

DYNAMIC PARALLELISM IN GPU OPTIMIZED BARNES HUT TREES FOR
MOLECULAR DYNAMICS SIMULATIONS

BY

MELISA CARRANZA ZÚÑIGA

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 2017

Winston-Salem, North Carolina

Approved By:

Samuel S. Cho, Ph.D., Advisor

David John, Ph.D., Chair

William Turkett, Ph.D.

Acknowledgments

Thanks to Sam, for being a splendid guide in my learning process. He makes science and teaching the highest of goals, my motivation in every step of the process is just a natural consequence of his pursuit.

I'm very grateful to my committee members, Dr. John and Turkett. Thanks for your time and help on this project and in my two years at Wake Forest University. You are both great instructors and researchers and working with you has been my pleasure. Thank you to Dr. Daniel Cañas for being a friend and a mentor and, above all, the person who has set me and thousands of other computer scientists in Costa Rica for success.

Thanks to my parents and sister, I am because of you. Thank you for celebrating every victory with me. To the rest of my family, my friends, and my extended American family, you've been rooting for me all the way and it has paid off.

Finally, to all the faculty and staff at Wake Forest University, your commitment to teaching and perpetuating academic work is truly remarkable. You have made my life significantly easier, thank you.

Table of Contents

Acknowledgments	ii
List of Figures	v
List of Tables	vii
Abstract	viii
Chapter 1 Introduction	1
Chapter 2 Overview of Supercomputing	5
2.1 History of supercomputing	6
2.2 Performance over time	9
Chapter 3 GPU Technology	14
3.1 Parallel computing	14
3.2 GPGPU	17
3.3 CUDA framework	19
3.3.1 Dynamic Parallelism	22
3.4 Further considerations	25
Chapter 4 Molecular Dynamics	27
4.1 Biomolecular Folding	28
4.2 Molecular Dynamics Algorithm	28
4.3 MD Simulations Granularity	31
4.4 Coarse-Grained Simulation Models	32
4.5 Molecular Dynamics with GPUs in Other Models	34
Chapter 5 Cutoff Methods	35
5.1 Motivation for Cutoff Methods	35
5.2 Verlet Neighbor List Algorithm	36
5.3 GPU Optimized Parallel Neighbor List	38
5.4 Cell List Algorithm	40
5.5 Hybrid Neighbor/Cell List Algorithm	41

Chapter 6	Barnes-Hut Algorithm	42
6.1	Parallel Barnes-Hut Tree	45
6.2	Dynamically Parallel Barnes Hut	46
Chapter 7	Performance of Barnes-Hut Algorithm in MD Simulations	53
7.1	CPU implementation	53
7.2	GPU implementation	58
Chapter 8	Conclusions and future work	62
Bibliography		64
Appendix A	CPU Barnes Hut Tree Construction	70
Appendix B	GPU Barnes Hut Tree Construction	75
Curriculum Vitae		83

List of Figures

2.1	Eniac	6
2.2	ASCI Red	8
2.3	Moore's Law decline	9
2.4	Nvidia GTX 1080	10
2.5	GPUs and CPUs	11
2.6	Nvidia: flops over time	12
2.7	Nvidia: Memory bandwidth	13
3.1	GPU vs CPU architecture	18
3.2	Heterogeneous programming	20
3.3	Grid and thread blocks	21
3.4	Grid and thread blocks	22
3.5	CUDA Memory hierarchy	24
3.6	Fluid simulation	25
3.7	Nested kernel launches	26
5.1	Verlet Neighbor List	37
5.2	Parallel Neighbor List	39
5.3	Cell list algorithms	40
6.1	Barnes Hut tree	44
6.2	Lightweight locking	46
6.3	Work performed by each block in the parallel BH	47
6.4	Barnes Hut tree construction at depth 0	50
6.5	Barnes Hut tree construction at depth 1	51
6.6	Barnes Hut tree construction at depth 2	51
6.7	Barnes Hut tree construction at depth 3	52
6.8	Final construction of the Barnes Hut tree	52
7.1	Data input pair list	54
7.2	Approximated distances and bead to bead distance calculations	56
7.3	Theta influence on computation rate	57
7.4	Cutoff method recalculated every N steps and BH tree adaptive recal- culation	58

7.5	Seconds per timestep, varying theta	59
7.6	Van der Waals energy relation for simulations with different theta values	60
7.7	Pairs distributed in thread blocks	61

List of Tables

3.1	Cuda memory types	23
7.1	CPU simulation time breakup	57

Abstract

Melisa Carranza Zúñiga

Since the beginning of the modern computing era, high performance computing has been pushing the boundaries of the types of problems that can be solved in many different disciplines. One of the leading fields is computational biophysics where molecular dynamics (MD) simulations provide microscopic resolution details of how biomolecules move, fold, and assemble into intricate complexes that perform biological functions. However, it still remains a challenge to accurately perform MD simulations of biologically relevant complexes at timescales that can be directly compared with experiments. While the fundamental features of biomolecular dynamics, folding, and assembly are very interesting, their misfolding or misassembly can lead to deleterious repercussions that lead to diseases such as Parkinson's and Alzheimer's. MD simulations have played key roles in successes so far in directing experiments that lead to therapies, but advances in high performance computing hardware and algorithms will expand the scope of the problems that can be solved.

For the past decade, graphics processing units (GPUs), which were originally designed for rendering images, have been successfully repurposed for scientific computing largely due to its parallel architecture that allow many cores to quickly implement parallel algorithms. Continuous advances in the capabilities of GPUs have introduced novel parallel algorithms that are specifically optimized for the GPU architecture. One of the key new capabilities of GPUs is dynamic parallelism that allows a natural implementation of recursive algorithms, which have been difficult to implement except through intricate workarounds. In this thesis is presented the first application of the Barnes-Hut algorithm for Self-Organized Polymer (SOP) Model MD simulations on the GPU. To our knowledge, Barnes-Hut algorithm, which has been used extensively in other types of simulations in other fields, most notably astrophysics, has not been implemented for MD simulations of biological systems.

Chapter 1: Introduction

What constitutes a computer has changed a lot. A simple abacus consisted of beads and strings to function as a calculator. More sophisticated mechanical calculators were invented by mathematicians such as Blaise Pascal and Gottfried Leibniz. More automatic machines that were programmable with algorithms were later introduced by Charles Babbage and Ada Lovelace [1, 2]. Since then and for a very long time afterwards, these sophisticated calculating machines, or computers, that started off advancing primarily mathematics and physical sciences has now become widespread to many other areas including economics, social studies, and medicine, to name a few. Indeed, modern computers have now impacted almost every area of society, and they continue to evolve in terms of the hardware and software.

Computing capacity especially skyrocketed during the past 50 years, moving from the order of the thousands to the billions of transistors in consumer computing products [3]. This rate of hardware improvement has grown even faster in recent years, largely due to the introduction and continued development of GPUs, which can now perform operations in the order of teraflops [4]. One of the greatest advantages of GPUs is that they are specialized for parallel calculation tasks with a significant lower cost than more general computing architectures such as a traditional CPU. Furthermore, they offer the computing power for performing scientific computing that was previously only available on large supercomputers in a personal desktop. While the magnitude of a GPU in a desktop cannot match a supercomputing facility, it is still sufficient for sufficiently high performance computing, and these increasingly affordable devices are now used by researchers to tackle significant scientific questions in an affordable system.

A well-known application in high performance computing is the N -body problem. For a system with a set of bodies, each of those bodies interact with each other to determine where they will have new positions [5]. Variants of these N -body simulations are widely used in astrophysics, fluid mechanics, weather forecasting, biophysics, and many other fields where there still exist significant dynamic systems of interacting particles. In biophysics, molecular dynamic (MD) simulations model the interactions of biomolecular systems, including proteins, RNA, and DNA molecules. These biomolecular systems consists of atoms that are grouped together into linear chains of amino acids and nucleic acids. Further, they interact with one another through harmonic chemical bonds and long-range electrostatic interactions. Through studying the motions of these biomolecules and their evolved ability to fold into specific structures and bind to partners, one can study how these biomolecules carry out their biological functions at a molecular resolution [6]. The arguably greatest value of MD simulations, however, is the ability of these simulations to complement or even direct *in vitro* and perhaps even *in vivo* experiments. Such MD simulations can provide new insight that cannot be accessed through low resolution experiments. Also, MD simulations can narrow the scope of expensive and time consuming experiments by performing virtual screening of a large number of candidates to instead focus on a small set of the most promising candidates, as is the case with virtual drug design approaches.

