interaction types. A total of 16 bits will always be unused.

(c) Only 32 bits are required when using a single 32-bit ushort2 to represent two interaction indices and two interaction types. There will be no unused bits when simulating the 70s ribosome.

Figure 5.7: Memory footprints when using different data types to represent two interaction indices and two types. The bits that will be used in the course of a simulation of a 70s unit are highlighted in red.
to 16 bits, leading to a 33% reduction in memory requirements. Additionally, the number of read and write operations is cut in half because only one data type will be transferred instead of two.

In each interaction there are a total of two different beads that must be stored, so two ushort values will need to be stored. CUDA provides an implementation of a ushort2 data type, which is essentially two 16-bit ushort values combined into a 32-bit data type. Using a ushort2 instead of two ushort values takes up the same amount of memory, but can reduce the number of memory reads required to transfer data from main memory to thread-level memory, depending on access patterns. The ushort2 data type can be thought of a two dimensional vector with an $x$ and $y$ component. These components can be accessed in order to extract either of the two ushort values stored in the ushort2.

The interaction index and interaction type are combined into a single $x$ or $y$ component of a ushort2 by using the 2 highest-order bits of one of the 16-bit ushort2 components to store the type of the interaction and the 14 low-order bits to store the identifying number of the bead. These values are combined by setting the component’s value to be equal to the value of the interaction type left shifted by 14 bits and using a bitwise OR operation to set the low-order fourteen bits to the identifying bead number. Once these values are stored in a ushort2 component, they can be retrieved very easily. To find the type of the interaction that is stored in the component, the value is simply right shifted by 14 bits. Alternatively, to find the identifying number stored in the variable, a bitwise AND operation is performed to set the 2 highest-order bits to zero, leaving the 14 lowest-order bits unchanged.

---

2This approach allots 14 bits to represent the bead index, for a total of $2^{14} = 16,384$ possible beads. This places a limitation on the size of biomolecules that can be simulated, but still allows for a biomolecule with roughly 60% more beads than the 70s ribosome to be simulated.
Listing 5.1: Type Compression and Decompression C Code

// The 2-bit type value and the 14-bit index value are combined into a single 16-bit value by left shifting the type value and then ORing the results with the 14-bit index value.
#define COMBINE(type, idx) ((type << 14) | idx)

// The type of interaction stored in a type compressed value can be retrieved by right shifting the value by 14 bits
#define GET_TYPE(combined) ((combined) >> 14)

// The index of an interaction stored in a type compressed value can be retrieved by ANDing the value with 0x3FFF (equivalent to the binary value 0011 1111 1111 1111), which sets the two high-order bits to 0 while leaving the 14 low-order bits unchanged.
#define GET_IDX(combined) ((combined) & 0x3FFF)

5.6 cuRAND

When writing molecular dynamics simulation software, it is often necessary to include a set of random forces that will simulate the interaction of the biomolecule and its surrounding environment. If this is not done, the simulation will model a biomolecule in a complete vacuum. Since this will never be the case in any natural environment, it is common to program an environment wherein the biomolecule interacts with some kind of medium, most often water. Simulating a large number of beads of water molecules would put a large strain on the simulation and would involve many more beads to represent the water molecules than it would to simulate the actual biomolecule. Since these interactions are essentially random, however, it is possible to apply a random force to each bead of the biomolecule at every timestep to simulate its interaction with the surrounding environment. This approach does away with the necessity of simulating a large number of extraneous water beads while still
maintaining the accuracy of the simulation.

In practice, applying this simulated random force involves generating a random, normally distributed three dimensional vector by generating three random numbers, one each for the $x$, $y$, and $z$ direction of the force, for each bead in a simulation and then applying that force to the bead in addition to the other forces that are calculated at every timestep. A total of $3N$ random numbers will be needed every timestep, where $N$ is the number of beads in the simulation.

In computer programming, generating random numbers can be a very difficult problem. Computers are by their very nature deterministic machines, making the generation of truly random numbers without some kind of truly random external input impossible. There are, however, many methods for generating streams of numbers that possess many of the statistical properties of a truly random stream of numbers. Many mathematical functions have been discovered that can generate these kinds of series of numbers and have been incorporated into many different software RNGs that allow programmers to introduce apparently random numbers into computer programs. Though RNGs are still deterministic, they are made much more unpredictable by using “seed” values to set up the initial state of the RNG. By using different seeds, the same RNG can produce different series of random numbers, reducing the amount of predictability of the process. Most common languages such as C/C++ and Java have RNGs implemented in their standard library of functions, allowing programmers to easily access RNGs. These libraries are often relied on instead of writing custom RNGs due to the mathematical complexity of designing a robust and reliable RNG.

NVIDIA’s cuRAND library[35] allows RNGs to be created for use on a GPU that can generate streams of random numbers in parallel. To use the cuRAND library in GPU code, the programmer must first create one or more “generator” objects which will handle the parallel generation of random numbers. Each generator has its own
independent state, involving a seed and sequence offset, that allows it to generate a sequence of random numbers on demand. If each generator is to produce a different sequence of random numbers, each must be supplied with a unique seed number. When a kernel requires a random number, the `curandGenerate()` function is called on a generator, which will cause the generator to return a random number and update its internal sequence counter. If a total of \( N \) parallel threads will need access to RNGs, there must be \( N \) generators available in the GPU’s memory. Having fewer than \( N \) independent RNGs will require that some threads share RNGs, leading to a reduction in performance. Fortunately, it was determined that each RNG requires only 40 bytes of memory, meaning that it is possible to have a large number of cuRAND RNGs in memory at any given time.

Without cuRAND or a similar library, using random numbers in a general purpose GPU calculation would require either writing a random number generator from scratch—a process that requires a great deal of mathematical knowledge and effort—or relying on a CPU-based random number generator to output random numbers that would be transferred to the GPU. This would involve either transferring all of the needed numbers to the GPU before a simulation was run, generating the numbers as needed and then transferring them to the GPU, or some variation of one of these approaches. For a 70s simulation involving 10,219 beads, three 32-bit single precision floating point random numbers per bead per timestep would require a total of \( 10,219 \times 3 \times 32 = 120\text{kB} \) of information to be transferred. Every 1GB worth of random numbers would therefore be sufficient for only 8,333 timesteps—a very short timeframe when running a long simulation consisting of millions or even billions of timesteps.

