GPU-OPTIMIZED HYBRID NEIGHBOR/CELL LIST ALGORITHM FOR
COARSE-GRAINED MOLECULAR DYNAMICS SIMULATIONS

BY

ANDREW J. PROCTOR

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 2013
Winston-Salem, North Carolina

Approved By:
Samuel S. Cho, Ph.D., Advisor
David J. John, Ph.D., Chair
William H. Turkett, Ph.D.
Acknowledgements

I would like to thank my advisor, Dr. Samuel S. Cho, for his guidance and insight while conducting my graduate research and writing this thesis. He is very intelligent and hardworking and has been the perfect role model while working with him. Without his input or encouragement, this thesis would not have been possible.

I would also like to thank Dr. David J. John and Dr. William H. Turkett 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 an environment that fosters learning and scholarship in Computer Science. Their help and support has helped me better my understanding of Computer Science and has better equipped me to solve the problems I will face in the future.

I would like to thank Dr. Joshua Anderson for his help with HOOMD. His willingness to help implement our SOP model Hamiltonian into HOOMD was crucial and this particular analysis would not have been possible without his assistance.

My parents Rick and Melody Proctor, my sister Megan, and all of my friends have been a constant source of support and encouragement throughout my time at Wake Forest University. They have all helped shape me into the person I am today and I would have never made it here without them.

Lastly, I am grateful for the monetary support from the National Science Foundation as this work was partially funded by (CBET-1232724).
# Table of Contents

Acknowledgements ........................................................................ ii

List of Figures ........................................................................... v

Abstract .................................................................................... vi

Chapter 1 Overview of Supercomputing ........................................ 1
  1.1 History of Supercomputing .................................................. 2
  1.2 Supercomputing Milestones ............................................... 2

Chapter 2 Graphics Processing Units ............................................ 8
  2.1 Computer Graphics Technology and Parallel Computations ..... 9
  2.2 GPU Computing for General Purpose Programming .......... 10
  2.3 General Purpose GPU Programming Languages ............... 11
    2.3.1 CUDA for General Purpose GPU Programming .......... 11
    2.3.2 GPU Programming Bottlenecks ............................... 14

Chapter 3 Background on Molecular Dynamics ............................ 16
  3.1 Biomolecular Folding Processes ....................................... 17
  3.2 Molecular Dynamics Algorithm ...................................... 17
  3.3 Coarse-Grained Simulation Models ................................. 21
  3.4 Molecular Dynamics with GPUs .................................... 22

Chapter 4 Background on Cutoff Methods ................................. 24
  4.1 Verlet Neighbor List Algorithm ...................................... 25
  4.2 GPU-Optimized Parallel Neighbor List Algorithm ............. 27
  4.3 Cell List Algorithm ..................................................... 29

Chapter 5 GPU-Optimized Cell List and Hybrid Neighbor/Cell List Algorithms 32
  5.1 Possible Implementations of the Cell List Algorithm on GPUs 32
  5.2 Parallel Cell List Algorithm ........................................... 33
  5.3 Hybrid Neighbor/Cell List Algorithm ............................... 35
Chapter 6 Performance Analyses

6.1 Implementation of the SOP model on HOOMD

6.2 Performance Comparison with HOOMD

6.3 Hybrid Neighbor/Cell List Performance Improvements

6.4 Comparisons of MD Simulations Execution Times with GPU-Optimized Cutoff Method Algorithms

Chapter 7 Conclusions

Bibliography

Appendix A Cell List Kernel: Attractive List

Appendix B Cell List Kernel: Repulsive List

Appendix C CUDA Kernel: Compute Cell IDs

Appendix D HOOMD Input Scripts

Appendix E HOOMD .xml File

Curriculum Vitae
# List of Figures

1.1 ENIAC ................................................................. 3  
1.2 ASCI Red ........................................................... 4  
1.3 Intel Microprocessor Transistor Count .......................... 5  
1.4 NVIDIA GPU ....................................................... 6  
1.5 Comparison of CPU and GPU Performance ....................... 7  
2.1 CUDA Thread, Block and Grid Hierarchy ......................... 12  
2.2 Dynamic Block Execution ....................................... 13  
4.1 Illustration of long-range interaction cutoff methods ........... 26  
4.2 Pseudocode for a CPU-based Neighbor List algorithm .......... 27  
4.3 Serial and parallel algorithms for constructing the neighbor and pair lists 29  
4.4 Schematic of the Cell List algorithm .......................... 30  
5.1 Cell List Algorithm ............................................... 33  
5.2 Periodic Boundary Conditions .................................. 34  
5.3 Progression of cutoff algorithms ............................... 36  
6.1 Comparisons of execution times with HOOMD .................... 40  
6.2 Execution times comparison for the Neighbor and Hybrid Lists 41  
6.3 Hybrid Neighbor/Cell list speedups over the parallel Neighbor List and original Hybrid List ....................... 42  
6.4 Execution times .................................................. 43
Abstract

Molecular Dynamics (MD) simulations provide a molecular-resolution picture of the folding and assembly processes of biomolecules, however, the size and timescales of MD simulations are limited by the computational demands of the underlying numerical algorithms. Recently, Graphics Processing Units (GPUs), specialized devices that were originally designed for rendering images, have been repurposed for high performance computing with significant increases in performance for parallel algorithms. In this thesis, we briefly review the history of high performance computing and present the motivation for recasting molecular dynamics algorithms to be optimized for the GPU. We discuss cutoff methods used in MD simulations including the Verlet Neighbor List algorithm, Cell List algorithm, and a recently developed GPU-optimized parallel Verlet Neighbor List algorithm implemented in our simulation code, and we present performance analyses of the algorithm on the GPU. There exists an \( N \)-dependent speedup over the CPU-optimized version that is \( \sim 30 \times \) faster for the full 70s ribosome (\( N=10,219 \) beads). We then implement our simulations into HOOMD, a leading general particle dynamics simulation code that is also optimized for GPUs. Our simulation code is slower for systems less than around 400 beads but is faster for systems greater than 400 beads up to about 1,000 beads. After that point, HOOMD is unable to accommodate any more beads, but our simulation code is able to handle systems much larger than 10,000 beads. We then introduce a GPU-optimized parallel Hybrid Neighbor/Cell List algorithm. From our performance benchmark analyses, we observe that it is \( \sim 10\% \) faster for the full 70s ribosome than our implementation of the the parallel Verlet Neighbor List algorithm.
Chapter 1: Overview of Supercomputing

Today, supercomputers play an imperative role in multiple scientific fields of study because a supercomputer can perform many mathematical calculations in a short period of time. Science is not restricted to just experimentation and observation, but a third way scientists can attempt to study a problem is through mathematical modeling[13]. Scientists and engineers often use mathematical models to describe or predict what will happen in very complex systems. Predictive models of weather, climate, earthquakes, the formation of galaxies, the spread of diseases, and the movement of traffic in big cities are just a few examples of phenomena scientists typically study[13]. One such class of models studies the possible pathways of folding for biomolecules. These simulations, known as Molecular Dynamics (MD), involve an atomistic representation of a biomolecule where the movement of atoms are calculated using Newton’s equations of motion. To do this, all forces acting on an atom are summed up and used to determine the atom’s acceleration. From acceleration, velocity and positions can be derived to determine the biomolecule’s new velocity and position at a future timestep. These calculations are made over many timesteps while representing biomolecules with thousands of atoms. As more atoms are represented, this model becomes increasingly complex. A problem of this nature clearly can not be solved by a person or even a single computer because the model is so complex and requires many mathematical calculations[13]. In fact, “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”[30].
1.1 History of Supercomputing

Supercomputers are often comprised of thousands of processors and gigabytes of memory so that they are capable of performing the immense number of calculations required by some applications. The very definition of a supercomputer is a computer at the cutting edge of current processing capacity, particularly with regards to the speed of the calculations it is able to perform. The speed or number of calculations a supercomputer is capable of is measured in floating point operations per second (FLOPS). The number of FLOPS a computer can perform is often prefixed with the common metric prefixes kilo-, mega-, etc., when dealing with large values, and it is the most common and reliable way to compare two different supercomputers[30]. For example, the current title holder of the world’s fastest supercomputer, Titan at Oak Ridge National Laboratory, is capable of achieving roughly 17 PetaFLOPS or 17 quadrillion operations per second, and it is a testament to how far high performance computing has come since the conception of the ENIAC supercomputer, only a little more than half a century ago.