For MD simulations to be performed at a level that they can be directly compared with experiments, however, there exist computational challenges. Specifically, all N -body simulations, including MD simulations, involve interactions of many particles that can each potentially interact with all other particles in the system. As such, the number of interactions that need to be computed for each timestep of the simulation grows as $\mathcal{O}(n^2)$, where n is the number of particles. The typical size of a biologically

relevant biomolecule ranges from about 100 to 10,000 residues and nucleotides, and could be much larger. Furthermore, after these interactions are calculated, their positions and velocities are updated, and the process is repeated iteratively until a trajectory results. For these trajectories to be meaningfully compared to experiments, they must be long enough to have timescales that are comparable to experiments, which remains a significant computational challenge for even the smallest biomolecules.

Therefore, scientists and researchers have developed new algorithms and approaches for reducing the computational burden of these simulations while retaining the accuracy of their simulations. Some well-known algorithms in MD simulations include the Verlet Neighbor List, Cell List, and hybrid approaches [8, 9, 51]. The main focus of these algorithms is to reduce the number of interactions one must compute by only calculating the most significant ones.

This document is structured in the following manner. In Chapter 2 a very brief review of supercomputing history and its importance is given for context. Chapter 3 is concerned with parallelism concepts and how they are used to measure performance. GPU technology is introduced, in particular, as a means of achieving affordable supercomputing via data parallelism for problems that require a high amount of float point calculations. Chapter 4 introduces the concept of Molecular Dynamics, their importance, and how they can be simulated as a form of the N -Body problem. It also formally describes the Self-Organized Polymer (SOP) model that we apply for the our studies presented here, primarily in Chapter 7. Chapter 5 summarizes previous methods utilized to decrease computation times in the N -Body problem and parallel implementations of MD. One type of these methods are cutoff algorithms make approximations to reduce the amount of computations removing negligible interactions from computations. Chapter 6 describes the Barnes-Hut algorithm, a cutoff method widely used in astrophysics N -body simulations, and we outline its potential for ap-

plying to MD simulations. In particular it describes parallel implementations of the Barnes-Hut tree to reduce the amount of computations required. Chapter 7 shows the results of the implementation of the parallel and sequential Barnes-Hut tree in the SOP model MD simulation code. Finally, Chapter 8 presents conclusions and future work.

Chapter 2: Overview of Supercomputing

Since computers were first invented, they have played important roles in our society by speeding up mathematical tasks so that larger, more fundamental problems can be solved. In the second half of last century, hardware technology has made sustained exponential improvements in performance, but it became clear at the beginning of this century that the physical limits of miniaturization, energy consumption, and heat dissipation made further progress at that rate to be untenable [12]. However, the demand for computing power still continues unabated, and there are no signs of any slowdown. Indeed, numerically demanding techniques such as modeling, simulations, and machine learning continue to be used in increasingly complex approaches for interdisciplinary research such as weather forecasting and astrophysics simulations[5]. Also, brand new approaches in neuroscientific simulations are developed that require even more superior computing capabilities because of the extensive computations that they require.

As a result, new computers were created that had more processing units that could work in a problem in a parallel fashion. These computers that use the combined power of many processors are called supercomputers, and they are programmed in a fundamentally different way than single core computers. Thus, they represent a challenge for the software development community to develop brand new algorithms that are optimized for parallel architectures. The field of parallel programming techniques was created to study algorithms that took full advantage of all the processors clustered in a supercomputing system. Parallel algorithms themselves are not new and has been studied since the 1960s. However, what is new is that the new parallel architectures offers a particular challenge in how to obtain the optimal theoretical speedup in these

new architectures.

In this chapter, I will briefly outline some of the major milestones in the development of supercomputers and new parallel architectures found in GPUs.

2.1 History of supercomputing

When supercomputers were first built, what constituted a "supercomputer" is very different from what it is today. In 1941, Konrad Zuze built the Z3, an operational relay based computer with 64-word floating point binary memory. The Z3 is considered the first first fully functional program-controlled electromechanical digital computer in the world. The first general purpose Turing-complete electronic supercomputer built was the ENIAC (Electronic Numerical Integrator and Computer) in 1946, and it was constructed for the United States Army by the Ballistic Research Laboratory in the Moore School of the University of Pennsylvania to speed the calculation of firing and bombing tables.

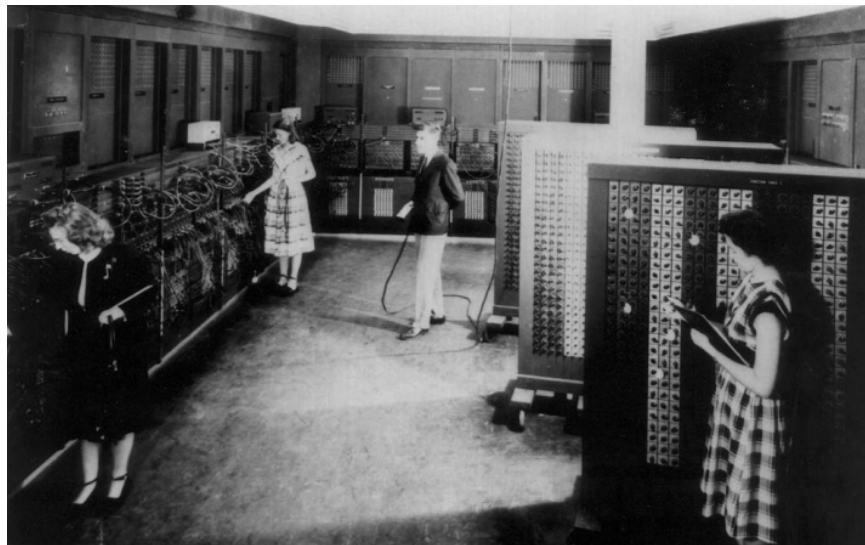


Figure 2.1: ENIAC in the Ballistic Research Laboratory

As seen in Figure 2.1, the ENIAC computer occupied a room. It had 40 units with 18,000 vacuum tubes, 1,500 relays, 70,000 resistors, 10,000 capacitors, and miles of wire. It was eight feet high, three feet deep, and 100 feet long, weighed 30 tons and consumed 130 kW of power. Once completed, ENIAC cost a total of roughly \$500,000 [13].

After these initial milestones in computing, the vacuum tubes were replaced by integrated circuits that were smaller, faster, and cheaper. Moore suggested in the late 1990s that the complexity of the components that could be integrated into a board and duplicated for the same cost [14]. He further predicted that this rate would be sustained or even increased in the years to come. The number of floating operations per second (FLOPS) that a processor can execute has also steadily grown over several decades of computing history. Increasingly powerful hardware allowed for computers to become miniturized into accessible personal computers, and there are now available handheld computer devices that are exponentially more powerful than the ENIAC. However, the miniturizing of computers does not mean that supercomputers were no longer necessary. In fact, it just meant that supercomputing facilities continued to become powerful to solve even more sophisticated scientific problems. What it meant, though, is that these supercomputers no longer needed to be built from custom parts but instead from consumer-grade parts since they were now available to the general public [12].

The U.S. Department of Energy's ASCI (Accelerated Strategic Computing Initiative) finished building the ASCI Red in 1997 and it held the place of the fastest supercomputer for eight years. It was not only the first supercomputer to reach the TeraFLOP mark, but it was also built from 9635 Intel Pentium II Over-Drive processors. This meant that it was a very scalable supercomputer since it was based on consumer level processors [15]. The creation of a computer like this indicated that

bigger computation capabilities could be achieved by adding together thousands of regular, cheap processing units in a distributed way. Every computing node had its own resources and worked in a network to carry out large computations as a single cluster, parallelizing thousands of operations at a time.



Figure 2.2: ASCI Red consisted of 104 cabinets, taking up about 2,500 square feet in the Sandia National Laboratories.

As of 2016, the fastest supercomputer was the Sunway TaihuLight, built in the National Supercomputing Center in Wuxi, China. This supercomputer has over 10,000,000 cores and executes at 93,014.6 TFLOPS [15]. It is based on Sunway many core processors [16]. The Sunway TaihuLight is a huge net growth from the beginning of digital computations, and it will serve to perform intensive computations that could significantly help in the fields of machine learning and deep neural networks. The third fastest supercomputer is the Titan Cray XK7 that has been in the top 3 for the past 4 years since it entered the list in 2012 as the fastest supercomputer. In

additional to CPUs, it has GPUs (Graphic Processing Units), a very common trend in the past decade of supercomputers. With 560,640 cores, the Titan can reach 17,590 TFLOPS [15].