A variation on this approach that is much more efficient would be to generate the random forces on the CPU and then transfer them to the GPU every timestep. If
the random forces are calculated before any other forces, an array representing the forces can be populated with the random forces and then transferred to the GPU each timestep, doing away with the requirement to store a large number of random numbers on the GPU. Once each random number is generated, it can simply be stored in the force array without requiring any arithmetic operations to be done. If the random force was calculated after the other forces, however, the array representing the total of all of the previous forces would need to be copied to the CPU, where the random forces would be added to the previously calculated values, and then these newly calculated forces would need to be transferred back to the GPU. This involves an extra transfer to the CPU and many more arithmetic operations than would be required if the random forces were generated before any other forces.

Though generating the random forces before any others on the CPU is more efficient than other methods, it still requires the GPU to idle during the computations and involves a transfer of data between the CPU and GPU. However, utilizing the cuRAND library allows these performance bottlenecks to be overcome by performing all of the random number generation in parallel on the GPU, eliminating both the serial portions of random number generation and the memory transfers.

5.7 Force Evaluation Optimizations

Evaluating the forces in a molecular dynamics simulation involves iterating through each bead and calculating the different forces that are acting on the bead. To optimize the calculation of a specific type of force for use on a GPU, it is possible to launch a thread for each bead that the force is acting upon and execute those threads in parallel.

A separate kernel for each force will be launched and, in the case of the random forces, FENE forces and soft sphere angular forces, the kernels will contain a number
of threads equal to the total number of beads in the biomolecule being simulated. The attractive and repulsive van der Waals forces, however, are the result of interactions between pair of beads and will be set up in a different way.

The neighbor and pair list evaluations determine which interactions have non-negligible van der Waals forces, so the number of threads launched by the van der Waals kernels will be equal to the number of attractive and repulsive interactions that are part of the pair list. Each thread within a van der Waals kernel will update the forces acting on two separate beads and a given bead may have multiple threads calculating forces for it. For this reason, the force updates calculated by the van der Waals kernels must be done atomically to ensure that memory write collisions do not interfere with correct calculation of the total forces acting on an individual bead. The atomic actions will execute more slowly than non-atomic operations, but the nature of the van der Waals calculations require that they be used.

The evaluation of the different forces can be further optimized by launching the kernels concurrently. The original CPU code evaluated each force in succession, but CUDA GPUs are capable of launching multiple kernels at once in order to execute them concurrently using what are known as “streams.” Streams allow multiple kernels to be executed at once, which can potentially increase the overall throughput of calculations if a set of kernels that do not fully utilize the GPU are ran simultaneously. Since each force can be evaluated independently of all the others, there is no reason that they cannot be calculated concurrently. Since all of the kernels will be executing simultaneously, it is possible that two kernels will attempt to write to the same place in memory at the same time, which can lead to incorrect results if write collisions occur. Streaming the kernels in the manner will therefore require that the updates to the forces acting on each bead be done atomically, just as the van der Waals forces were updated. The performance reduction caused by the atomic operations was found
to be negligible compared to the performance gains that resulted from streaming the
force evaluation kernels, so streams are used as the preferred force calculation method.

5.8 Floating Point Considerations

On the current generation of NVIDIA GPUs, single precision floating point calculations perform much faster than equivalent double precision calculations. An obvious optimization was therefore to use single precision instead of double precision. This optimization was straightforward and involved simply replacing every occurrence of a double precision value with a single precision value.

Though performance will increase when changing from double to single precision, there was a chance that a significant level of accuracy might be lost in the process. We will compare the accuracy of single and double precision simulations in Section 6.2. It should be noted that when implementing the double precision MD code it was not possible to use double precision values for all of the floating point values in the simulation.

CUDA currently does not allow using double precision values with atomic operations, but the force evaluations required that atomic operations be used. For this reason the force array was implemented using single precision values instead of double precision values. Additionally, the native distance array used for some of the van der Waals forces must be sorted by CUDPP at several points in the algorithm. CUDPP 1.1.1 only supports the use of 32-bit values in its sorts, meaning that 32-bit single precision values must be used. The SOP GPU implementation using all double precision for all values except the force and native distance values will hereafter be referred to as “hybrid double/single” precision. In practice this will not create a problem because the values that are read into the program to populate the native distance array at the program’s startup have only a few significant digits, so there would be no difference
between storing these values as single or double precision.

When performing calculations involving either of these arrays of single precision values, they are cast as double precision numbers at all intermediate points of calculation and are converted back to single precision only when they are to be stored. Performing all of the calculations after casting these values as double precision prevents any precision from being lost during computation and limits it to the casts that must take place when the final result of a calculation is stored in memory. The loss in precision using this method is negligible, as will be shown in Section 6.2.3.
Chapter 6: GPU-Optimized Molecular Dynamics Simulation Performances

The optimizations discussed previously significantly increased the performance of MD simulations and allowed larger biomolecules to be simulated on GPUs with smaller amounts of memory than would have been possible otherwise. With any type of optimization, however, it is important that accuracy is not sacrificed for performance. Molecular dynamics simulations can be sensitive to loss of accuracy due to the precise nature of biological processes. In this chapter, we present the performance increases that were made by optimizing molecular dynamics code for GPUs and demonstrate that these optimizations were not made at the expense of accuracy.

6.1 Performance Improvements

The use of CUDA and NVIDIA’s GPUs in performing molecular dynamics simulations has led to a large degree of improvement over a strictly CPU-based approach. Though the entire process has many portions of code that must be run one after another, this does not prevent the performance of the GPU from being utilized in order to produce very efficient code. Each of the different sections of the molecular dynamics algorithm has different characteristics that lend themselves to varying degrees of parallelization.

6.1.1 Overall Improvements

When all of the improvements described above are applied to the SOP model, the total performance of the application greatly improves when simulating molecules above a certain size. Figure 6.1 shows the execution time for a one million timestep simulation...
of five different biomolecules using CPU and GPU code with both double and single precision.