1.2 Supercomputing Milestones

The first general purpose Turing-complete electronic supercomputer ever built was ENIAC, constructed for the United States Army in 1946[45]. Like most supercomputers, ENIAC took up a large amount of space, covering a total of 1,800 square feet[8]. Once completed, ENIAC cost a total of roughly $500,000 in 1946[8] (equivalent to $6 million today) and could run at up to 300 FLOPS[8]. ENIAC was one thousand times faster than any other calculating machine at the time and could perform 5,000 additions, 357 multiplications or 38 divisions in one whole second[7]. Although these measures are trivial by today’s standards, these calculations had to be performed by
people. Even with the aid of special devices, these calculations were considered to be an extraordinary technological leap in terms of the types of problems one could solve. The Army used ENIAC for two notable classes of problems; calculating artillery trajectory tables[8] and performing simulations relevant to the creation of the hydrogen bomb[26]. Without ENIAC, these calculations would have been very time consuming or practically impossible to complete given the technology that was available at the time. Despite its monstrous frame and low-poweredness in comparison to today’s technology, ENIAC played an important role in the development of future computers because it demonstrated the value of a computational instrument in several practical scenarios.

Over the next few decades, several noticeable trends in computers would arise. As computers became more accessible to consumers, vast improvements in the technology would occur. One of the most influential improvements was the invention of the modern day transistor. “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 transistors. 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”[30].

Not only did computer hardware get smaller, but it also got faster. Technology began advancing at a rate that allowed the number of transistors that could fit into a given space to roughly double every 18 months, leading to exponential increases in memory sizes and processor performance, a trend described by the well known Moores Law[29, 31]. Figure 1.3 shows the transistor count of Intel microprocessors from 1971 to 2004 and we see that the number of transistors used per microprocessor roughly doubles every two years just as Moore’s Law predicted. 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[27]. Supercomputers didn’t have to be special, custom build machines anymore. This led to the construction of supercomputing clusters
that were comprised of a collection of individual computers. The individual computers
were networked together to allow for communication between each individual machine
to “coordinate their computations and perform as if they were a single unified ma-
chine”[30]. The clusters were an inexpensive alternative to traditional vacuum tube
supercomputers and, in turn, resulted in more supercomputers being built.

ASCI Red was another major milestone in the advancement of supercomputing.
Completed in 1997 and utilizing a total of 9,298 Intel Pentium Pro processors[6], ASCI
Red was the first computer to enter the world of TeraFLOP computing by being capa-
bile of performing one trillion floating point operations per second[6]. Also of particular
note was that it was constructed primarily using off-the-shelf parts which meant that
its construction and maintenance was significantly less specialized. ENIAC required
a collection of custom made components and therefore programming changes would
take the technicians weeks to complete[7]. Using off the shelf parts was advantageous
because ASCI Red could be designed and built without “enlisting a large team of
electrical and computer engineers”[30]. When it came to physical size, there was little that separated ASCI Red from ENIAC. However, ASCI Red was able to achieve performance levels that dwarfed those of its predecessor.

Traditionally, most supercomputers had been built by relying solely on central processing units (CPUs) and using many of them to build a collectively powerful computing machine. However, graphics processing units (GPUs) made significant advances in power and programmability that have led to an outpacing of CPUs, shown in Figure 1.5. These advances have resulted in many to begin using GPUs in conjunction with CPUs in certain hybrid applications that utilize both types of processors[30]. The final milestone we cite is the Tianhe-1A because it became the world’s fastest supercomputer in 2010 by using GPUs and CPUs in a heterogeneous computing environment. The Tianhe-1A uses 7,168 NVIDIA Tesla GPUs and 14,336 CPUs to achieve 2.507 PetaFLOPS (2.507 ×10\(^{15}\) FLOPS) of computing power[23]. Using GPUs didn’t just make the Tianhe-1A into a performance marvel, it made it more energy efficient too. “A 2.507 PetaFLOP system built entirely with CPUs would consume more than 12 megawatts. Tianhe-1A consumes only 4.04 megawatts, making it 3 times more power efficient; the difference in power consumption is enough to provide electricity to over 5000 homes for a year”[23]. The Titan supercomputer at Oakridge National
Laboratory is currently the world’s fastest supercomputer, and it uses a similar hybrid architecture with CPUs and GPUs. This trend will likely continue into the foreseeable future.
Chapter 2: Graphics Processing Units

By 1980, the need for better computer graphics was brought to light by computer programmers who were seeking to create “3D models for display on a standard monitor”. This need could not be met by the standard CPU due to the complex calculations involved in generating computer graphics. These rendering of images and the operations required to be computed were limiting the CPU from performing other tasks. In order to free up the CPU to run various system tasks, hardware companies created specialized accelerators called graphics cards to render more realistic images in a timely fashion. The specialized hardware was not able to perform the amount of diverse tasks a CPU was capable of, but they did what they were designed to do, namely the computation of floating point operations in parallel, with respectable proficiency[30].

In time, the competition between different vendors were motivated by performance increases in floating point operation computation while decreasing the overall cost of the graphics card devices. Additional functionalities were added to graphics cards to allow a greater range of operations and in 1999 NVIDIA introduced the graphics processing units (GPUs) to computing. 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”[33].
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. To create an image for display on a 2D screen to represent a 3D scene, a GPU must perform a basic set of rendering operations called graphics primitives. Since an image is composed of many points called vertices that are grouped into triangles and are collectively composed into an image[5], the GPU must take the set of triangles and compute various properties such as lighting, visibility, and relative size to render an image. These operations are performed independently[27]. The majority of these computations are floating point operations[27] and GPUs were optimized specifically for performing these operations quickly[30].

<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

Since a GPU is a specialized device for only one set of operations, namely the computing of many floating point operations very quickly, it is designed to do this very, very well. On the other hand a CPU’s function is to perform many tasks so it must have the overhead of not only computing floating point operations but also maintaining other devices such as the monitor, keyboard, and optical driver and managing the communication between them. Essentially, the GPU must sacrifice all of the flexibility of the CPU to perform its specialized operations optimally. The vast differences in flexibility between the CPU and GPU are reflected in their architectures. In order to perform a wider range of tasks, the cores of a CPU have much higher clock
speeds in comparison to the cores on a GPU. However, on average, CPUs have on the order of four to eight computational cores, while the GPU has upwards of hundreds of cores. While these cores generally have slower processing speeds, the larger number of cores on the GPU makes them better suited to many types of parallel tasks (Table 2.1)[30].

2.2 GPU Computing for General Purpose Programming

As GPUs became increasingly advanced, many people began to realize the practical applications of using GPUs for general purpose computing. As we mentioned in Chapter 1, supercomputers are used to solve various types of problems that require the independent calculations of floating point numbers. Due to their parallel architecture, GPUs are well suited to solve these types of problems. However, to use GPUs for general purpose computing, these calculations must be translated to a set of rendering operations that the GPU can understand. While it’s not impossible, writing such an application would be very difficult for the standard programmer[27, 30].

The progression of video games and the desire of their designers to incorporate a more realistic graphical experience in their titles led to GPU manufacturers allowing for more custom computations on the GPU. Video game designers wanted the freedom to use their own rendering functions instead of being confined to fixed function units and the hard-coded algorithms GPU developers had built into their hardware. Parts of the rendering process were changed to allow for more flexible units that could perform these custom computations[27]. Researchers also welcomed this change because it opened the door to use GPUs for a wider range of scientific applications. The change signalled a shift in the industry from rigid GPUs with limited flexibility to GPUs that were more programmable[30].
2.3 General Purpose GPU Programming Languages

With more programmable GPUs on their agenda, NVIDIA, the leading manufacturer of GPUs, hired Ian Buck to begin working on their Compute Unified Device Architecture (CUDA) library[12]. NVIDIA hired Buck primarily for his role and experience in developing Brook, one of the first implementations of a general purpose GPU programming language. Buck, along with researchers at Stanford University[9], developed Brook with the intentions of providing programmers with the ability of using GPUs for use in non-graphical, general purpose computations in a programming environment similar to the C programming language. While the hardware used for the Brook project was not nearly advanced as GPUs today, programs written in the Brook programming language were still capable of run times up to eight times faster than equivalent codes executed on the CPU[9, 30]. NVIDIA took notice of the Brook project’s success and wanted to emulate that success with their own CUDA programming library.