2.2 Performance over time

Moore's law has remained true for about 50 years, but it seemed to have reached its limit in the early 2000's when the power of computers did not double and energy consumption, heat, and other factors indicated that Moore's Law might not predict the next decades [17]. Since then, researchers and scientists have funneled efforts towards finding alternatives for computers to continue growing their power at the rate they were in the past five decades.

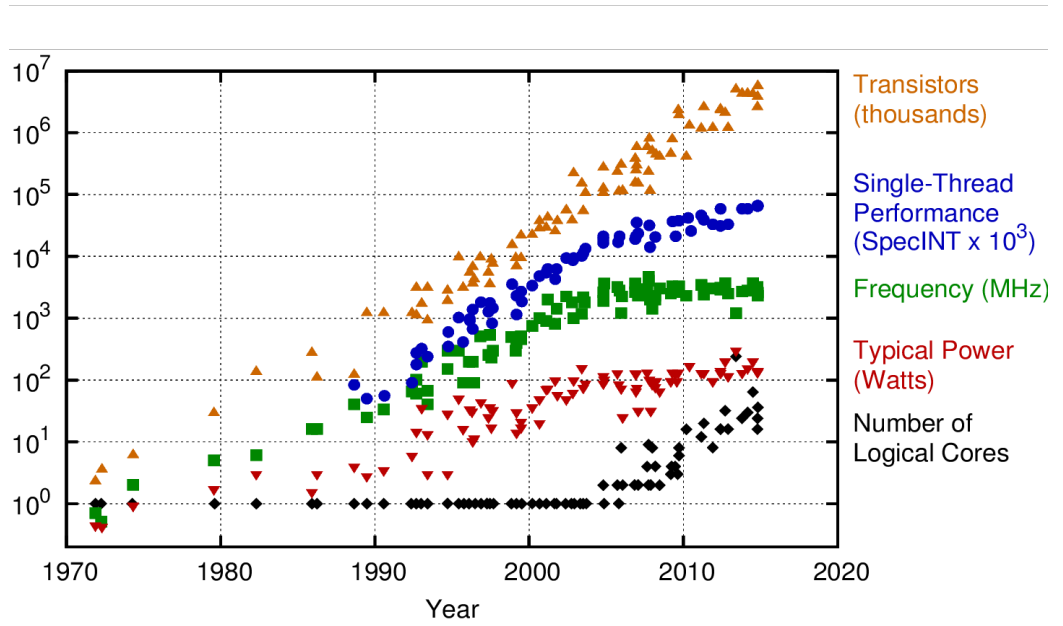


Figure 2.3: A graphic that shows the decline of the clock speed, number of cores, speed per thread, and transistors on chip. Miniaturization has also followed this trend. [18]

To date, the most impactful alternatives for finding increasing computing power at efficient energy rates is graphics cards that were initially designed to speed up calculations in graphics applications and provide control for displays. The first graphics processor units (GPUs) were invented by NVIDIA in 1999 [20], and it had enhanced floating-point performance when compared to CPUs. It became apparent that these GPUs could have more general applications in a wide variety of applications in many fields, and General Purpose GPUs (GPGPUs) were introduced in 2003 to take advantage of the GPU architecture using libraries and toolkits for developers to program the graphics devices directly and interact with the CPUs. Their configuration with hundreds and even thousands more cores than a CPU allow for many operations to be performed at the same time in parallel. As shown in Figure 2.5 these cores can be used as matrices of threads that perform thousands of computations at the same time in multiple data points.



Figure 2.4: The Nvidia GTX 1080 was launched in 2016, supporting 2560 CUDA cores. Most modern user oriented CPUs have 4 to 8 computing cores [18]. These cards are marketed and mostly used for video games and graphics applications but are increasingly being used for optimized general and scientific programming.

As consumer level products, GPUs are sold in a range of prices that are generally very low when compared to most scientific equipment. Any person with a computer can get a top-of-the line GPU for under \$1000. For a system that will parallelize with thousands of cores, this price is a very reasonable one when compared to the cost of an equivalent set of nodes in a supercomputer center. It also performs very efficiently

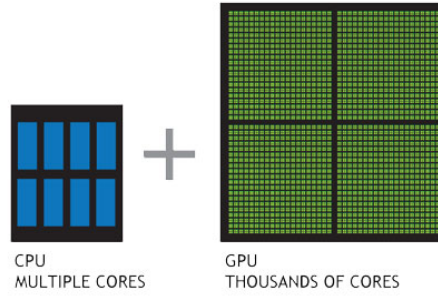


Figure 2.5: GPUs have thousands of processing cores that can be logically handled as 2D or 3D matrices [4].

in terms of floating point operations per second and memory (Figures 2.6 and 2.7) and energy.

The significant gains in performance are due to the nature of CPUs and GPUs; while the former are optimized to execute sequential instructions with sophisticated control logic very efficiently, the latter are designed to execute instructions in parallel. Given the significant performance boost that can be gained from graphics cards technology, GPUs have been increasingly used in scientific fields. The number of scientific papers, software tools, open code repositories and educational resources concerned with GPU technology keeps increasing. Although parallel computing and super computing are not the only options to reinvent computing after Moores’s Law decline, it is looks to be the most promising one to date. Therefore, the popularity of this technology to speed up computation of large problems is likely to continue.

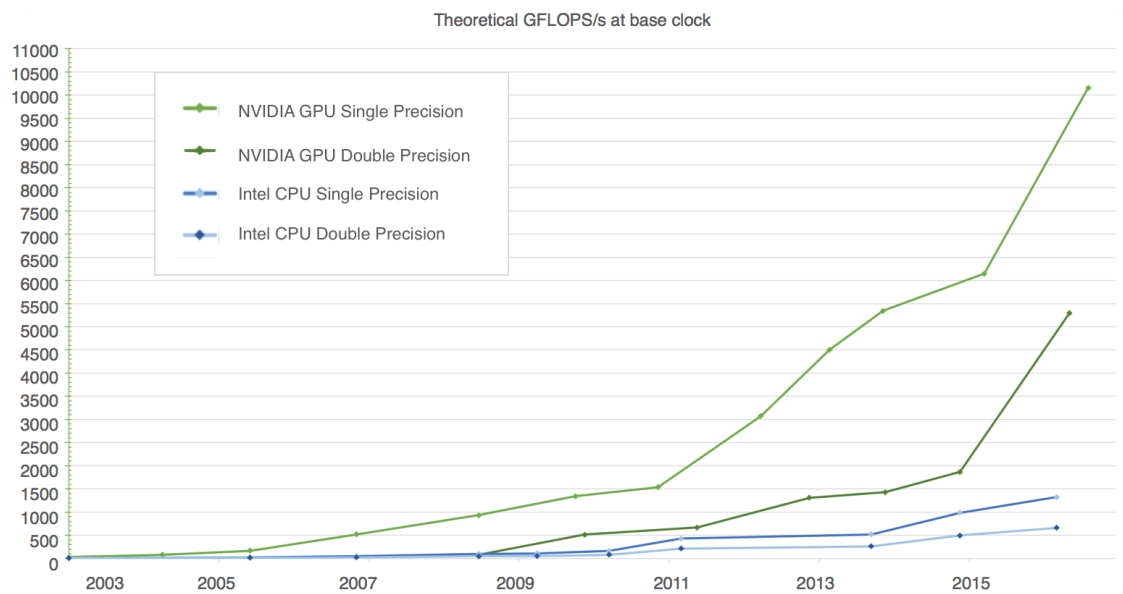


Figure 2.6: This graphic shows how in the past 14 years Nvidia GPUs have increased the number of GFLOPS that they can perform. The growth of this indicator has been largely exponential even for double precision calculations. [4].