![Execution Time Graph](image)

<table>
<thead>
<tr>
<th>Number of Beads</th>
<th>tRNA\textsuperscript{phe}</th>
<th>16s</th>
<th>30s</th>
<th>50s</th>
<th>70s</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Single</td>
<td>292.45</td>
<td>389.31</td>
<td>641.89</td>
<td>1,191.87</td>
<td>2,204.87</td>
</tr>
<tr>
<td>GPU Double</td>
<td>292.45</td>
<td>411.94</td>
<td>737.73</td>
<td>1,409.57</td>
<td>2,647.22</td>
</tr>
<tr>
<td>CPU Single</td>
<td>142.90</td>
<td>5,293.94</td>
<td>16,827.40</td>
<td>36,209.40</td>
<td>77,009.10</td>
</tr>
<tr>
<td>CPU Double</td>
<td>137.16</td>
<td>4,959.63</td>
<td>15,619.10</td>
<td>33,975.30</td>
<td>71,249.80</td>
</tr>
</tbody>
</table>

Figure 6.1: Execution times in seconds for different molecules and CPU/GPU floating point precisions

The run times of the simulations decrease dramatically when run on the GPU in the case of all of the biomolecules except for the tRNA\textsuperscript{phe} unit which consists of only 76 total beads. The low number of beads prevents the parallel nature of the GPU from being fully utilized and any performance benefits that the GPU may give are cancelled out by the amount of overhead introduced in launching and managing the CUDA kernels from the CPU.

However, with biomolecules over a certain size, the parallel nature of the GPU
can be taken advantage of more effectively. The overhead of launching and managing kernels becomes negligible compared to the execution time of the kernel, which will be much faster than an equivalent CPU-based calculation. Starting with the 16s subunit, consisting of 1,530 beads, the GPU code runs much faster than the CPU code. This behavior continues through the largest biomolecule studied for this thesis, the 70s ribosome, consisting of 10,219 beads.

To quantify the performance difference between the CPU and GPU code, we observe the speedup of the GPU code, defined as $\frac{\text{Time}_{CPU}}{\text{Time}_{GPU}}$, where $\text{Time}_{CPU}$ and $\text{Time}_{GPU}$ are the execution times for the CPU and GPU, respectively. The execution times of both the GPU and CPU code were observed when using single precision floating point numbers and when using double precision floating point numbers. The CPU had the best performance when using double precision, so all GPU speedups are in comparison to the double precision CPU code.

The speedup of the single and double precision GPU code is shown in Figure 6.2. Simulations of the 70s biomolecule were shown to execute over 32 times faster when using the GPU single precision code and nearly 27 times faster when using the GPU double precision code. It is noteworthy that the speedup is not constant across all sizes and increases with the size of the biomolecule. This is due to the fact that with larger biomolecules, a greater number of calculations need to be performed due to the increased number of beads. When simulating a biomolecule with a small number of beads, all of the parallel cores of the GPU will not be utilized by the kernels that are launched during the simulation. However, when a simulation contains a larger number of beads, more processors can be utilized by the calculations, leading to greater throughput and higher utilization.

Though individual double precision calculations will take two or more times longer than equivalent single precision calculations on a GPU, depending on the exact hard-
Figure 6.2: Speedup of GPU code using single and double precision.

<table>
<thead>
<tr>
<th>Number of Beads</th>
<th>tRNA&lt;sub&gt;phe&lt;/sub&gt;</th>
<th>16s</th>
<th>30s</th>
<th>50s</th>
<th>70s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision</td>
<td>0.48</td>
<td>12.74</td>
<td>24.33</td>
<td>28.51</td>
<td>32.31</td>
</tr>
<tr>
<td>Double Precision</td>
<td>0.47</td>
<td>12.04</td>
<td>21.17</td>
<td>24.1</td>
<td>26.91</td>
</tr>
</tbody>
</table>

Figure 6.3 compares the execution time of the GPU single precision and double precision code as well as the percentage difference in execution times.

The performance difference is shown to be proportional to the size of the unit being simulated. The simulation of the tRNA<sub>phe</sub> unit was slowed down by only 3.26%, whereas this increases to 20.06% with the 70s ribosome. We can see that the performance reduction scales with the size of the biomolecule, but does not cause the simulation to run 2 or more times slower, as might be expected due to the reduced double precision throughput. The performance is not drastically reduced by the use of double precision floating point values because of the variety of operations that are performed during the course of a simulation. In addition to floating point calculations,
integer and memory operations are performed, which will both perform identically regardless of which precision floating point values are used.

<table>
<thead>
<tr>
<th>Number of Beads</th>
<th>tRNA&lt;sub&gt;16s&lt;/sub&gt;</th>
<th>16s</th>
<th>30s</th>
<th>50s</th>
<th>70s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Precision</td>
<td>283.23</td>
<td>389.31</td>
<td>641.89</td>
<td>1,191.87</td>
<td>2,204.87</td>
</tr>
<tr>
<td>Double Precision</td>
<td>292.45</td>
<td>411.94</td>
<td>737.73</td>
<td>1,409.57</td>
<td>2,647.22</td>
</tr>
<tr>
<td>% Difference</td>
<td>3.26%</td>
<td>5.81%</td>
<td>14.93%</td>
<td>18.27%</td>
<td>20.06%</td>
</tr>
</tbody>
</table>

Figure 6.3: Execution times in seconds for different biomolecule simulations using single and double precision on the GPU.
6.1.2 Component Speedup

Though it is important to quantify the overall speedup of a simulation, insights into the performance characteristics of the simulation can be gained by observing the speedups of each section of code in a simulation. Figure 6.4 shows the speedups of each portion of the single precision GPU-optimized SOP program as well as the overall speedup of the entire program for reference. As was the case with the total speedup, the speedup of each section of code tended to increase along with the size of the biomolecule. The only exception to this trend was the slight decrease in speedup of the Neighbor List evaluation for the 70s ribosome, from 56.33 with the 50s unit to 51.62 with the 70s ribosome.

This reduced speedup is caused by the additional overhead introduced by performing the CUDPP 1.1.1 neighbor list algorithm for simulations involving more than 32 million interactions. Though the algorithm requires more overhead for larger units, the performance is not degraded to a point that would make it unusable. Even with the additional overhead of the CUDPP 1.1.1 neighbor list algorithm, the neighbor list is still calculated over 50 times faster than it would be on the CPU.

The speedups for the other sections of code tend to vary significantly, showing that the types of calculations that are performed at each step in the molecular dynamics algorithm are not uniform in complexity. When evaluating forces, the GPU is able to perform nearly 17 times faster with the 70s ribosome and the speedups of the other portions of code increase to up to a 143 times speedup when performing the position updates.