2.3.1 CUDA for General Purpose GPU Programming

One of the most obvious characteristics of CUDA is that it looks very similar to C/C++ code. Rather than forcing programmers to utilize GPUs through graphics APIs, NVIDIA’s top priority was to create a programming environment that was familiar to coders and also one that allowed access to the GPU with as much ease as possible. As a result, programmers could focus on writing parallel algorithms instead of struggling with the mechanics of the language. CUDA uses a heterogeneous computing environment that is comprised of a host that is traditionally a CPU and at least one device that functions as a parallel processor (GPU) [27]. In this way, a program has access to both the CPU and GPU with some portions of the program executed on the CPU and others offloaded to the GPU. The heterogeneous nature of
CUDA allows a program to utilize the strengths of both processors\cite{cuda}.

In CUDA, to execute a section of code in parallel one would create and launch a “kernel - the CUDA equivalent of a parallel function call”\cite{cuda}. A kernel by definition is a function that runs on the device and once it is finished it returns control back to the host or CPU. The kernel spawns a collection of sub-processes known as threads and each thread is responsible for carrying out the set of instructions specified by the kernel\cite{cuda}.

Threads on a GPU are considered extremely lightweight because they require relatively little creation overhead as compared to CPU threads. CUDA uses thousands of these threads to achieve efficient parallelism. To assist the programmer in managing the numerous threads in a parallel CUDA application, threads are arranged into a multilevel hierarchy as shown in Figure 2.1. When a kernel gets launched, it creates a grid of thread blocks. The grid arranges the thread blocks in two dimensions and
threads inside the thread blocks can be arranged in three different directions. Each thread block executes independently of one another and they do not execute in any set order. A single core on the GPU will carry out the execution of a thread block and all its associated threads. There is a limit on how many blocks can be executed at once on the GPU. Depending on the number of computational cores a GPU has at its disposal, the thread blocks will be scheduled accordingly. If there are more blocks than cores on the GPU, multiple blocks will be assigned to each core and each set of blocks will be carried out in succession. Figure 2.2 shows how the thread blocks may be scheduled for two GPUs with a different number of cores[30].

There are several types of memory available within a CUDA program and they are arranged in a hierarchical structure similar to threads, thread blocks, and grids. At the very top, “general purpose global memory, read only constant memory and read-only texture memory” are available to every thread and every thread block[30]. In addition each block has its own shared memory. Shared memory is much smaller,
but has a latency that is 100x lower than global memory. Finally, at the lowest level, each thread has a small amount of private memory. Private memory is the fastest and most limited memory available in CUDA[30].

NVIDIA GPUs implement an execution model that is a generalization of the single instruction multiple data paradigm (SIMD). As is well known in parallel computing, in a computer hardware with a SIMD architecture, a set of computational units like processor cores access different parts of memory and execute the exact same set of instructions on only the region of data allocated to it. Clearly, SIMD architectures are optimal for executing massively parallel tasks. In the NVIDIA GPU model, multiple threads are executed at the same time and follow the same set of instructions of the kernel, resulting in a single instruction, multiple thread (SIMT) paradigm[30]. Of course, CUDA also has functionality for barrier synchronization by halting CPU execution and waiting for a specified kernel call to complete, and this can be done either 1) after a kernel is finished executing to make sure all global threads have completed their tasks before progressing, or at any point in a kernel execution such that all threads in a shared memory block must complete before progressing [30].

2.3.2 GPU Programming Bottlenecks

Though GPUs lend themselves to massively parallel applications, they still have two major downfalls. The transfer of data between the host and device and vice versa is arguably the biggest performance bottleneck to overcome. Due to the physical distance between the CPU and GPU, all relevant data must pass through the PCI Express bus. To minimize the memory transfer bottleneck, there should only be two transfers: one at the beginning of the program and one at the end to copy back any relevant results[30].

With some programs there are inherently serial parts that must be executed.
Although GPUs have a large number of computational cores, these cores are optimized to calculate floating point numbers. This specialization comes at the cost of flexibility. The lack of flexibility is less than ideal to perform the general calculations/instructions in serial portions of code. In addition to a lack of flexibility and slower clock speeds, the use of only one core leaves the majority of the GPU idle and leads to a very low utilization of resources. Serial portions of the code should be rewritten to make full use of the GPU’s parallel architecture[30].
Chapter 3: Background on Molecular Dynamics

An area of active research in high performance computing is molecular dynamics simulations of biomolecules. All living cells are composed of a complex milieu of molecular machines that perform life processes or functions[1]. They are composed of many different types of biomolecules including proteins, RNA, DNA, lipids, and sugars[1]. Although these biomolecules are classically seen in textbooks as static images, they are actually dynamic and rapidly fluctuating around its local environment or transitioning from one state to another. These biomolecules are exceedingly tiny, but they are very active to carry out functions that can be very generally described as storing information and accomplishing tasks that allow cells to live and reproduce. To quote Richard Feynman, “Everything that living things do can be understood in terms of the jiggings and wigglings of atoms.”[20]. This is similar to the problems that computer scientist face because computers are meant to store enormous amounts of information in a tiny little space, but there are also complex algorithms to perform tasks that solve problems. In the same spirit, biomolecules move, form, and assemble to for complex structures[4, 16] that store our genetic information and carry out functions including enzyme catalysis, structural support, transport, and regulation, just to name a few [18], so that that organisms can live.

In this thesis, I will focus on two classes of biomolecules called proteins and RNA (ribonucleic acids). Proteins are made of 20 naturally occurring types of amino acids that are strung together like beads on a string. On average, the length of a protein is about 400 amino acids, and they can interact with other proteins or other biomolecules to form complex structures. Similarly, RNA are made of 4 naturally occurring types of nucleic acids that are also linearly strung together, and they can also interact
with other biomolecules to form complexes. Biomolecular complexes assemble into biomolecular machines.

### 3.1 Biomolecular Folding Processes

A biomolecule converts from an unfolded state to a folded state through many different possible pathways through a process called folding. [14, 17, 19, 46]. We note that the study of biomolecular folding processes is distinct from the field of biomolecular structure prediction, which involves using statistical methods to predict the folded structure of a biomolecule[28]. This approach has been very successful in determining the final folded structure of biomolecules but does not give any information about the structures of the biomolecule as it is transitioning between its unfolded and folded states.

Biomolecules will generally self-assemble in solution into a specific structure in the absence of any helper molecules to properly perform their cellular functions[4, 16]. However, a misfolding event could also result that can be detrimental. Many diseases such as Alzheimer’s and Parkinson’s are caused by problems that occur during the folding processes[15]. It is therefore crucial that the folding process of different biomolecules be understood if these diseases are to be more effectively combated.

### 3.2 Molecular Dynamics Algorithm

Biomolecular folding processes can be studied using experimental techniques, but the direct observation of folding processes is currently beyond the state of the art. This is largely because the resolution of the timescales of experiments are inaccessible to folding events, which are on the microsecond timescale for the fastest folding proteins[17]. A computational approach that does not suffer from this resolution limitation is molecular dynamics simulations. Using molecular dynamics, one can study
the movement of atomic of molecule structures and the way they interact and behave in the physical world using computational methods. The physical description of the biomolecule in MD simulations is determined by an energy potential that determines how the interacting atoms or molecules behave and interact. Using molecular dynamics simulations, the entire folding process can be studied by simulating and recording the position of each part of the system at predetermined intervals of about one picosecond (10^{-12} seconds)[17]. By splicing together 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 like a movie.

There are two sets of limitations of MD simulations, and it is a problem of length-scale and time-scale. Due to the computational demanding algorithms of MD simulations, there are limits on the size of the system and the length of time one can perform these simulations. Ideally, one would use a detailed description of the biomolecule, and this can be accomplished using quantum mechanical calculations. However, one must include a description of all of the electrons in the biomolecule, which may actually be irrelevant to the problem at hand. Others use empirical force field MD simulations where classical mechanics is used instead by approximating biomolecules at an atomistic resolution. 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 [42], 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. Still others use coarse-grained MD simulations where groups of atoms are approximated as beads[11, 38]. By using a less detailed description of a biomolecule, we gain instead larger systems and longer timescales that one can simulate.
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, and other interactions. We will describe in detail the Self-Organized Polymer (SOP) model energy potential[24, 35] 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 $\vec{F} = m\vec{a} = -\zeta \vec{v} + \vec{F}_c + \vec{\Gamma}$ 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 $\vec{\Gamma}$ 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 \( \vec{r}(t) + \Delta t \vec{v}(t) + \frac{\Delta t^2}{2m} \vec{F}(t) \) 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.