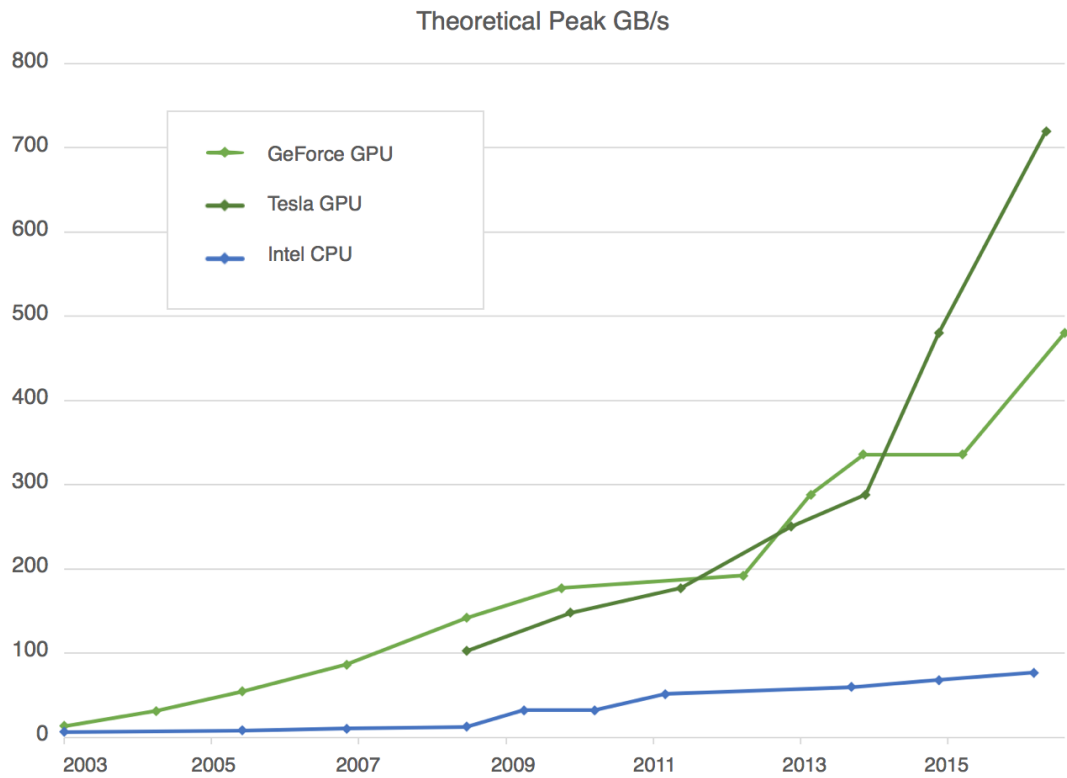


Figure 2.7: Memory is a significant bottleneck in old and new architectures. GPU cards have had a significant improvement in memory bandwidth to better leverage the speed of the parallel calculations it can perform. [4].

Chapter 3: GPU Technology

3.1 Parallel computing

There continues to be a high demand for greater computational speed present in many different problems such as numerical modeling, simulations, machine learning, and data processing. In order for such computations to be useful in their areas, they need to be calculated within a reasonable time. Problems such weather forecasting or astrophysics simulations could take weeks or even months to obtain the needed results, which would render the computation efforts useless or slow down progress. Computer scientists have used the power of multiple computers to solve one single problem by adding up their power to work on subproblems. Systems that can work with different computing nodes together are called parallel computers, and the programming approach is named parallel programming. Parallel computers can have different, heterogeneous architectures and setups: specifically designed on board systems, clustered CPU based computers with different network topologies, distributed clusters, single computers with multi core capabilities, GPU based systems, multiple GPU systems, etc. In principle, parallel computers could reduce the computational speed by an optimal factor of $1/p$, with p being the number of processing nodes available in the system. However, optimal improvement is rarely met since there are communication and memory costs [5], although some of those costs can be hidden by performing parallel operations concurrently with those latencies.

However, for many computational problems, the gain from parallelizing certain computations can be significant. How well a parallel computation performs can depend on the granularity of the computation, the number of subproblems that one large problem can be divided. Communication costs can also increase the computation

time. The higher the computation to communication ratio, the better a parallelized computation will perform. Another important measure for parallel algorithms is the speedup factor which is defined as:

$$S(n) = \frac{\text{execution time using one processor}}{\text{execution time using } n \text{ processors}} = \frac{t_s}{t_p}$$

Alternatively, this can be expressed in terms of the computational steps in the form:

$$S(n) = \frac{\text{number of computational steps using one processor}}{\text{number of computational steps using } p \text{ processors}}$$

Since the computational steps also reflect the speedup, the best way to compare algorithms and calculate speedup is to use the best sequential algorithm available to compare with its parallel counterpart. Very often, the algorithms for the same problem fundamentally differs between the parallel and sequential version [5]. This also means that the nature of the problem highly impacts if there is benefit to be gained from parallelization. If the problem has a lot of interdependent calculations that need to be performed sequentially, then parallel solutions will not lead to a significant improvement.

The best possible speedup on an optimal CPU computation is achieved when the computation time is evenly divided by the p processors used to parallelize the problem:

$$S(n) = \frac{t_s}{t_s/n} = n$$

Superlinear speedup is achieved when $S(n) > n$, and it occurs when the sequential algorithm used for the calculation is suboptimal. Speedup is used to measure the

scalability of an algorithm. A scalable algorithm has a high speedup value, and the algorithm will perform increasingly better as more processors are added to distribute the workload.

Additionally, the type of parallel machine used to optimize the problem influences how the algorithms are coded and ultimately perform. Flynn [19] established very early in the study of parallel computing the classifications for very high speed computers based on the way they issued instructions for data streams. He classified computer architectures into four groups:

1. Single Instruction, Single Data (SISD).
2. Single Instruction, Multiple Data (SIMD).
3. Multiple Instruction, Single Data (MISD).
4. Multiple Instruction, Multiple Data (MIMD).

In his work, he refers to an instruction stream as the sequence of instructions executed and the data stream as the sequence of data used by the instruction stream. Multiplicity refers to several instructions or data manipulations that can be executed simultaneously. A traditional CPU based computer with a single computational node would fall in the first category, since a single instruction. A GPU based computer would also fall in the category of Single Instruction Stream, but with multiple data for executing the single instruction. The remaining categories have been used for fault tolerance and distributed operations. With the G80 architecture, NVIDIA introduced the Single Instruction, Multiple Thread type of parallelism which implied that multiple threads can execute concurrently a single instruction [20]. These classifications describe parallel computations in terms of how their architecture and processing works, since every category can be programmed in a different way.

All of these factors need to be taken into account when addressing a problem in a parallel manner, since each aspect will impact the performance of the computation. Parallelized code is usually designed to make each of the computing nodes perform independent calculations in a balanced manner and data transfers, be it through memory or network communication. If cores spend a lot of clock cycles waiting for other cores to finish their computations, performing conflicting memory accesses or getting data to calculate in, the speedup of the algorithm will drop closer to the sequential performance.

3.2 GPGPU

Graphics calculations require many independent float point calculations. This is because graphic displays are composed of a matrix of rgb (red, green blue) and graphics rendering consists in calculating the color of each pixel. For digital images, each pixel can be calculated, without needing to know the color of the remaining pixels, based on a geometrical description of what is contained in the scene. Therefore, the process of rendering an image entails calculating distances, geometric figures, textures, motion, physics, collisions, intersections, light incidence, reflection, and color intensities for each pixel. These calculations are computationally intensive and depend on floating point operations. For a screen of resolution 1,600x900 (can be used in a reasonable small laptop), over 1,440,000 floating point operations are required. And in videogames they need to be computed fast with a low energy consumption, because the scene will be moving and delays or heating will significantly impact the quality of the experience. Although techniques were created to reduce the amount of computations required (e.g., ray tracing versus polygon techniques), the hardware optimizations were necessary as well. As a result, GPUs were developed to reduce the time spent in rendering more complicated images as the computer graphics techniques

progressed. GPUs were developed with many more cores than CPUs (see Figure 3.1), memory, cache, and control structures that made it easier to calculate all pixel values in parallel, thus significantly reducing the time to produce a digital image.

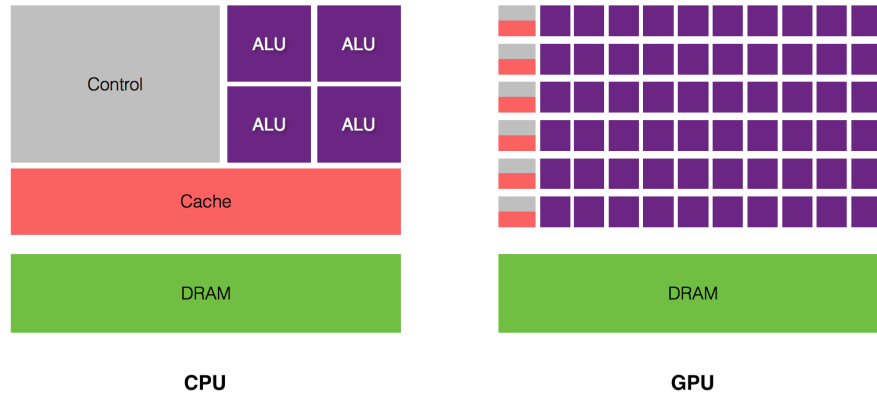


Figure 3.1: CPUs have fewer ALUs, a bigger Control Unit, a single cache and memory. GPUs have a lot more grouped ALUs that share their own control and cache and global memory for all the cores to access.