The only slowdown that was observed aside from those involving the tRNA$^\text{phe}$ was with the logging portion of the program. This portion of code involves writing data about the current state of the simulation to disk, which is an operation that must be done serially by the CPU. Though the portions of code that output data
are unchanged between the CPU and GPU implementations, the GPU code must first transfer up-to-date data from the GPU to the CPU for logging. Though this introduces some additional overhead into the calculation, in the worst case performance drops by only 5%, showing that the transfer times are relatively insignificant compared to the time it takes to output the data to disk.
Figure 6.4: Speedup of individual GPU simulation components
6.1.3 Force Evaluation Speedup

The force evaluation component of the molecular dynamics algorithm involves calculating four different forces: random forces, FENE forces, soft sphere angular (SSA) forces and van der Waals (VDW) forces. Although they will be calculated one after another on the CPU, the GPU is able to calculate all of them in parallel by launching kernels for each force in a separate stream. The force speedup observed above is the speedup of the concurrent, streamed force evaluation in comparison to the CPU.

The performance of each force evaluation kernel can, however, be observed in isolation to determine which kernels benefit most from being ran in parallel. This is done by simply running each kernel sequentially instead of concurrently in streams, evaluating their execution times and comparing them to the execution times of equivalent calculations on the CPU. Though there is no reason to execute the force evaluations sequentially on the GPU in a real world application, it can be beneficial to carry out performance evaluations on each force in isolation.

Figure 6.1 shows the speedups of each individual force evaluation compared to the equivalent CPU-based evaluations. As would be expected, the speedups tend to increase along with the size of the biomolecule being simulated, the only exception being the random forces evaluation with the 70s ribosome. The evaluation of the random forces and soft sphere angular forces benefit the most from the parallel nature of the GPU. The Fene force evaluation likewise benefited, but to a lesser extent. Evaluating the van der Waals forces benefited the least, as would be expected due to the large number of potential atomic operation collisions. Even with these collisions, these evaluations still ran over eight times faster with the 70s ribosome, demonstrating that the evaluation kernels were computationally involved enough to outweigh the drawbacks of a large number of atomic write collisions.
Table 6.1: Speedup of individual force evaluations
6.1.4 Neighbor and Pair List Improvements

The most critical optimization made with the molecular dynamics code was the implementation of the GPU-based sort and scan neighbor list algorithm. Figure 6.5 shows the execution times of the CPU-only neighbor list algorithm implementation, the “naïve” CPU- and GPU-based algorithm, and the fully optimized GPU sort and scan algorithm as well as the execution times for the unoptimized CPU-based molecular dynamics code.

![Figure 6.5: Execution times in seconds of simulations using different neighbor list implementation](image)

<table>
<thead>
<tr>
<th>Number of Beads</th>
<th>tRNA\text{pHe}</th>
<th>16s</th>
<th>30s</th>
<th>50s</th>
<th>70s</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU NL</td>
<td>283.23</td>
<td>389.31</td>
<td>641.89</td>
<td>1,191.87</td>
<td>2,204.87</td>
</tr>
<tr>
<td>Naïve NL</td>
<td>111.68</td>
<td>683.57</td>
<td>1,614.22</td>
<td>3,133.38</td>
<td>5,691.48</td>
</tr>
<tr>
<td>CPU-only NL</td>
<td>166.67</td>
<td>3,523.74</td>
<td>12,422.7</td>
<td>28,339.3</td>
<td>63,790.1</td>
</tr>
<tr>
<td>CPU Code</td>
<td>137.16</td>
<td>4,959.63</td>
<td>15,619.10</td>
<td>33,975.30</td>
<td>71,249.80</td>
</tr>
</tbody>
</table>

The importance of minimizing the amount of data that is transferred between
the GPU and the CPU is highlighted by the very similar performance of the CPU-based code and the GPU code that uses the original CPU-based implementation of the neighbor list algorithm. The original approach required a large amount of information to be transferred every time the neighbor and pair lists were evaluated. Even though the neighbor list and pair list evaluations represented a small section of the molecular dynamics algorithm, the transfers involved caused the code to be held back and perform only marginally better than it did on the CPU.

![Neighbor and Pair List Algorithm Comparison - Speedups](image)

<table>
<thead>
<tr>
<th>Number of Beads</th>
<th>tRNA$^{phe}$</th>
<th>16s</th>
<th>30s</th>
<th>50s</th>
<th>70s</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU NL</td>
<td>0.48</td>
<td>12.74</td>
<td>24.33</td>
<td>28.51</td>
<td>32.31</td>
</tr>
<tr>
<td>Naïve NL</td>
<td>1.23</td>
<td>7.26</td>
<td>9.68</td>
<td>10.84</td>
<td>12.52</td>
</tr>
<tr>
<td>CPU-only NL</td>
<td>0.82</td>
<td>1.41</td>
<td>1.26</td>
<td>1.2</td>
<td>1.12</td>
</tr>
</tbody>
</table>

Figure 6.6: Speedup of simulations using different neighbor list implementations

Implementing the naïve algorithm allowed some calculations to be offloaded to the GPU and reduced some of the memory transfers that were required. This allowed the performance to increase significantly compared to the CPU-only neighbor list
implementation. The implementation of the fully parallel sort and scan algorithm increased the performance even more by doing away with all memory transfers and performing all of the calculations in parallel.

The speedups shown in Figure 6.6 show the different performance characteristics of each of the three neighbor list algorithm implementations. When using the CPU-only neighbor list algorithm, the performance gains are fairly negligible, causing a reduction of performance in the tRNA\textsubscript{phe} unit and showing performance increases of only 12% to 41% in the other units. The naïve implementation, though not optimal, gave speedups that were higher than those given with the CPU-only algorithm. Using this algorithm also allowed the tRNA\textsubscript{phe} unit to perform faster, but this was only by 23%. However, the parallel sort and scan algorithm gave the best performance speedups by far, in most cases doubling that of the naïve implementation.
6.1.5 Fractional Execution Time

The proportion of time spent in each of the different sections of the molecular dynamics algorithm are shown in Figure 6.7. This data reveals trends in the behavior of the application with varying sizes of biomolecules.