3.3 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 acide or nucleotide (the C$_\alpha$ or C2’ position) for proteins or RNA, respectively, thereby reducing the total number of simulated particles[22, 35].

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

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

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

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

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

\[
+ \sum_{i=1}^{N-3} \sum_{j=i+3}^{N} \epsilon_l \left( \frac{\sigma}{r_{i,j}} \right)^6 (1 - \Delta_{i,j})
\]

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: $k = 20\text{kal/(mol} \cdot \text{Å}^2)$, $R_0 = 0.2\text{nm}$, $r_{i,i+1}^0$ 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 $t$. 

21
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 8Å in the folded structure. If beads \(i\) and \(j\) are a native pair, \(\Delta_{i,j} = 1\), otherwise \(\Delta_{i,j} = 0\). The \(r_{ij}^0\) 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 radius of the interacting particles, is chosen to be 3.8Å, 5.4Å, or 7.0Å 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.

### 3.4 Molecular Dynamics with GPUs

An approach that is particularly well-suited for performing MD simulations with a demonstrated track record of success is the use of graphics processor units (GPUs) (Fig. ??). Recently, several studies developed and demonstrated that GPU-optimized MD simulations, including the empirical force field MD simulation software NAMD [39] and AMBER [21] and the general purpose particle dynamics simulation software suites HOOMD [3] and LAMMPS [36], can significantly increase performance. MD simulations lend themselves readily to GPUs because many independent processor cores can be used to calculate the independent set of forces acting between the beads in a MD simulation.

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 4:  Background on Cutoff Methods

The long-range interaction between two objects is generally inversely related to the distance between them. Although this relationship does not always hold in close proximity, as we see below, two objects that are sufficiently far apart have essentially no interaction between them. A classical example of this is Newton’s Law of Universal Gravitation, which was first described in his seminal work, *Principia* in 1687[32].

In his description of planetary motion, Newton empirically introduced the relation (in modern terms)[10]

\[ F = G \frac{m_1 m_2}{r^2} \]

where \( G \) is the universal gravitational constant, \( m_1 \), and \( m_2 \) are the masses of the two interacting objects, and \( r \) is the distance between them. The interacting force, \( F \), diminishes as \( r \) increases until it becomes zero it the limit of infinite distance. The relationship holds even when we consider many bodies, \( i \) and \( j \), interacting with one another.

\[ \sum_{i,j}^N F = \sum_{i,j}^N G \frac{m_i m_j}{r_{ij}^2} \]

Indeed, this is the basis for the well-known n-body problem in astrophysics [40] that can describe the planetary motion of celestial bodies. Trivially, we can see that the computational complexity of an n-body calculation scales \( O(N^2) \), where \( N \) is the number of interacting bodies. A similar inverse square distance relationship is seen in other phenomena, such as Coulomb’s Law for charges[10].

For interactions at the molecular scale, we also see a similar inverse distance-interaction relationship. As we saw in Chapter 3, an example of long-range nonbonded
interactions is the van der Waals interactions that is described by the Lennard-Jones equation:

\[ \sum_{ij}^N E_{vdw} = \varepsilon_{ij} \left[ \left( \frac{R_{\text{min},ij}}{r_{ij}} \right)^{12} - 2 \left( \frac{R_{\text{min},ij}}{r_{ij}} \right)^{6} \right] \]

Again, we note that in the limit of infinite distance the interactions between the pair of interacting beads goes to zero, and the calculation of the interactions scales \( O(N^2) \), where \( N \) is the number of interacting bodies.

We can halve the computations of the interactions by noting that each interaction between \( i \) and \( j \) is identical to the interaction between \( j \) and \( i \), but this remains a computationally demanding problem. A common approach to minimize computations while retaining accuracy is to introduce a cutoff method. Since the interactions between distant beads results in negligible interactions, one can partition the set of possible interactions and only compute those that are proximal to one another. In principle, these “zero” forces could all be calculated to maintain perfect accuracy, but they could also be disregarded without significantly impacting the simulation.

In this chapter, I will review a Verlet Neighbor List algorithm and a GPU-optimized parallel Verlet Neighbor list algorithm, as well as its implementation in our MD simulation code. I will the turn our focus on the Cell List algorithm and a GPU-optimized parallel Cell and Hybrid Neighbor/Cell List algorithms.

4.1 Verlet Neighbor List Algorithm

In the Verlet Neighbor List Algorithm, instead of calculating the force between every possible pair of long-range interactions, a subset “neighbor list” is constructed with particles within a skin layer radius of \( r_l \). The skin layer is updated every \( N \) timesteps, and the interactions within the cutoff radius \( r_c \) are computed at each timestep. These
Figure 4.1: Illustration of long-range interaction cutoff methods. tRNA$^{Phe}$ is shown with blue lines representing all nonbonded interactions (van der Waals). Nonbonded interactions are calculated between all bead pairs (left) by comparing the distance between two beads $i$ and $j$ ($r_{ij}$) and comparing that to a cutoff distance ($r_c$). Bead pairs with distance $r_{ij} < r_c$ form a subset of the all pairs list (right) and are computed accordingly. Calculating all pairs has a complexity on the order of $O(N^2)$, but by using the neighbor list cutoff the complexity is reduced to $O(Nr_c^3) \approx O(N)$.

Interactions become members of the “pair list”, which holds a subset of all interactions. The values of $r_c$ and $r_l$ are chosen as $2.5\sigma$ and $3.2\sigma$, respectively as was done in Verlet’s seminal paper[41]. With the Neighbor List algorithm, the computations of the interactions become $O(Nr_c^3) \approx O(N)$, which becomes far more computationally tractable[30, 41].

Figure 4.2 shows a serial pseudocode that computes the distances between all bead pairs for a Neighbor List algorithm. The outer loop goes over each bead while the inner loop goes over all other beads and computes the distance between the two. The Euclidean distance is calculated using the differences in the x, y, and z directions and is compared to a distance cutoff.

In our simulation code, an interaction list is used. The interaction list contains numbered interactions along with two interacting particles $i$ and $j$. Instead of looping over each bead, a loop would iterate over each interaction and compute the distance between beads $i$ and $j$ and label the interaction as true or false depending on whether $r_{ij} < r_{cut}$. The interaction list also eliminates the need for either for-loop on the GPU.
for \( i = 1 \rightarrow N \) do
for \( j = 1 \rightarrow N \) do
  if \( i == j \) then
    \text{cycle}
  end if
  \( dx \leftarrow x(j) - x(i); \)
  \( dy \leftarrow y(j) - y(i); \)
  \( dz \leftarrow z(j) - z(i); \)
  \( rsq \leftarrow dx \ast dx + dy \ast dy + dz \ast dz; \)
  \( r \leftarrow \sqrt{rsq}; \)
  if \( r \leq r_{\text{cut}} \) then
    \( \text{isNeighborList}[i][j] = \text{True}; \)
  else
    \( \text{isNeighborList}[i][j] = \text{False}; \)
  end if
end for
end for

Figure 4.2: Pseudocode for a CPU-based Neighbor List algorithm

Instead a kernel with \( N \) threads is needed where \( N \) is the number of interactions in the list. A single thread computes and compares the distance for a single interaction between two beads.

### 4.2 GPU-Optimized Parallel Neighbor List Algorithm

To fully parallelize the algorithm to run on a GPU, Lipscomb et al. developed a novel algorithm that utilizes the key-value sort and parallel scan functionalities of the CUDPP Library[tyson]. The new algorithm generates a Neighbor List that is identical to the one generated by the serial Neighbor List while using only parallel operations on the GPU and, in turn, avoids the memory transfer bottleneck of sending information to and from the CPU and GPU.

The CPU version of the Verlet Neighbor List algorithm works by serially iterating through an array of all interactions between two beads \( i \) and \( j \) known as the Master List. When a pair of beads are within a distance cutoff, that entry in the Master List is copied to the Neighbor List[30].