Starting in the 2000's the two major GPU producing companies, ATI and NVIDIA, have competed for the market share in graphics cards, mostly for videogame systems. This has led to a steep increase in the computational power of the cards. GPGPU was introduced early in their development so there was also competition for supercomputer systems and scientific usage. Frameworks, interfaces, libraries, and toolkits were created for graphic and scientific computing. OpenCL[21], CUDA[22], OpenACC[23], OpenMP[24], MPI[25], OpenVIDIA[26] and other frameworks have been created over the past 15 years. The set of languages that can be used has also grown, while originally C and C++ were the options, now Python, Matlab, Java, etc, can be used. Additionally, libraries with physics, fast fourier transform, linear algebra, higher level programming abstractions, random number generation, graphics and countless other free implementations on public code repositories.

3.3 CUDA framework

The NVIDIA CUDA framework works in heterogeneous parallel computing systems. This refers to computers that host both a CPU for central control and a GPU for graphics and/or GPGPU computing. This framework was released by NVIDIA to work with their GPUs and it has been widely adopted in scientific usage for parallelization when the challenging problems are not big enough to require a supercomputer or if it is not possible to access one. The Figure 3.2 shows how the flow of a heterogeneous program starts with the host CPU execution that can launch device kernels. Kernels execute grids of thread blocks in the device and then return to the host when all thread blocks complete execution. All threads in a kernel launch execute the same device function on the data. CUDA programs have code to be executed both in the CPU and GPU (host and device) that can be compiled with the NVIDIA C Compiler (NVCC) [12].

Computationally expensive tasks have a lot of operations to perform on a lot of datapoints. Most of these problems can be broken down to process a subset of the data, leading to data parallelism. Each of the threads launched in a kernel call can access a particular data point, and the CUDA framework provides a way of doing it via the `threadIdx` and `blockIdx`, which are variables handled by the framework. This allows that thread i can access and perform calculations on the i th element of a data structure without interfering with neighboring calculations. This execution model is the base for data parallel code to run, and it is flexible enough to allow ways for the calculations to be fully independent or, if necessary to solve the problem, introduce some interaction and data dependency[12].

The thread execution hierarchy in CUDA allows all threads to be identified by coordinates. All threads in a block share the same `blockIdx`. This hierarchy stemmed from the nature of the graphics calculations that the GPUs are designed to address,

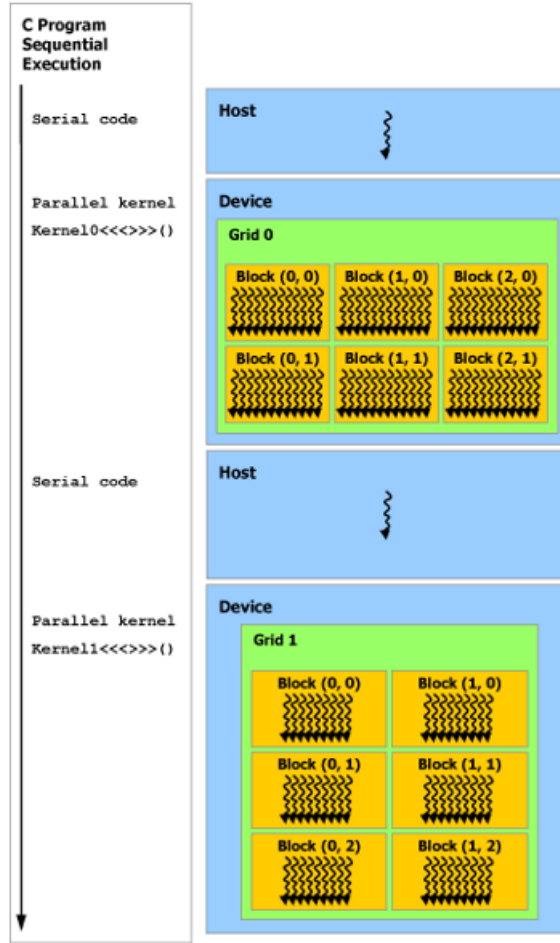


Figure 3.2: The diagram shows how GPU kernel executions alternate between single thread CPU executions to produce a heterogeneous program [4].

but it also fits the nature of other problems in which the data or the algorithm works with levels of embedding. When launching a kernel, the program specifies the size of the grids and blocks, which can be one, two or three dimensional. Since they are parameters to the CUDA execution, the parallelism that can be performed can be scaled up according to the size of the problem, to some architectural limitation. Once the grid is launched, the dimensions in which threads are arranged do not change. Choosing whether a grid and block should be three dimensional largely depends on the nature of the problem and the potential gain of processing the data in one way

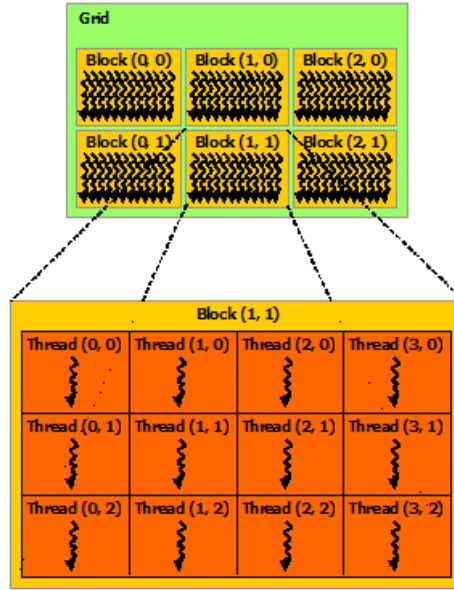


Figure 3.3: Each kernel is launched with grid and block dimensions. A grid contains a group of blocks and a block contains a group of threads. Blocks and threads have identifiers that match their indices in the parallel hierarchical structure. [4].

or another. Vector addition for example, is a problem that is usually solved with a single dimensional grid and block. Matrix multiplication of a 2D matrix, due to both arguments being two dimensional, naturally fit with using two dimensional blocks and grids. Three dimensional problems such as particle or weather simulations will more likely be processed in three dimensional grids and blocks[12].

Threads in a block are the units to be executed simultaneously in the GPU, although the order in which the blocks are executed is not guaranteed. To synchronize threads in a block, there are barrier functions such as `__syncthreads()` that work to synchronize threads within the same block. Since blocks perform in an independent way, this allows for scalability across CUDA devices as shown in Figure 3.4.

Another aspect of CUDA programming that impacts the performance of the parallel code is memory usage. Memory-bound programs can benefit from appropriately choosing how to allocate the memory, according to the CUDA memory types (listed in

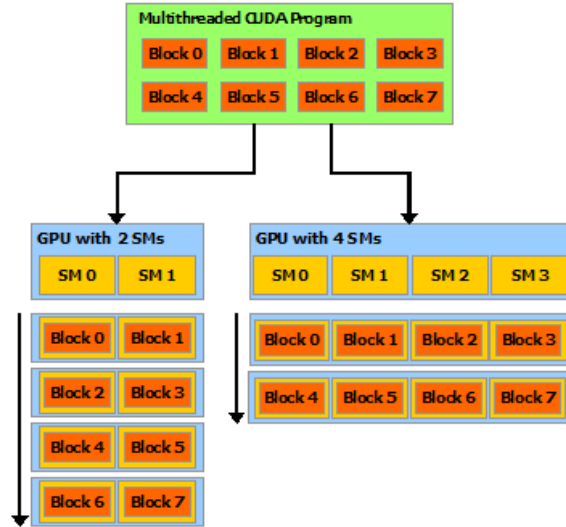


Figure 3.4: Devices capabilities change in terms of the number of blocks they can perform in parallel but the CUDA code is transparently scalable. [4].

Table 3.1). Each type of memory is designed for a specific purpose. Off-chip memory types are slower and present the problem of the bandwidth consumption. They do need to load data from the host in the cases of the global and constant memory, so they are memory types for the host to communicate with the device and to distribute memory across blocks. On-chip memory types have lower latency, are faster to access and require less energy. Registers are the fastest of the options, but using a high amount of them can reduce performance as there are limited registers to be used by a Streaming Multiprocessor (SM). Shared memory is very useful to reduce computation times as it is faster to use than global memory and is still visible by all threads in a block, thus allowing cooperation between threads to load and update the data.