![Execution Time Breakdown](image)

<table>
<thead>
<tr>
<th>Number of Beads</th>
<th>tRNA\textsuperscript{phe}</th>
<th>16s</th>
<th>30s</th>
<th>50s</th>
<th>70s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position Update</td>
<td>1.4%</td>
<td>0.9%</td>
<td>0.6%</td>
<td>0.3%</td>
<td>0.2%</td>
</tr>
<tr>
<td>Forces Update</td>
<td>13.3%</td>
<td>51.6%</td>
<td>46.6%</td>
<td>45.9%</td>
<td>35.4%</td>
</tr>
<tr>
<td>Velocity Update</td>
<td>1.2%</td>
<td>0.8%</td>
<td>0.5%</td>
<td>0.3%</td>
<td>0.2%</td>
</tr>
<tr>
<td>Logging</td>
<td>0.1%</td>
<td>1.9%</td>
<td>5.6%</td>
<td>7.4%</td>
<td>9.9%</td>
</tr>
<tr>
<td>Neighbor List Update</td>
<td>1.6%</td>
<td>6.4%</td>
<td>19.6%</td>
<td>26.0%</td>
<td>39.5%</td>
</tr>
<tr>
<td>Pair List Update</td>
<td>82.0%</td>
<td>37.9%</td>
<td>26.4%</td>
<td>19.1%</td>
<td>13.7%</td>
</tr>
<tr>
<td>Other</td>
<td>0.4%</td>
<td>0.5%</td>
<td>0.7%</td>
<td>1.0%</td>
<td>1.1%</td>
</tr>
</tbody>
</table>

Figure 6.7: Execution time breakdown
6.1.6 Hardware Dependent Speedups

There are many different CUDA capable GPUs being produced by NVIDIA. Each GPU has slightly different hardware characteristics and will therefore have varying levels of performance when running an application. To determine which GPU would be best suited to our study, we performed benchmarks on three different GPUs for comparison: the Tesla C2070, the GeForce 480GTX, and the GeForce 580GTX. The GeForce cards are consumer-grade components, whereas the C2070 is targeted primarily at scientific computing applications. The hardware specifications of each card are listed in Table 6.2.

<table>
<thead>
<tr>
<th>GPU</th>
<th>Tesla C2070</th>
<th>GeForce 480GTX</th>
<th>GeForce 580GTX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>448</td>
<td>480</td>
<td>512</td>
</tr>
<tr>
<td>Speed</td>
<td>1.15 GHz</td>
<td>1.4 GHz</td>
<td>1.54 GHz</td>
</tr>
<tr>
<td>Memory Size</td>
<td>6 GB</td>
<td>1.5 GB</td>
<td>1.5 GB</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>144 GB/s</td>
<td>117.4 GB/s</td>
<td>192.4 GB/s</td>
</tr>
</tbody>
</table>

Table 6.2: Hardware specifications for the Tesla C2070, GeForce 480GTX and GeForce 580GTX GPUs

As shown in Figure 6.8, the performance of the 580GTX is better than that of the 480GTX and the C2070 GPUs when running single precision molecular dynamics simulations. This is to be expected, as it is a newer GPU than the 480GTX and the benefits of the C2070—larger memory and better double precision throughput—would not be expected to be of much benefit in this case because all the data relating to all of the biomolecules can fit in the 1.5 GB of RAM that the 580GTX has and no double precision calculations are being made.

The speedups associated with the three GPUs are shown in Figure 6.9. Even though the hardware specifications of the 580GTX are not vastly different from those of the 480GTX and the C2070, the minor differences translate into a much better speedup.
The consumer-grade 580GTX outperformed the scientifically oriented C2070 in single precision simulations by a fairly large margin. This is to be expected based on the better hardware specifications of the 580GTX. This trend continues even when double precision values are used.
Figure 6.8: Execution times in seconds for different GPUs using single and double precision
<table>
<thead>
<tr>
<th>Number of Beads</th>
<th>tRNA&lt;sub&gt;phe&lt;/sub&gt;</th>
<th>16s</th>
<th>30s</th>
<th>50s</th>
<th>70s</th>
</tr>
</thead>
<tbody>
<tr>
<td>580GTX Single</td>
<td>0.48</td>
<td>12.74</td>
<td>24.33</td>
<td>28.51</td>
<td>32.31</td>
</tr>
<tr>
<td>480GTX Single</td>
<td>0.42</td>
<td>10.47</td>
<td>17.07</td>
<td>21.39</td>
<td>24.74</td>
</tr>
<tr>
<td>C2070 Single</td>
<td>0.35</td>
<td>8.25</td>
<td>14.28</td>
<td>17.76</td>
<td>20.41</td>
</tr>
<tr>
<td>580GTX Double</td>
<td>0.47</td>
<td>12.04</td>
<td>21.17</td>
<td>24.10</td>
<td>26.91</td>
</tr>
<tr>
<td>480GTX Double</td>
<td>0.41</td>
<td>9.59</td>
<td>14.32</td>
<td>18.37</td>
<td>20.77</td>
</tr>
<tr>
<td>C2070 Double</td>
<td>0.34</td>
<td>8.31</td>
<td>12.80</td>
<td>15.86</td>
<td>17.80</td>
</tr>
</tbody>
</table>

Figure 6.9: Speedups for different GPUs using single and double precision
6.1.7 CUDPP Performance

As we mentioned in Section 5.4, the performance of the CUDPP 2.0 library was significantly lower than that of the CUDPP 1.1.1 library, which led to our development of the algorithm outlined in Section 5.4 for neighbor list evaluation in simulations of large systems. As shown in Figure 6.10, the code utilizing the CUDPP 1.1.1 performs much faster than the code utilizing the CUDPP 2.0 library.

![CUDPP 1.1.1 and CUDPP 2.0 Execution Times](image)

<table>
<thead>
<tr>
<th>Number of Beads</th>
<th>tRNA\textsuperscript{tRNA}</th>
<th>16s</th>
<th>30s</th>
<th>50s</th>
<th>70s</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDPP 1.1.1</td>
<td>283.23</td>
<td>1,530</td>
<td>3,883</td>
<td>6,336</td>
<td>10,219</td>
</tr>
<tr>
<td>CUDPP 2.0</td>
<td>649.81</td>
<td>992.42</td>
<td>1,693.15</td>
<td>2,074.47</td>
<td>3,728.99</td>
</tr>
<tr>
<td>Time\textsubscript{2.0}/Time\textsubscript{1.1.1}</td>
<td>2.29</td>
<td>2.55</td>
<td>2.64</td>
<td>1.74</td>
<td>1.69</td>
</tr>
</tbody>
</table>

Figure 6.10: Execution times in seconds of code using the CUDPP 1.1.1 library and the CUDPP 2.0 library in seconds

We quantified the differences by observing the speedup of the CUDPP 1.1.1 code by calculating Time\textsubscript{2.0}/Time\textsubscript{1.1.1} so see how much faster the CUDPP 1.1.1 code ran. Depending on the the biomolecule being simulated, using CUDPP 2.0 would cause
the code to run 1.69 to 2.64 times slower than equivalent code using CUDPP 1.1.1. For this reason it was obvious that the CUDPP 1.1.1 library was a much better option for our purposes than the CUDPP 2.0 library.
6.2 Floating Point Performances

In this section we show the accuracy differences between using single and double precision code on both CPUs and GPUs. The accuracy was measured by finding the Pearson Correlation Coefficient (PCC), denoted by $r$, of the coordinates recorded in two trajectories. An $r$ value close to 1 indicates that the trajectories are closely correlated, whereas an $r$ values closer to 0 indicates no correlation and an $r$ values close to -1 indicates an inverse correlation. It was found that using single precision did not have a large effect on any of the biomolecules studied.