In the GPU optimized version of the algorithm, a second array of equal length to
the Master List is created called the Member List. When the algorithm determines two beads are within a distance cutoff, represented by an entry in the Master List, the corresponding entry in the Member List is assigned a value of “true”. Alternatively, if the two beads are not within the cutoff distance, the corresponding entry in the Member List is assigned a value of “false”[30].

The first step of this parallel method is to perform a key-value sort on the data. A key-value sort involves two arrays of equal length, a keys array and a values array. Typically the keys are sorted in ascending or descending order with the corresponding entries in the values array moving with the sorted entries in the keys array. For example, if the 5th entry of the keys array is sorted to be the new 1st entry, the 5th entry of the values array will become the 1st entry in the values array. In the context of this algorithm, the Member List is used as the key array and the Master List is used as the value array[30].

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. When using the Member List as keys in the key-sort, the “true” values move to the top of the list and the “false” values move to the bottom. The entries in the Master List move along with their counterparts in the Member List which results in the interactions belonging to the Neighbor List residing at the top of the Master List, as shown in Figure 4.3[30].

In order for the entries to be copied from the Master List to the Neighbor List, the number of “true” entries needs to be known. A parallel scan, also known as a parallel reduction, sums the elements of an entire array on the GPU in parallel. Using the parallel scan on the Member List counts the number of “true” values and indicates how many entries of the Master List are within the cutoff distance (Figure 4.3). Once the arrays have been sorted and scanned, that number is used to copy the first x entries of the Master List to the Neighbor List[30].
4.3 Cell List Algorithm

Unlike the Neighbor List algorithm, the Cell List algorithm does not compute the distance between interacting pairs. Rather, the Cell List algorithm works by dividing the simulation box into many rectangular subdomains or cells as shown in Figure 4.4. The particles are sorted into these “cells” based on their x, y, and z coordinates and as the simulation progresses, particles float from one cell to another. On a CPU, the Cell List algorithm is typically implemented using linked lists to keep track of which beads belong to specific cells. Each cell has its own linked list and all atoms
Figure 4.4: Schematic of the Cell List algorithm. The simulation box is decomposed into many rectangular cells. When computing long range interactions, only atoms in adjacent or neighboring cells are computed.

in the cell are added as nodes in the list. Beads in the same cell and adjacent cells are considered to be interacting, are marked “True”, and later added to the pair list to be calculated. A single cell has eight neighboring cells to compute in a 2D environment (One above, below, left, right, and one in each diagonal direction) as well as computing atoms in the same cell. In a 3D implementation, there are 26 neighboring cells to check. At every timestep, the cell list algorithm evaluates beads from the all possible interactions. The biggest advantage of the cell list algorithm is not having to compute the Euclidean distance from all atoms in the biomolecule.

$$\text{distance}(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}$$

Computing its neighbors from adjacent cells is less expensive computationally, because the Euclidean distance requires computing the square of the differences in the x, y, and z directions. A cell list algorithm can also account for periodic boundary conditions. Periodic boundary conditions simply say that interactions occurring on the edge of the simulation box wrap around to the other side. In the context of the cell list algorithm, this means cells at the end of the x, y, or z directions are adjacent to cells.
at the beginning of those dimensions.
Chapter 5:  GPU-Optimized Cell List and Hybrid Neighbor/Cell List Algorithms

As we had seen in Chapter 4 with the recasting of the Verlet Neighbor List algorithm for GPUs, the implementation of even well-established MD simulation algorithms on GPUs is very nontrivial and requires rethinking of the problem in the context of the GPU architecture to take advantage of the parallel architecture while limiting the performance bottlenecks from information transfer to and from the GPU device.

In this chapter, I will focus on my implementation of a Cell List algorithm and a related Hybrid Neighbor/Cell List algorithm for GPUs. As described in Chapter 4, the original cell list algorithm is typically implemented by organizing beads belonging to a cell into a linked list structure. While this approach is optimal for a CPU based architecture, the memory overhead associated with the linked list data structure is unlikely to be optimal for GPUs because of the limited memory and the performance cost associated with memory transfers.

5.1 Possible Implementations of the Cell List Algorithm on GPUs

In order to implement the cell list on a GPU, one would have to rethink and rewrite the serial algorithm into an equivalent parallel version that exploits the underlying hardware. In one possible algorithm on the GPU, each cell should be associated with a CUDA thread block and a single thread should be used to compute the forces for one particle. This approach, similar to the linked-list approach, takes advantage of the parallel architecture of the GPUs by distributing the computational load among
Comparing cells in the x direction

\[
x_i \leftarrow x(i)/cellLength;
x_j \leftarrow x(j)/cellLength;
\]

\[
dx \leftarrow (x_i - x_j)\%(numCells - 1);
\]

\[
\text{if } dx \geq -1 \&\& dx \leq 1 \text{ then}
isNeighborList[i][j] = \text{True};
\]

\[
\text{end if}
\]

The same evaluation is done in the y and z directions. If two beads are in the same or neighboring cells in the x, y, and z directions, the two beads are within the distance cutoff.

the different particles that are interacting independently. However, it can leave a large number of threads unused. If a simulation is not saturated with particles, the vast majority of simulations will have empty cells. For each cell that is unoccupied, it leaves an entire thread block idle for the duration of that kernel call. Another potential approach would be to evaluate the forces by examining each bead individually. This would require each thread to locate the cell a bead was in, determine the neighboring cells, and then loop over each bead in each adjacent cell. This method is also less than ideal because it requires each thread to perform several loops and does not distribute the work to all the processors available.

5.2 Parallel Cell List Algorithm

One of the most prominent features of our simulation code is the interaction list. Unlike several other general purpose particle dynamics simulations that treat a particle
Figure 5.2: When applying periodic boundary conditions to the simulation box, beads $i$ and $j$ are calculated as interacting beads.

as a type (i.e. Proteins or RNA) and use the type to calculate a uniform force for all particles of that type, our overall design was to evaluate the individual interactions. In keeping with the SOP model Hamiltonian, some interactions are attractive and others are repulsive. We evaluate cutoffs by looking at each interaction individually. In our implementation of the cell list algorithm, we allocate $N$ threads, one for each interaction in the interaction list. Each thread evaluates whether two beads $i$ and $j$, are in neighboring cells. Instead of calculating a global cell id from the $x$, $y$, and $z$ coordinates of the beads and determining all of the neighbors, we compared the beads in each individual direction.

As a pedagogical illustration of the Cell List algorithm, we refer to Figure 5.1A, where two beads $i$ and $j$ are compared in the $x$ direction. In this example, we chose a cell length of 10 to evenly divide the simulation box into 75 cells in a single direction. To determine the cell id in the $x$ direction, the $x$ coordinate of the bead is integer divided by the cell length as shown in Figure 5.1B. In Figure 5.1A, bead $i$ has an $x$
position of 27 and when divided by the cell length of 10, \( x_i = 2 \) or it exists in \( \text{Cell.x} = 2 \). We performed the same set of operations for bead \( j \) and determine it lies in \( \text{Cell.x} = 3 \). After taking the difference of \( \text{Cell.x} \) for both beads, an “if” statement checks if \( dx \) is within the range \(-1 \leq dx \leq 1\). If the beads were in the same cell, then \( dx \) would equal 0. If \( dx \) equals 1 or -1, the beads are neighboring each other either to the left or right.

By examining a particle’s position in only one dimension, periodic boundary conditions are much more straight forward to implement. To account for periodic boundary conditions, we want beads in the first cell and last cell to be recognized as neighbors. In our simulation code, this is done by computing \( dx \) modulo \( \text{numCells} - 1 \). In Figure 5.2, \( \text{Cell.x} 5 \) and 0 are neighboring cells under the definition of periodic boundary conditions. If two beads are in these cells, \( dx \) will equal 5 or -5. Computing modulo \( \text{numCell} - 1 \) would assign \( dx \) a value of 0 thus allowing beads in the first and last cell to pass through the “if” statement. Beads \( i \) and \( j \) are compared in the \( x, y, \) and \( z \) directions in 3 nested “if” statements. If they are neighboring in all three directions, they are flagged as true and are added to the pair list to have their forces evaluated. This kernel, shown in Appendix A and B, is applied to each entry in the interaction list. A single thread is tasked with executing the kernel code for one pair of beads \( i \) and \( j \) and each thread does so independently and in parallel.