3.3.1 Dynamic Parallelism

When CUDA was released, kernel functions could only be launched from host code, which caused programmers to come up with workarounds for recursive algorithms. This forced adaptations of algorithms that were not intuitive and were very demanding

Variable declaration	Memory	Scope	Lifetime	Host access
<code>int variable;</code>	Register	Thread	Kernel	No
<code>int arrayVar[5];</code>	Local*	Thread	Kernel	No
<code>__shared__ int sharedVar;</code>	Shared	Block	Kernel	No
<code>__device__ int globalVar;</code>	Global*	Grid	Application	Yes
<code>__constant__ int constVar;</code>	Constant*	Grid	Application	Yes

Table 3.1: Cuda memory types and characteristics[12]. *Off chip memory

in memory bandwidth. Dynamic Parallelism was introduced as an extension to the framework that works in the newer architectures (compatible with CUDA computing capability 3.5 and higher) to allow device kernels to launch new kernels to adapt to certain kinds of problems. It is very useful to problems that require a varying amount of work per block and need to dynamically allocate it. Recursive problems such as the Mandelbrot, Bezier curves, and tree implementations have greatly improved by the addition of Dynamic Parallelism to the framework. An example of this can be seen in Figure 3.6, where regions of the analysis require different resolution in the computation while other areas require significantly less work to be processed.

The syntax for a nested kernel launch is the same as it is to launch a kernel from the host. Memory rules and types apply the same for nested blocks in execution as regular blocks that were launched in CPU code. Global memory remains visible for parent and children blocks, as is constant memory. Local memory remains exclusive to each thread and shared memory is only visible within the block scope. CUDA memory allocation and release can be performed from a device kernel bound to the capabilities of the memory heap on the device, but behavior is undefined if memory that was allocated in the host is released in a kernel and vice versa. Kernel calls from device functions are non blocking and they behave the same as launching the threads from the host. Synchronization can be called on device level for all threads in all blocks (nested or not) with `cudaDeviceSynchronize()` and `__syncthreads()`.

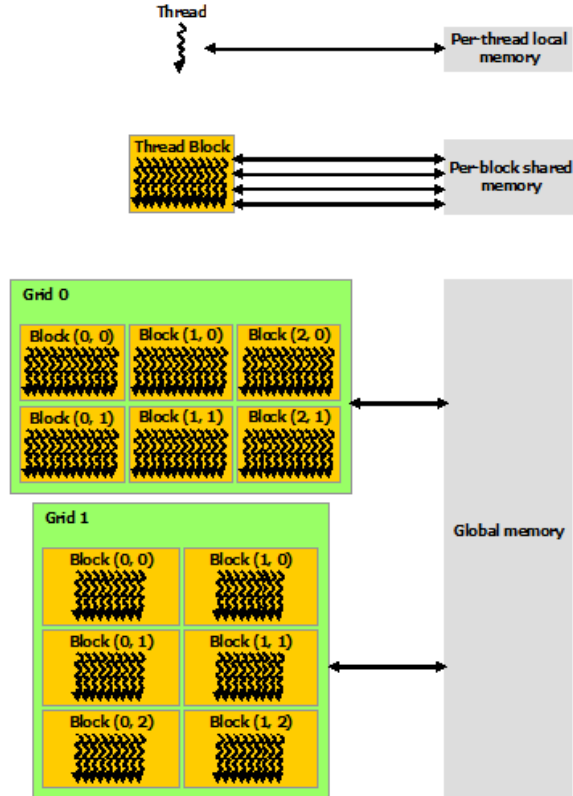


Figure 3.5: The diagram shows how the memory types that can be allocated by CUDA programs are distributed in the hardware. [4].

However, the runtime will make sure that children blocks are synchronized before exiting the parent kernel to prevent undefined behaviors in case the parent is finished before the children blocks.

Dynamic parallelism is a very flexible addition to the CUDA framework since it allows more intuitive programming for certain types of algorithms. In particular, this work will make use of it for the construction of an octree, an inherently recursive data structure. In the past, algorithms that built octrees would have to work with locks and flat data structures in convoluted tree walks and insertions to keep track of nodes and leaves [28]. In the upcoming chapters, an implementation of the Barnes-Hut tree that makes use of dynamic parallelism will be explained and used in an application

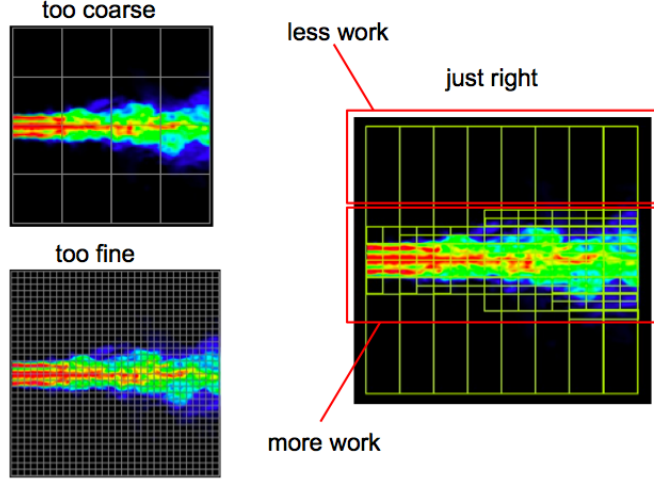


Figure 3.6: A fluid simulation with an adaptive mesh. [27].

for Molecular Dynamics Simulations.

3.4 Further considerations

CUDA is not the only framework for GPGPU parallel computing, but it is the pioneer and it works reliably with NVIDIA GPUs and is frequently updated for both video game technology as well as general high performance computing. For the development of the present work there were computers with NVIDIA GPUs GTX 480 readily available. Even if these are not top of the line graphics cards, the CUDA framework and NVIDIA support backwards compatibility, so the year the cards were released is not a problem as long as the architecture is Fermi or newer. This equipment is not expensive to get and the setup is affordable and simple, as well as vertically scalable via the Scalable Link Interface (SLI) of NVIDIA. As a result, working with this setup has accessibility advantages when compared to a big cluster or supercomputer, while still allowing to test the potential of the parallel algorithms and optimizations. Even if regular CPUs are still capable of some degree of parallelism, launching threads in a CUDA architecture is significantly less expensive than it is in CPU architectures due

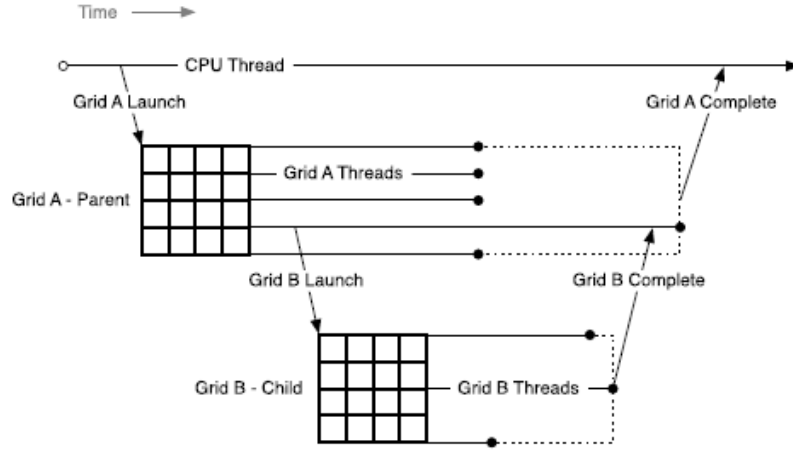


Figure 3.7: Device kernels can launch nested device kernels as host code does. [4].

to the nature of their control structures. However, there are particular challenges to work with CUDA parallelism. Coalescing memory is achieved when contiguous memory locations are used by the threads in the same phase of execution, thus reducing memory loads of new data blocks. Achieving coalescing memory is very often far from straightforward, particularly in algorithms that make use of memory in very irregular patterns. When working with dynamic parallelism, it is important to keep in mind that launching grids with very few threads will affect the performance in a negative way, since the overhead of the kernel launches might dominate the cost of the full computation. And finally, grid and block sizes need to have careful consideration to obtain optimal performance since the amount of blocks that can be executed in a SM and the number of warps that can be scheduled with reduced latency.

Chapter 4: Molecular Dynamics

High performance computing has been used for interdisciplinary scientific research in the past, and one of the areas in which it has had success and relevance is Molecular Dynamics. All forms of life are made of biomolecules that perform varying biological functions to keep cells working and organisms alive. Proteins, RNA, DNA, lipids, enzymes, ribosomes, etc. They are all biomolecules crucial for life and reproduction of cells [29]. These biomolecules undergo a replication processes and fold to form a three dimensional structure that determines how it works and what functions they will have [32]. Studying the dynamics of how molecules are created, interact with other molecules and different forces is an area in which biophysicists and biochemists have a lot of information to gain. Consequently, simulations that can provide clearer insight on such processes are of great value.