We also compare the energies associated with a single precision GPU-based simulation and a double precision GPU-based simulation in order to observe the energies associated with them. Some researchers using older generation GPUs observed an “energy drift” in their calculations using single precision numbers where the energy of the system slowly increased[8]. This is significant because the simulations were performed for an isolated system such that there should be no energy entering or escaping the system (i.e., the energy should be constant). Of course the energies of subsets of the system can and do change, but those changes must be compensated by an equal and opposite change. This concept is known as “detailed balance,” and it is a fundamental law in physics. Therefore, the observation of an energy drift would violate detailed balance and lead to unphysical dynamics as a result of the incorrect numerical calculations.

6.2.1 CPU Double Precision and CPU Single Precision

The MD simulations that we ran for this thesis were primarily done using single precision GPU code. To compare the accuracy of single precision MD simulations and double precision MD simulations, we first modified the original CPU-based double precision code to use single precision floating point values. This allowed us to see the
impact on accuracy that reduced precision would have under identical simulation conditions. We chose to compare the CPU single precision and double precision simulations before observing the accuracy of the GPU single precision simulations because of the variability introduced into the simulation by the undefined order of block execution. The $r$ values of the single and double precision comparisons are given in Table 6.3 and scatter plots of the coordinates for the tRNA$_{phe}$ and 70s simulations are given in Figure 6.11.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Beads</th>
<th>Pearson Correlation Coefficient ($r$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tRNA$_{phe}$</td>
<td>76</td>
<td>0.93374</td>
</tr>
<tr>
<td>16s</td>
<td>1,530</td>
<td>0.99914</td>
</tr>
<tr>
<td>30s</td>
<td>3,883</td>
<td>0.99864</td>
</tr>
<tr>
<td>50s</td>
<td>6,336</td>
<td>0.99968</td>
</tr>
<tr>
<td>70s</td>
<td>10,219</td>
<td>0.99964</td>
</tr>
</tbody>
</table>

Table 6.3: Pearson Correlation Coefficient for single and double precision CPU code

These results demonstrate that after one million timesteps the lower accuracy of single precision floating point numbers does not severely affect the accuracy of the simulation, producing $r$ values of greater than 0.9 in all cases.
6.2.2 CPU Double Precision and GPU Single Precision

The $r$ values for the double precision CPU and single precision GPU code are given in Table 6.4 and the scatter plots for the tRNA$^{phe}$ and 70s ribosomes are shown in Figure 6.12. Performing the comparison between the CPU double precision code and the GPU single precision code produced similar results as the comparison between CPU double and single precision simulations, but with one exception. The $r$ value of the tRNA$^{phe}$ unit was found to be slightly higher with the single precision GPU simulation than it was with the single precision CPU simulation and the scatter plot showed a slightly tighter grouping of coordinates. The most likely explanation for this gain in performance on the GPU is the extra precision given by the GPU’s FMA units, which perform a multiply and an add operation without rounding the result between each calculation.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Beads</th>
<th>Pearson Correlation Coefficient (r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tRNA$^{phe}$</td>
<td>76</td>
<td>0.98764</td>
</tr>
<tr>
<td>16s</td>
<td>1,530</td>
<td>0.99912</td>
</tr>
<tr>
<td>30s</td>
<td>3,883</td>
<td>0.99864</td>
</tr>
<tr>
<td>50s</td>
<td>6,336</td>
<td>0.99967</td>
</tr>
<tr>
<td>70s</td>
<td>10,219</td>
<td>0.99967</td>
</tr>
</tbody>
</table>

Table 6.4: Pearson Correlation Coefficient for single precision GPU code and double precision CPU code

Figure 6.12: Accuracy comparison between CPU double precision and GPU single precision
The high degree of accuracy of the single precision GPU simulation demonstrates that the use of single precision floating point numbers does not introduces significant inaccuracies into the calculation. For this reason we can safely say that the performance gains given by using single precision GPU simulations does not come with any significant drawbacks and can be safely used when performing MD simulations.

6.2.3 CPU Double Precision and GPU Hybrid

Double/Single Precision

As mentioned previously, due to the limitations of the CUDPP 1.1.1 library and the inability of CUDA to perform atomic operations on 64-bit data types, not all of the floating point values in the SOP model could use double precision values. The PDB distance array and the force array were required to be single precision values, but every other floating point value could be double precision.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Beads</th>
<th>Pearson Correlation Coefficient (r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tRNA\text{phe}</td>
<td>76</td>
<td>0.91914</td>
</tr>
<tr>
<td>16s</td>
<td>1,530</td>
<td>0.99938</td>
</tr>
<tr>
<td>30s</td>
<td>3,883</td>
<td>0.99864</td>
</tr>
<tr>
<td>50s</td>
<td>6,336</td>
<td>0.99968</td>
</tr>
<tr>
<td>70s</td>
<td>10,219</td>
<td>0.99968</td>
</tr>
</tbody>
</table>

Table 6.5: Pearson Correlation Coefficient for single and double precision CPU code

Figure 6.13: Accuracy comparison between CPU double precision and GPU double/single precision hybrid
The \( r \) values of the double precision CPU code and the hybrid double/single precision GPU code are shown in Table 6.5 and scatter plots of the coordinates for the tRNA\textsuperscript{phe} and 70s simulations are shown in Figure 6.13. The \( r \)-values show that the loss of precision when using single precision values for the two arrays mentioned above does not create a significant difference in the accuracy of the simulations.

### 6.2.4 Energy Drift

As shown in Figure 6.14, the different energies associated with both the single and double precision GPU simulations remain constant over the course of a 10 million timestep simulation. This demonstrates that there is no energy drift associated with these simulations and the physical law of detailed balance is not violated. We can therefore conclude that our simulations give an accurate picture of the underlying physical process of protein folding.