5.3 Hybrid Neighbor/Cell List Algorithm

In the Verlet Neighbor List algorithm, the computations required for calculating the interactions between beads in our simulations are significantly reduced while maintaining high accuracy by instead calculating the distance between them first to determine whether to compute the interaction using a distance cutoff. If the two beads are sufficiently far away, we know that the interaction is effectively zero so the interaction
only negligibly contributes to the overall dynamics and can be ignored. To further improve performance, the Neighbor List algorithm maintains two lists: 1) a Neighbor List with a larger distance cutoff that is updated every N timesteps and 2) a pair list with a smaller distance cutoff with interactions that are a subset of the Neighbor List. As we saw previously, the Neighbor List algorithm was successfully recast for implementation on the GPU using only parallel operations.

In the Cell List algorithm the relatively expensive distance calculations in the Neighbor List algorithm, while still cheaper than calculating the actual interaction, is replaced by the less expensive conditional operations. By dividing up the simulation box into cells, the actual distances do not have to be calculated. As we described in the previous section, this algorithm can also be recast for GPUs using only parallel operations. In addition, by choosing appropriate cell lengths, the accuracy of our computations can be maintained while benefiting from the expected performance improvements.

It has been shown in a previous study[43] that combining the Neighbor and Cell Lists into a hybrid method has been advantageous on a CPU. We then asked whether it would be possible to take these two approaches and combine their benefits into a Hybrid Neighbor/Cell List algorithm for GPUs.
Figure 5.3 shows the progression from the Neighbor and Cell Lists to the Hybrid Neighbor/Cell List algorithm by using the cell list’s cutoff to construct an outer “cell list” that is updated every N timesteps. Once the cell list has been constructed, a radius distance cutoff ($r_c$) is applied to its members and the interactions within $r_c$ are copied to the pair lists. The pair lists are built using CUDPP’s parallel key sort and parallel scan function in the same way we described earlier in section 4.2. At every timestep, the forces are evaluated for the members of the pair list.
Chapter 6: Performance Analyses

Now that we have developed the GPU-optimized algorithms for coarse-grained MD simulations we make comparisons with HOOMD, a well-known and leading general particle dynamics simulation code. We then present execution times comparing the GPU-optimized Neighbor List with the original Hybrid Neighbor/Cell List and the Hybrid Neighbor/Cell List that is currently implemented. We proceed with showing the N-dependent percent speedup of the Hybrid Neighbor/Cell List over its original counter-part and the Neighbor List algorithm. Finally, we compare the N-dependent execution times of the Neighbor List, Cell List, and the Hybrid Neighbor/Cell List.

6.1 Implementation of the SOP model on HOOMD

HOOMD stands for Highly Optimized Object-oriented Many-particle Dynamics. It is a widely used general purpose particle dynamics simulation code that is designed to be executed on a single computer while utilizing GPUs to achieve a high level of performance. HOOMD incorporates a number of different force fields and integrators, including the NVT integrator which we use in our SOP model code, to accommodate the diverse needs of researchers performing MD simulations. Simulations in HOOMD are configured and run using python scripts to specify the choice of force field, integrator, the number of time steps, and much more. “The code is available and open source, so anyone can write a plugin or change the source to add additional functionality” [3]. The availability of the code, its high flexibility, and its standing in the community make it an ideal software package for us to implement the SOP model Hamiltonian for the purpose of performance benchmarking against our own code. As a consequence of its generic nature, HOOMD is able to compute many different types
of potentials such as the harmonic bond and angle and Lennard-Jones potentials used in our SOP model simulations. We implemented the following energy potentials in HOOMD to replicate the SOP model Hamiltonian:

\[
V(\vec{r}) = V_{\text{bond}} + V_{\text{angle}} + V_{\text{VDW}}^{\text{ATT}} + V_{\text{VDW}}^{\text{REP}}
\]

\[
= \sum_{i=1}^{N-1} \frac{k_r}{2} \left( r_{i,i+1} - r_{i,i+1}^0 \right)^2
\]

\[
+ \sum_{i=1}^{N-2} \frac{k_\theta}{2} \left( \theta_{i,i+1,i+2} - \theta_{i,i+1,i+2}^0 \right)^2
\]

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

\[
+ \sum_{i=1}^{N-3} \sum_{j=i+3}^{N} \epsilon_l \left( \frac{\sigma}{r_{i,j}} \right)^6 (1 - \Delta_{i,j})
\]

The first and second terms are the short-range harmonic bond and angle potentials. The parameters are: \( k = 20 \text{ kcal/mol} \) and \( r_{i,i+1}^0 \) and \( \theta_{i,i+1,i+2}^0 \) is the distance and angle between neighboring beads in the folded structure, and \( r_{i,i+1} \) and \( \theta_{i,i+1,i+2} \) are the actual distance and angle between neighboring beads at a given time \( t \), respectively.

While these terms are explicitly different, they serve the same purpose as the original SOP model we implemented in our MD simulation code. Furthermore, we expect qualitatively identical simulation results[37].

The third and fourth terms are the Lennard-Jones interactions that are identically implemented. As such, the performance scales \( O(N) \). The HOOMD code is designed to accommodate the Lennard-Jones interactions of a homogeneous and heterogeneous particle systems of different types of particles with different \( r_{i,j}^0 \). We note, however, that the implementation of the Lennard-Jones interactions requires a different type
for each interaction pair because the $r_{i,j}^0$ is different for each native interaction pair, and the memory storage scales as $O(N^2)$. The HOOMD code was not designed to handle this many different "types" of interactions, and the current release version of the code limits the number of types so that the parameters fit in shared memory. We therefore explicitly removed the limit so that the GPU hardware memory will instead determine the limit on the number of types[37]. To run the simulations in HOOMD, we used 2 input files: 1) a python script(Appendix D) to set the size of the simulation box, the correct Neighbor List parameters, and the aforementioned forces described above and 2) an xml file(Appendix E) used to load the exact same biomolecule we simulate inside our SOP model.

6.2 Performance Comparison with HOOMD

We implemented the SOP model into HOOMD and compared our simulation code to that of HOOMD. We ensured that the average number of particles being evaluated by the Neighbor List were roughly same in both our MD simulation code and in HOOMD. This measure was chosen largely because HOOMD already carried out this calculation, however, we do not expect significant differences in our conclusions using
different measures. The original HOOMD code has a limit on the number of different types of interactions, we trivially modified the simulation code to remove this limit. There also exists a hardware limit in the amount of memory available to the GPU. In the case of the NVIDIA C2070, which we used for our simulations, that limit is 6 GB of memory.

For systems less than ~400 beads, HOOMD has a faster execution time than our MD simulation code (Figure 6.1A). However, for larger systems up to ~1,000 beads, our simulation code execution time is markedly less. After that point, the HOOMD code can no longer execute the simulations due to memory limits. However, our simulation code is able to handle larger systems because of the type compression and other optimization techniques we used to reduce memory transfer bottlenecks, and we again observe N-dependent execution times up to ~10,000 beads (Figure 6.1B).
6.3 Hybrid Neighbor/Cell List Performance Improvements

Initially, the hybrid list algorithm had a slower execution time than the neighbor list (Figure 6.2). However, adding an additional CUDA kernel call and storing a few additional pieces of information significantly reduced its execution time, listed in Appendix C. When the hybrid list is comparing the location of two beads i and j, it must compute which cell they reside in for each dimension. Long range interactions exist between every particle in the system therefore, each bead is involved in \( n - 1 \) interactions where \( n \) is the number of beads. There are \( n - 1 \) interactions because beads do not interact with themselves. This means each bead’s cell location is computed \( n - 1 \) times. As the system size increases, the cell location is needlessly computed more and more. To avoid this inefficiency, an additional CUDA kernel with \( n \) threads is used to compute and store the cell number for each bead. An array of size \( n_{\text{bead}} \) stores the Cell.x, Cell.y, and Cell.z in the x, y, and z components of a float3 data type. When the interactions are being computed the cell positions are accessed from memory rather than being calculated again.
Figure 6.4 shows the size dependent speedup of the optimized hybrid list over its original version and over the previously implemented neighbor list. The percent speedup over the neighbor list is on average \( \sim 10\% \) faster with the exception of our smallest system, the 1ehz(76 beads), which runs faster on the CPU. The 1ehz is so small that there are not enough beads to fully utilize the GPUs and therefore no speedup is to be expected. It is worth noting the \( \sim 10\% \) speedup over the neighbor list is additional to the \( \sim 30x \) speedup of the neighbor list over the CPU version of the code for the 70s ribosome (10,219 beads).