In this work, two types of biomolecules that will be the focus for the simulations: ribonucleic acids (RNA) and proteins. Proteins are made of strings of 20 naturally occurring types of amino acids and can average a length of 400 of these beads [33]. Proteins can have very complex three dimensional structures and interact with other proteins to carry out different tasks in the cell. RNA is made of 4 types of nucleic acids linearly chained, they can also interact with other biomolecules and are a center component in cell reproduction. Biomolecular complexes like proteins and RNA assemble into biomolecular machines and have specific functions in the cell, such as movement, energy production and transport [30].

4.1 Biomolecular Folding

Folding is the process through which biomolecules assemble into their final tertiary structure, which is a very specific three dimensional form to perform their functions [31]. This process can follow a number of pathways from the string to the final structure, and there can be errors inbetween, which can be completely harmless or on the other hand, have severe consequences for the cell and organism. Biomolecular folding is different from biomolecular structure prediction, which makes use of statistics and the primary structure to predict the final structure of the biomolecule. Such methods have had success in the past predicting the outcome with reasonable confidence [39], however they do not provide information about the intermediate steps.

It has been discovered that folding errors leads to several diseases in humans like Alzheimer's and Parkinson's [35]. For this reason, folding information has gained relevance among research communities, highlighting the importance of understanding how it works in biomolecules. As a result, Molecular Dynamics simulations became a very useful tool to study folding pathways and to work on prevention and treatment of diseases.

4.2 Molecular Dynamics Algorithm

Experimental work can be used to observe the unfolded and folded states of biomolecules, but current technology is not capable of enabling observation during the folding process. This is because the resolution of the timescales of experiments are inaccessible to folding events, which are on the microsecond timescale for the fastest folding proteins. Molecular dynamics simulations are capable of showing the intermediate time periods of the folding process, with defined time intervals of about one picosecond (10^{-12} seconds) [36].

Molecular dynamics is the study of systems of atomic or molecular structures and the way they interact and behave in the physical world. Simulations follow the equations and physical rules in the system showing how the molecular structures would behave in experimentation. They can vary in their granularity level going from simulating electrons to atoms to groups of atoms in different organizational levels. 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 [40, 41] 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 energy gradient is used to compute the forces acting on each bead. The forces acting upon each body produce a new set of positions and velocities by solving Newton’s equations of motion ($\vec{F} = m\vec{a}$) after

a time interval Δt . The process is repeated until a series of sets of positions and velocities (snapshots) result in a trajectory [9].

Since the equations of motion cannot be integrated analytically, many algorithms have been developed to numerically integrate the equations of motion by discretizing time Δt 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 $\vec{r}$. ζ is the friction coefficient and Γ is the random force.

When the Langevin equation is numerically integrated using the velocity form of the Verlet algorithm [8], 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 Δt is given by

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

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 [51].

4.3 MD Simulations Granularity

In computational simulations and in general with calculations, precision and time require a tradeoff. Molecular dynamics simulations are no exception, there are various levels of precision at which a simulation at molecular level can be run. If necessary, a simulation can be designed to represent biomolecules at quantum level by doing calculations over each individual electron in the structure. However this would result in a system way too large and a simulation that is too computationally expensive. At this resolution of the problem, the size of the problem and the time that can be computed reduces to only about 100 atoms and the time scale to the order of femtoseconds (10^{-15}) [42, 41].

The next level of granularity could be atomistic simulations, each bead represents an atom in the system. This level allows simulations to run for more time and can handle larger systems than quantum level simulations. But it would still stay in the range of 50 amino acids and less than a microsecond of folding process in a reasonable computation time [46]. Average biomolecules can be up to 8 times larger than such

length and fold much more slowly. A resolution that has been found to perform reasonable simulation time and size is coarse-grained. This type of simulations can increase the timescale of the experiment while still accurately capturing folding and binding mechanisms of biologically relevant protein and RNA biomolecules. They group atoms together in a higher level structure that is represented with a bead or a group of beads. This reduces the number of particles to represent a system, thus computing more time than both quantum and atomistic level simulations.

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_α or C2' position) for proteins or RNA, respectively, thereby reducing the total number of simulated particles[42, 41]. Recent studies have demonstrated that MD simulation of biomolecules using the SOP model reproduces experimentally measured mechanical properties of biomolecules with remarkable accuracy [47, 48, 49].

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

$$\begin{aligned}
V(\vec{r}) &= V_{FENE} + V_{SSA} + V_{VDW}^{ATT} + V_{VDW}^{REP} \\
&= - \sum_{i=1}^{N-1} \frac{k}{2} R_0^2 \log \left[1 - \frac{(r_{i,i+1} - r_{i,i+1}^0)^2}{R_0^2} \right] \\
&\quad + \sum_{i=1}^{N-2} \epsilon_l \left(\frac{\sigma}{r_{i,i+2}} \right)^6 \\
&\quad + \sum_{i=1}^{N-3} \sum_{j=i+3}^N \epsilon_h \left[\left(\frac{r_{i,j}^0}{r_{ij}} \right)^{12} - 2 \left(\frac{r_{i,j}^0}{r_{ij}} \right)^6 \right] \Delta_{i,j} \\
&\quad + \sum_{i=1}^{N-3} \sum_{j=i+3}^N \epsilon_l \left(\frac{\sigma}{r_{i,j}} \right)^6 (1 - \Delta_{i,j})
\end{aligned}$$

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: $k = 20 \text{ kal}/(\text{mol} \cdot \text{\AA}^2)$, $R_0 = 0.2 \text{ nm}$, $r_{i,i+1}^0$ the distance between neighboring beads in the folded structure, and $r_{i,i+1}$ the actual distance between neighboring beads at a given time t .

The second term (V_{SSA}), 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 (V_{VDW}^{ATT}) 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 $8 \text{\AA}$ 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 (V_{VDW}^{REP}) is a Lennard-Jones type repulsive term for van der Waals interactions between all pairs of beads that are non-native, and σ , the

radiuses of the interacting particles, is chosen to be $3.8\text{\AA}$, $5.4\text{\AA}$, or $7.0\text{\AA}$ 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 in the next chapter.

4.5 Molecular Dynamics with GPUs in Other Models

MD simulation software has already successfully parallelized on HOOMD[44], LAMMPS [45] and NAMD [43].

Chapter 5: Cutoff Methods

As described in detail in the previous chapter, MD simulations are defined by an energy function that is used to 1) evaluate the forces between each of the interacting beads and 2) update the positions and velocities for a successive time step. This process is repeated over and over until the end of a simulation trajectory. Since the update of the positions and velocities depend on the previous step, this process is inherently sequential and there exists no method of reliably parallelize this process to date. However, the evaluation of the forces between the interacting beads is independent and can be parallelized. Even still, the number of interactions is still very large and grows $O(n^2)$ and consists of calculations of interactions that are negligible. Even before parallel algorithms were applied to MD simulations, cutoff methods were developed that first determined which interactions were significant before computing them. In this chapter, we will briefly review some commonly used cutoff methods and recent efforts to parallelize these algorithms for the GPU.

5.1 Motivation for Cutoff Methods

In this section, we illustrate how cutoff methods generally work by using the well-known N-body problem as an example. According to Newton's Law of Universal Gravitation, the interaction between two bodies is inversely proportional to the distance between them[50]. His formulation is shown below:

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

Where G is the universal gravitational constant, m_1 , and m_2 are the masses of the two interacting bodies, and r is the distance between them.

It can be readily seen that as r increases, the value of the interaction force between them, F , also decreases progressively towards zero. The implication is that for a distance sufficiently large, bead interactions are negligible in the force calculations and can be eliminated without major impact on the precision of the model. This behavior holds true for other MD simulation approaches, such as the SOP model, since the distance between interacting beads is also in the denominator of the most computationally intensive force calculation terms. In this work in particular, this is the case for the van der Waals interactions described by the Lennard-Jones equation, and they will be a particular focus of optimization.

Without cutoff methods, the naive all pairs approach would consist of calculating every interaction for each timestep, and it has a computational complexity of $O(n^2)$ for each timestep. The trivial straightforward parallelization of this problem will result in a better computational time proportional to the processing units added, optimally $O(n^2/p)$. However, there would still exist memory transfer time and parallelization overhead that can prevent the trivial algorithms from having low execution times. Therefore, it is desirable to reduce the number of interactions while still maintaining accuracy, even in the parallel case, to optimize the MD simulation performance. This requires new algorithms that are specifically optimized for the parallel architecture. Some of the most widely used cutoff methods will be reviewed in the upcoming sections, as well as some recent implementations of GPU-optimized parallel variants of those algorithms.