![Figure 6.14: Energy of single and double precision GPU simulations over 10 million timesteps](image)
6.3 Memory Improvements

Through the utilization of type compression and array reuse, the total memory footprint of the SOP model was drastically reduced. Reducing the amount of memory that is taken up by a simulation can be very important when using GPUs for MD simulations because the amount of memory available to a GPU is often much smaller than the amount that can be made available to a CPU. Though the C2070 GPU has a total of 6 GB of memory, the faster and more affordable GeForce 480GTX and GeForce 580GTX GPUs are limited to only 1.5 GB. Without the implementation of type compression and array reuse, the 70s ribosome could not be simulated on the GeForce series GPUs due to the amount of memory that it required.

<table>
<thead>
<tr>
<th>Beads</th>
<th>4 integers, 3 arrays</th>
<th>2 ushort, 3 arrays</th>
<th>ushort2, 2 arrays</th>
<th>ushort2, 1 array</th>
</tr>
</thead>
<tbody>
<tr>
<td>tRNA\textsuperscript{pre}</td>
<td>76</td>
<td>161 KB</td>
<td>66.5 KB</td>
<td>42.8 KB</td>
</tr>
<tr>
<td>16s</td>
<td>1,530</td>
<td>73.7 MB</td>
<td>33.6 MB</td>
<td>24.6 MB</td>
</tr>
<tr>
<td>30s</td>
<td>3,883</td>
<td>465 MB</td>
<td>177 MB</td>
<td>149 MB</td>
</tr>
<tr>
<td>50s</td>
<td>6,336</td>
<td>1,236 MB</td>
<td>547 MB</td>
<td>394 MB</td>
</tr>
<tr>
<td>70s</td>
<td>10,219</td>
<td>3.35 GB</td>
<td>1.71 GB</td>
<td>0.99 GB</td>
</tr>
</tbody>
</table>

Table 6.6: Memory footprint of various biomolecules using different memory arrangements

As Table 6.6 shows, the original implementation using four integers to represent the beads and type of interaction present in the Master, Neighbor and Pair list entries without any array reuse required that 3.35 GB of memory be available, which is more than twice the amount available to the GeForce GPUs. Representing these entries using two ushort and two uchar data types, however, reduced the memory footprint by almost one half, but still required 1.71 GB of memory to be used. However, type compression allowed the 70s ribosome to be simulated on the GeForce GPUs by reducing the memory footprint to a total of 1.38 GB.
Furthermore, the partial array reuse algorithm implemented in the CUDPP 1.1.1 code and the full array reuse algorithm implemented in the CUDPP 2.0 code reduce the memory footprint to 1.18 GB and 0.99 GB, respectively. Though these reductions were not necessary to simulate the 70s ribosome on the 480GTX or 580GTX, it allows it to be simulated on GPUs with smaller amounts of available memory, expanding the array of hardware that can be utilized when performing MD simulations.
Chapter 7: Conclusions

The optimizations performed with the MD simulation algorithm produced a significant performance increase when utilizing GPU hardware for general purpose computing. Simulations that previously would have taken months to complete can now be performed in a matter of days in many circumstances. Though single precision floating point numbers are now used in place of double precision numbers, the simulations still remain accurate and produce results that are scientifically relevant and useful.

When using GPUs in applications that are traditionally done by CPUs, there are sometimes ways to optimize algorithms that are quite apparent and easy to implement. However, the parallel nature of GPUs and their relatively limited amount of memory require that new approaches and techniques be developed when using them if peak performance is to be achieved. This often involves rethinking previous algorithms in ways that might not be entirely obvious at first. Once algorithms are optimized for GPU computing, however, the power of GPUs can more easily be tapped and significant performance gains can be achieved.

It is not unlikely that the performance increases achieved by CPUs in recent decades will be mirrored in the increases that GPUs architectures will see in upcoming years. Though CPU designers traditionally relied on increasing clock speeds for gaining performance, GPUs will rely on a combination of increasing clock speed and increasing parallelization. Countless technological advances were made as a result of the increasing performance of CPUs over the years and technology will see similar advances made as GPUs become more powerful and see more utilization in more mainstream applications.
Furthermore, the development of GPUs and CPUs that reside on the same physical die will open more door in computing by creating processing units that unify the strengths of both CPUs and GPUs. Instead of CPUs and GPUs existing as independent, discrete units, they will be unified and will not be held back by some of the limitations involved in GPU computing, such as latency in communication between CPUs and GPUs and slow memory transfers. The unification of memory spaces will likewise increase the range of applications that GPUs can be utilized in. Operations that are generally not done on the GPU due to the overhead of memory transfers will become feasible and allow GPUs to accomplish these tasks alongside of CPUs to great success.

The history of computers is one filled with a seemingly endless procession of technological advances that has continuously opened the door to solving a wide array of problems that would have been thought impossible just decades ago. The utilization of GPUs in general purpose calculations is a relatively recent development, but there have been very substantial gains in computer performance in the relatively short time they have been used. This trend will, like so many others in computing, continue to increase at very high rates.


Appendix A: Atomic Bucket Sort van der Waals Evaluation Algorithm

As we discussed in Sections 5.7 and 6.1.3, the evaluation of the van der Waals forces involves a large amount of atomic operations that degrades performance. By utilizing a scheme to cache some of the summations done with atomic operations in each block’s local shared memory, it is possible that performance might be increased. When performing any memory operation, global memory will always be significantly slower than local memory. Performing atomic operations is no exception.

In the current implementation of the van der Waals force evaluation kernel, each thread will perform an atomic operation on two entries in main memory, one for each bead present in one of the interactions. A very large portion of these operations will cause memory write collisions, causing a reduction in performance. If atomic operations were first performed on block-level memory and global memory writes were delayed until the last step of execution, it is possible that the performance would be boosted.

To implement this setup, at the block level two data structures will be created. The first will be a variable-sized “bucket array” that will act as a cache to store the magnitude of the van der Waals forces acting on each bead that the block’s threads will act upon. This structure will initially be empty, but will have new values added to it as different threads begin calculating values. The second structure will be an “information array” that will keep track of which beads have had entries added to the cache structure and at what index in the structure the bead’s information is stored.