6.4 Comparisons of MD Simulations Execution Times with GPU-Optimized Cutoff Method Algorithms

We ran timings for each cutoff method in our simulation code on the NVIDIA GTX480. Figure 6.4 shows the \( n \)-dependent execution times while a list of the timings can be observed in Table 6.1. The cell list algorithm is much slower than both the neighbor and hybrid lists algorithms. The slower execution time is to be expected however, due to the fact the cell list must compute interactions from the all pairs list at each and every timestep. The neighbor and hybrid lists algorithms only have
<table>
<thead>
<tr>
<th>System</th>
<th>Neighbor List</th>
<th>Cell List</th>
<th>Hybrid Original List</th>
<th>Hybrid List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1ehz (76 beads)</td>
<td>27.22</td>
<td>25.5</td>
<td>27.09</td>
<td>27.35</td>
</tr>
<tr>
<td>16s (1,530 beads)</td>
<td>42.31</td>
<td>135.46</td>
<td>37.59</td>
<td>36.69</td>
</tr>
<tr>
<td>30s (3,883 beads)</td>
<td>80.64</td>
<td>735.92</td>
<td>82.40</td>
<td>76.40</td>
</tr>
<tr>
<td>50s (6,336 beads)</td>
<td>145.54</td>
<td>1787.23</td>
<td>147.83</td>
<td>132.18</td>
</tr>
<tr>
<td>70s (10,219 beads)</td>
<td>277.35</td>
<td>4766.35</td>
<td>296.00</td>
<td>253.74</td>
</tr>
</tbody>
</table>

Table 6.1: Execution times(s) simulated on the NVIDIA GTX480

to do this every N timesteps and the pair list for each of those methods only has to evaluate a small subset of the all possible interaction pairs.
Chapter 7: Conclusions

Over the course of time, computers and their capabilities have expanded to the point where they are becoming increasingly ubiquitous in our society because they can help us solve so many different problems. Specialized hardware such as GPUs arose from the need to render realistic images, but also showed great potential for scientific applications as parallel processors. On such application is computational biophysics, where researchers use the parallel computations of a GPU to calculate the physical interactions of atoms in a biomolecule to study its movement over time. This class of simulations, known as Molecular Dynamics, uses cutoffs as a truncation scheme for particles that are not in close proximity to significantly interact in order to extend the life of the simulation. Two popular cutoff algorithms, the neighbor and cell list, were originally designed to execute serially on the CPU and required rethinking and redesigning of the underlying algorithms so they could be executed in parallel and take full advantage of the GPU.

The implementation of a parallel cell list algorithm that updates at every timestep turned out to be significantly slower than its parallel counterpart, the Verlet Neighbor list. However, using the cell list cutoff with a neighbor list and a pair list in a new hybrid neighbor/cell list algorithm yielded a $\sim 10\%$ speedup over our previously optimized MD code. The hybrid list provides an alternative approximation as to which particles should be included in the calculation of Leonard-Jones forces while possessing a faster execution time. To benchmark the parallel neighbor list, we implemented the SOP model into HOOMD and compared the two using various system sizes. We found that HOOMD was faster for the smallest of systems we tested, but the largest system it could simulate was $\sim 1,000$ beads due to the memory constraints.
of the GPU. On the other hand, the SOP model is capable of simulating systems of more than $\sim 10,000$ beads.

In the last decade, GPUs have witnessed an astonishing growth in performance. This trend shows no signs of slowing and looks to continue well into the future. This will lead to landmark discoveries and the understanding of bigger, more biologically relevant systems. One such breakthrough is studying the human genome. With over 3 billion letters in a DNA sequence, the high throughput of GPUs will be instrumental in solving this problem. This problem alone could yield valuable insight to which genes are responsible for expressing particular phenotypes and which sequences result in disease or cancer. The genome is just one of many areas where significant strides are being made in understanding challenging problems. Forecasting weather, predicting earthquakes and the stock market, and developing medicines are all fields of study that have been augmented by the advancement of GPU technology.

While there are still many interesting problems waiting to be solved, it will all be driven by the improvements in future generations of hardware. One such advancement will be the integration of the CPU and GPU on a single die. With NVIDIA announcing that the next generation GPU “Maxwell” will take advantage of Unified Virtual Memory, which allows the CPU and GPU to see all of the system’s memory, it is only a matter of time before a CPU/GPU hybrid chip becomes a reality. The integration of both the CPU and GPU on the same chip would remove the memory transfer bottleneck and allow for serial and parallel parts of applications to run simultaneously without penalty. Advancements in hardware such as the combined processing unit may open the door to new understandings in fields of study utilizing the power of computing.
Bibliography


Appendix A: Cell List Kernel: Attractive List

```c
__global__ void update_cell_list_att_kernel(unsigned int *dev_is_cell_list_att, int ncon_att, PDB_FLOAT *dev_lj_nat_pdb_dist, ushort2 *dev_idx_bead_lj_nat, FLOAT3 *dev_cell_list, int ncell)
{
    ushort2 idx_bead_lj_nat;
    unsigned int ibead, jbead;
    int flag = 1;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    int dx, dy, dz;
    int x1, x2, y1, y2, z1, z2;
    if (i < ncon_att)
    {
        idx_bead_lj_nat = dev_idx_bead_lj_nat[i];
        //Get indices i and j for a particular interaction
        ibead = GET_IDX(idx_bead_lj_nat.x) - 1;
        jbead = GET_IDX(idx_bead_lj_nat.y) - 1;

        //Retrieve cell ids for bead i
        FLOAT3 ipos = dev_cell_list[ibead];
        x1 = ipos.x;
        y1 = ipos.y;
        z1 = ipos.z;

        //Retrieve cell ids for bead j
        FLOAT3 jpos = dev_cell_list[jbead];
        x2 = jpos.x;
        y2 = jpos.y;
        z2 = jpos.z;

        //Compute the differences between the cell
        //ids and apply periodic boundary conditions
        dx = (x1 - x2) % ncell;
        dy = (y1 - y2) % ncell;
        dz = (z1 - z2) % ncell;
    }
}
```
// Check if beads are in neighboring cells in each direction
if (dx <= 1 && dx >= -1)
{
    if (dy <= 1 && dy >= -1)
    {
        if (dz <= 1 && dz >= -1)
        {
            // Beads i and j are neighbors
            // Change the value of flag
            flag = 0;
        }
    }
}
// end if (dx)

// end if (dy)

// end if (dx)

// flag holds value of 0 or 1
// NOTE: 0 corresponds to true and 1 corresponds to false
dev_is_cell_list_att [i] = flag;

// end if (i)

} // end update_cell_list_att/kernel
Appendix B: Cell List Kernel: Repulsive List

```c
__global__ void update_cell_list_rep_kernel(unsigned int *dev_is_cell_list_rep, int xsize, int ysize, int ncon_rep, ushort2 *dev_idx_bead_lj_non_nat, FLOAT3 *dev_cell_list, int ncell)
{
    int dx, dy, dz;
    int x1, x2, y1, y2, z1, z2;
    int flag = 1;
    ushort2 idx_bead_lj_non_nat;
    unsigned int ibead, jbead;

    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int j = blockIdx.y * blockDim.y + threadIdx.y;

    //TODO: Clean the nested if’s up
    if (i <= xsize && j <= ysize)
    {
        unsigned int idx = j * xsize + i;
        if (idx < ncon_rep)
        {
            idx_bead_lj_non_nat = dev_idx_bead_lj_non_nat[idx];

            //Get indices i and j for a particular interaction
            ibead = GETIDX(idx_bead_lj_non_nat.x) - 1;
            jbead = GETIDX(idx_bead_lj_non_nat.y) - 1;

            //Retrieve cell ids for bead i
            FLOAT3 ipos = dev_cell_list[ibead];
            x1 = ipos.x;
            y1 = ipos.y;
            z1 = ipos.z;

            //Retrieve cell ids for bead j
            FLOAT3 jpos = dev_cell_list[jbead];
            x2 = jpos.x;
            y2 = jpos.y;
            z2 = jpos.z;

            //Compute the differences between the cell
```
// ids and apply periodic boundary conditions
dx = (x1 - x2) % ncell;
dy = (y1 - y2) % ncell;
dz = (z1 - z2) % ncell;

// Check if beads are in neighboring cells in each direction
if (dx <= 1 && dx >= -1)
{
    if (dy <= 1 && dy >= -1)
    {
        if (dz <= 1 && dz >= -1)
        {
            // Beads i and j are neighbors
            // Change the value of flag
            flag = 0;
        }
    }
}

// flag holds value of 0 or 1
// NOTE: 0 corresponds to true and 1 corresponds to false
dev_is_cell_list_rep[idx] = flag;

#endif

// end update_cell_list_rep_kernel
Appendix C: CUDA Kernel: Compute Cell IDs

// This CUDA Kernel was used to speed up the original Hybrid list

// Computes the cell id based on the x, y, and z coordinates

// Instead of calculating the cell id each time in the above kernels,
// the code below calculates it once for each bead and stores
// the cell id in dev_cell_list[]

// For example, dev_cell_list[1].x would hold the cell id in the x
dimension for bead #1

__global__ void compute_cell(FLOAT3 *dev_cell_list, FLOAT3 *dev_unc_pos, int nbead, FLOAT offset, FLOAT lcell)
{

// Calculate the current thread’s global index
// based on the grid dim and block dim
int t_per_b = blockDim.x * blockDim.y;
int b_index = blockIdx.x + blockDim.x * blockIdx.y;
int t_index = threadIdx.x + blockDim.x * threadIdx.y;
int global_index = t_per_b * b_index + t_index;

// Find the x, y, and z coordinates of the cell each bead belongs to.
if (global_index < nbead)
{
    dev_cell_list[global_index].x = (dev_unc_pos[global_index].x + offset)/lcell; // divide by cell_length
    dev_cell_list[global_index].y = (dev_unc_pos[global_index].y + offset)/lcell; // +375 to make all coords > 0
    dev_cell_list[global_index].z = (dev_unc_pos[global_index].z + offset)/lcell; // + offset to make all coords > 0
}
}
from hoomd_script import *
import math
import time

# Read particle positions, diameter, bonds, and angles
# from xml file to represent 1ehz biomolecule
init.read_xml(filename="1ehz_init.xml")

# Set harmonic bond forces for physically bonded particles
harmonic2 = bond.harmonic()
harmonic2.set_coeff('0', k=20.0, r0=4.685835)
harmonic2.set_coeff('1', k=20.0, r0=5.248249)
harmonic2.set_coeff('2', k=20.0, r0=5.015971)
harmonic2.set_coeff('74', k=20.0, r0=5.015971)

# Set harmonic angle forces for physically bonded particles
harmonic1 = angle.harmonic()
harmonic1.set_coeff('0', k=20.0, t0=0.79463310555556)
harmonic1.set_coeff('1', k=20.0, t0=0.8530889)
harmonic1.set_coeff('2', k=20.0, t0=0.83845157222222)
harmonic1.set_coeff('73', k=20.0, t0=0.73625928333333)

# Set Leonard–Jones forces for all interactions
lj = pair.lj(r_cut=3.0)
lj.pair_coeff.set('1', '1', epsilon=0.0, sigma=1.0, r_cut=1.0)
lj.pair_coeff.set('1', '2', epsilon=0.0, sigma=1.0, r_cut=1.0)
lj.pair_coeff.set('1', '3', epsilon=0.0, sigma=1.0, r_cut=1.0)
lj.pair_coeff.set('1', 'N', epsilon=1.0, sigma=12.2, alpha=1, r_cut=14)
lj.pair_coeff.set('2', '2', epsilon=1.0, sigma=12.2, alpha=1, r_cut=14)
lj.pair_coeff.set('2', '3', epsilon=1.0, sigma=12.2, alpha=1, r_cut=14)
lj.pair_coeff.set('76', '76', epsilon=0.0, sigma=1.0, r_cut=1.0)
dump.dcd(filename='lehz_sim.dcd', period=1000)
all = group.all()
integrate.mode_standard(dt=0.005)
integrate.nvt(group=all, T=0.00002, tau=0.5)

#Set Neighbor list parameters
nlist.set_params(r_buff=0.8, check_period=5)

#Get time before and after run command to measure
#the true computational time
#Don't include the time taken to set forces

start = time.time()
run(10000)
end = time.time()

print 'HOOMD took %0.3f s' % ((end-start))
Appendix E: HOOMD .xml File

<!−−xml file used in HOOMD input script for the 1ehz biomolecule−−>

<?xml version="1.0" encoding="UTF-8"?>
<hoomd_xml>
<configuration time_step="0">
<box Lx="750" Ly="750" Lz="750"/>
<position>
  1: -15.439708 -2.541125 24.874000
  2: -10.836100 -2.404900 25.737300
  3: -6.168826 -3.876870 23.841522
  ...
  76: -31.583545 -12.430227 31.556591
</position>
?type>
  1
  2
  ...
  76
</type>
<diameter>
  7.0
  7.0
  ...
  7.0
</diameter>

<!−−Left column are the bond indices−−>
<!−−Middle and right columns are the bonded particles−−>
<bond>
  0 0 1
  1 1 2
  2 2 3
  3 3 4
  ...
</bond>
<!----Left column are the angle indices---->
<!----Two Middle and right columns are---->
<!----the particles that make up the angle---->

  <angle>
  0 0 1 2
  1 1 2 3
  2 2 3 4
  3 3 4 5

  71 71 72 73
  72 72 73 74
  73 73 74 75

  </angle>

  </configuration>

  </hoomd_xml>
Andrew J. Proctor
4727 Burton Road, Thomasville, NC 27360
(336) 687-3176, Procta06@gmail.com

Education

Wake Forest University, expected May 2013  
MS, Computer Science

High Point University, NC, May 2011  
BS, Computer Science
BA, Mathematics
Minor Physics

Experience

Research Assistant, Wake Forest University  1/12 - Present
• High Performance Computational Biophysics
• GPU-Optimized Molecular Dynamics Simulations
• Developed and implemented parallel Cell and Hybrid Neighbor/Cell List algorithms

Lab Assistant, High Point University  1/08 - 5/11
• Supervised Department of Computer Science’s Unix Lab
• Tutored Computer Science
• Assisted professors with classroom technology

Research Assistant, The University of Auburn  5/10 - 7/10
• NSF Research Experiences for Undergraduates (REU)
• Worked with professors flying multiple, autonomous Unmanned Aerial Vehicles (UAVs)
• Implemented a collision avoidance algorithm for the UAVs

Publications


Poster Presentations

1. GPU Technology Conference (San Jose, CA). “Performance Analyses of a Parallel Neighbor List Algorithm for GPU-Optimized MD Simulations” (Competitive).

2. Center for Molecular Communication and Signaling Retreat (Winston Salem, NC). “Performance Analyses of a Parallel Verlet Neighbor List Algorithm for GPU-Optimized MD Simulations”

Academic Honors

Alumni Travel Award Fund  10/12 WFU
Presidential Fellows Scholarship  08/07 - 05/11 HPU
Dean’s List Fall ’07 and ’08, Spring ’07 HPU

Activities

Broader Engagement Workshop @ Super Computing ’12 Conference  11/12 WFU
Georgia Tech Network Security Contest Competitor  8/12 WFU
Student Volunteer for Wake @ Hanes Google CS4HS Teacher Workshop  8/12 WFU
• FOX8 News: “Buckley Report: Computers in middle school” (Interviewed)
  
- Wake Forest University News Center: “Computers in the middle” (Interviewed)
  http://news.wfu.edu/2012/09/04/computers-in-the-middle

Presenter @ Big South Undergraduate Research Symposium 4/11 HPU
ACM International Collegiate Programming Contest Competitor 11/(08,09,10) HPU
NC A&T University Programming Contest Competitor 04/(08,09,10) HPU
Mathematics, Actuary & Computer Science (MACS) Member HPU

<table>
<thead>
<tr>
<th>Computer Skills</th>
<th>Languages:</th>
<th>C, C++, CUDA, HTML, Javascript, PHP, Perl, VPython, MySQL, LATEX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Software:</td>
<td>Matlab, Easy Java Simulations, Lab View, NetLogo</td>
</tr>
<tr>
<td></td>
<td>Systems:</td>
<td>Linux, OS X, Windows</td>
</tr>
</tbody>
</table>