At the thread level, when an individual thread calculates the forces generated by an interaction between two beads, it will check the information array to see if a bucket
for each bead has already been created. For each of the two beads, the information array will be checked to see if a bucket already exists for the bead. If it does, the force calculated by the thread will be added to the force already present in the bead’s bucket. If a bead does not have an entry in the information array, the following three steps will be done atomically: a new entry will be added to the bucket array, the information array will be updated to indicate that the bead now has a bucket and the index of that bucket in the bucket array, and the force will be added to the bucket. At the end of each block’s execution, a global atomic add operation will be done for each bead that was examined within the block. This will add the forces calculated by the block to the global array of forces.

In situations where a block performs a large number of atomic operations on a single bead’s memory location, this algorithm could serve to reduce the amount of time taken to write the summation of forces to main memory. There will still be a large number of atomic operations performed in local memory, but for each block there will be only one global atomic write performed per bead. The atomic writes performed in local memory will execute much faster than if they were done to global memory, which will allow the block to execute much faster. Without utilizing this algorithm, however, each of the atomic operations are done in global memory, which is why the speedup seen when evaluating the van der Waals forces was comparatively lower than when evaluating the other forces.

Though the total execution time of all of the atomic actions will be decreased by this algorithm, it remains to be seen whether or not the overhead introduced by the bucket array and the information array would cause the overall execution time to drop compared to the current implementation. However, if managing the two arrays could be implemented in an efficient manner, the total number of atomic actions performed by a block could be as low as $N + 1$, where $N$ is the number of threads in a block,
whereas in the current implementation, a total of $N^2$ atomic operations per block always take place. In the best case scenario, each thread within a block will operate on a set of interactions between a single bead $b_0$ and a $N$ other beads, $b_i$. There will be one bucket created for bead $b_0$ and $N$ buckets created for the $b_i$ beads. Therefore, at the end of the block’s execution, one global atomic operation will be performed for $b_0$’s bucket and $N$ for the $b_i$ buckets, for a total of $N+1$ atomic operations. However, it is still possible that $N^2$ total atomic memory operations could take place, depending on how the interactions present in the Neighbor and Pair List end up being sorted.
Tyson J. Lipscomb
6 Murphy Way
Barboursville, WV 25504
(304) 634-0475
TysonLipscomb@gmail.com

Education

2012 M.S. Wake Forest University, Winston-Salem, NC
Master of Computer Science (GPA 3.95)
Thesis: “GPU-Optimized Molecular Dynamics Simulations”

2010 B.S. Marshall University, Huntington, WV
Major in Computer Science (GPA 3.92) and Applied Mathematics (GPA 4.0)
Cumulative GPA 3.86

Honors and Awards

John Marshall Scholar Marshall University, 2005
PROMISE Scholarship Recipient Marshall University, 2005
National Society of Collegiate Scholars Member Marshall University, 2006
Dean’s List Student Marshall University, Fall 2005-Spring 2010
Honor Student Marshall University, Fall 2005-Spring 2010
Computer Science Excellence Fellowship Wake Forest University, Fall 2010-Spring 2012
Upsilon Pi Epsilon Wake Forest University, Fall 2011-Present

Skills

Programming C, C++, CUDA, C#, Java, Python
Advanced Libraries CUDA Data Parallel Primitives (CUDPP) Library, Java Advanced Imaging API,
Gnu Scientific Library (GSL)
VR Experience Organic Motion motion capture system, Unity3D virtual environment creation,
Integration of VR devices with virtual environments, Blender 3D
Operating Systems Windows, Linux
Development Visual Studio, Netbeans, Eclipse
Miscellaneous Remote Linux administration, VMware virtual machine management,
Software development using Atmel AVR microcontrollers
Publications and Presentations


- “GPU-Based Molecular Dynamic Simulations Optimized with CUDA Data Parallel Primitives (CUDPP) and CURAND Libraries.” Competitively selected poster presented at the NVIDIA GPU Technology Conference, May 14-17, 2012, San Jose, California.


Research Experience

- Research assistant with Dr. Samuel S. Cho, Department of Computer Science, Wake Forest University, Spring 2011-Spring 2012. Developed and optimized a GPU version of a molecular dynamics simulation software in the CUDA programming language.

  - GPU Programming
  - Hardware Analysis and Assembly
  - Molecular Dynamics
  - Biophysical Structural Analysis
  - Parallel CUDA Optimizations
  - Parallel Algorithm Development

- Research assistant for the Center of Environmental, Geotechnical and Applied Sciences (CEGAS), January 2008-October 2009. Conducted research into emerging technological advancements for CEGAS in the following fields:

  - Augmented Reality Systems
  - Virtual Collaboration
  - Stereoscopic Visualization
  - Virtual Mine Training Scenarios
  - Virtual Medical Training
  - 3D Content Development

- Completed mathematics capstone research on Kurt Gödel's Incompleteness Theorems and the implications thereof, Fall 2009

- Conducted research into medical image analysis theory for Engineering and Science Problem Solving Honors Seminar, Fall 2007

Coursework

(* denotes class taken for Honors credit)

**Computer Science (Graduate)**
- Operating Systems
- Introduction to Numerical Methods
- Theory of Algorithms
- Object-Oriented Software Engineering
- Theory of Computation
- Nonlinear Optimization
- Principles of Compiler Design
- Machine Learning

**Computer Science**
- Computer Science I, II
- Algorithm Analysis and Design
- Programming Languages
- Artificial Intelligence
- Software Engineering I, II
- Internetworking
- Operating Systems
- Database Engineering
- Computer Graphics*
- Senior Project

**Mathematics**
- Calculus I*, II*, III
- Linear Algebra
- Introduction to Higher Math
- Probability and Statistics I, II
- Discrete Structures
- Differential Equations

**Other Related Coursework**
- Circuits I
- Engineering and Science Problem Solving Honors Seminar*
- Scientific and Technical Writing
Extracurricular Activities

**Member** Marshall University's ACM International Programming Competition team, 2006, 2007, 2009

**Member** Leadership council for peer-led Christian ministry and Bible study The Vine, 2007-Present

**Leader** Small group/Bible study for The Vine, 2007-Present

**Leader** Discussion group/Bible study for Graduate Christian Fellowship, Spring 2011-Spring 2012

Other Experience


**Technical Support Staff** Lipscomb Walker Insurance Agency, 2006-2010

**Technical Support** Freelance, 2006-Present

Volunteer Experience

**Volunteer** Prepared and served Sunday meals at the Huntington City Mission with The Vine, 2007-2010

  Supervisor: Gary Moore, Christ Community Church

References

References available upon request.