5.2 Verlet Neighbor List Algorithm

The widely used Verlet Neighbor List algorithm [8] pre-calculates the distances between pairs of interacting particles to determine whether their interactions are significant enough to calculate the computationally expensive force calculation. As such,

out of all possible force calculations that could be made between pairs of interacting particles in a naive approach, only a subset need to be made while still maintaining the accuracy of the simulation but improving performance. The main parameters needed for the Verlet neighbor list algorithm are two distance cutoff radii, an outer skin layer and an inner cutoff one. First, a Neighbor List is created containing only the interactions for which the distance is smaller than the skin layer radius (r_l). From the Neighbor List, a Pair list is computed of interactions in which the distance is smaller than the cutoff radius. The Neighbor List is updated every N timesteps, such that interactions in the pair list does not have enough time to move beyond the neighbor list, and the closer interactions in the Pair List are recomputed in every timestep. The complexity of this implementation is of $O(nr_c^3) \approx O(n)$, which is an important performance optimization when compared with the naive approach.

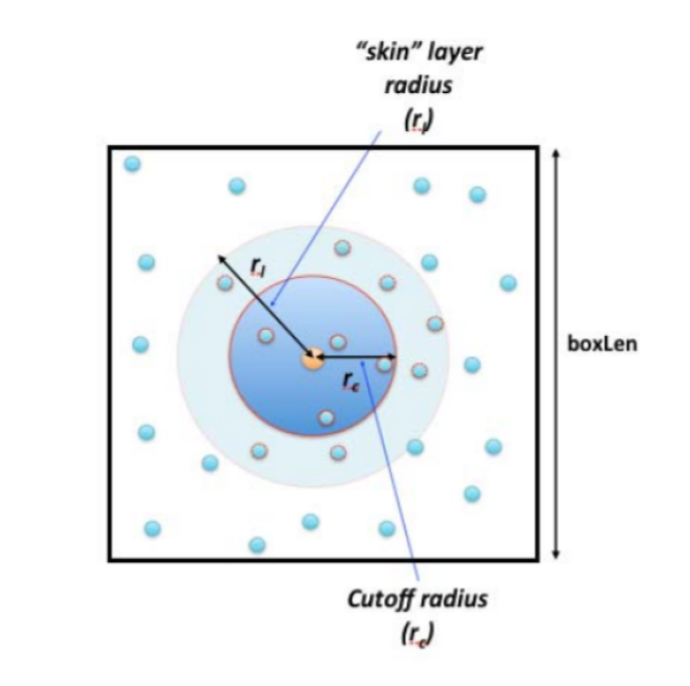


Figure 5.1: The light blue area is the skin layer radius and is recalculated every N timesteps. The pair list is calculated from the verlet list, thus examining only a subset of the total possible interactions [51].

5.3 GPU Optimized Parallel Neighbor List

A naive parallelization of all possible pairs of interacting beads would be expected to improve performance of MD simulations on the GPU, but this would be prohibitively computationally expensive because a) large sets of unnecessary calculations would be performed, and b) a very large list of interacting particles would need to be maintained on a device that is memory limited. As such, cutoff methods or other similar approximations can improve the performance of MD simulations on the GPU.

The Verlet Neighbor List algorithm, at least the way it was originally conceived, required generation of two subset lists from larger lists, which is inherently sequential. Therefore, implementation of the Verlet Neighbor List in its original form would be inefficient to compute in GPU due to the required memory transfers and lack of parallelization. In principle, the list construction, since it is inherently sequential, could be executed on the GPU by launching a single block with a single thread that is slower than the equivalent thread on the CPU. This approach would not only be slower than simply running the algorithm on the CPU, but it would waste a lot of available GPU threads. Another approach could involve calculating all of the distances between all pairs of interactions on the GPU and then transferring that information to the CPU to generate the list. However, this approach would involve a lot of memory transfer between the CPU and GPU. Ideally, we would need a parallel algorithm that can be performed entirely on the GPU without the bottleneck of large memory transfers.

Lipscomb et al. introduced a novel GPU-optimized parallel neighbor list algorithm that could be performed entirely on the GPU [51, 52]. The first step for the parallel neighbor list is to perform a key-value sort on the pair list, using the a Member List as keys and the Master List as values. The pairs in the Member List are interactions that have already been evaluated to be within the cutoff distance of the bead. The

Master list contains boolean values represented by integers (1 for true or 0 for false) representing membership of the smaller subset that will be used to calculate the force. Sorting the boolean values will make the “true” values occupy the first positions in the list and all the “false” values will be moved to the end. As a last step, a parallel scan outputs the number of true values. Then, a parallel copy of the n “true” top values of the list to the Neighbor List will produce the final data to begin the force calculations. This process produces a Neighbor List equivalent to the Verlet Neighbor List but using only parallel operations that do not require additional load in memory bandwidth [51]. Paired with memory optimizations and type compression techniques, this variant of the Neighbor List algorithm performs with very good results and scalability. In [51], the parallel neighbor list performed very similarly to the parallel all-pairs calculation in smaller systems but the performance increased significantly with increasing n while still maintaining accuracy. In their implementation, the Data Parallel Primitives Library (CUDPP) is used to optimize the computation of the lists, as seen in Figure 5.2.

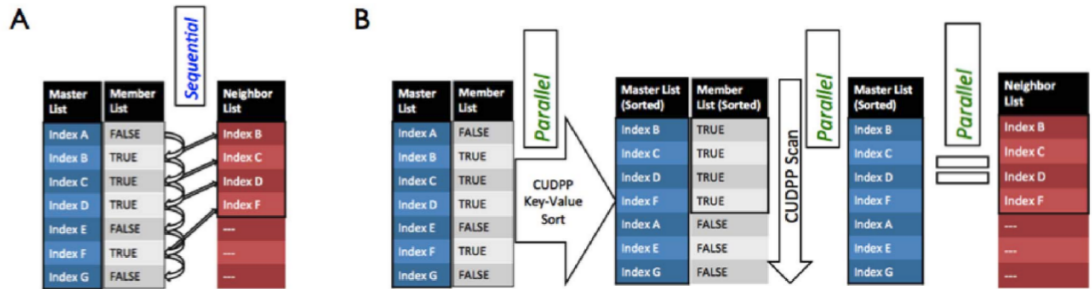


Figure 5.2: The process in the left shows how the inherently sequential Verlet Neighbor List performs the cutoff. On the right, the parallel version uses parallel sort and scan, producing a pair list with all the computations performed on the device. [51].

5.4 Cell List Algorithm

The Cell List algorithm was first thought as a way to speed up liquid simulations and it works by dividing the space into cells and assigning each bead to one of these cells [9]. In a two dimensional case, each cell has six neighboring cells plus itself to be considered for close enough neighboring beads as shown on the left side of Figure 5.4. A three dimension case that would be used in MD simulations would have 26 neighboring cells plus itself. A big advantage in this approach is that the Euclidean distance, which is relatively computationally expensive to compute, does not need to be calculated for the creation of a pair list. In the original implementation for the CPU, a linked list is maintained to keep track of beads in neighboring cells [9].

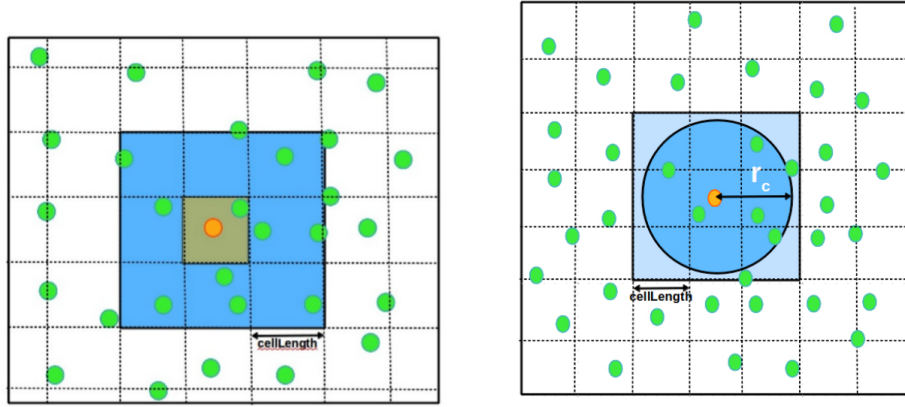


Figure 5.3: Each cell has a list of beads assigned that will be used to compute distances and forces with its own beads and beads in neighboring cells. Left diagram shows the traditional cell list algorithm and in the right the hybrid cell list approach [52].

Several parallel implementation methods for the Cell List cutoff method have been proposed. The main difference is that GPU implementations do not use linked lists due to overhead in handling dynamically allocated memory, pointers, reallocation and resizing that would be prohibitively expensive on the GPU. [52]. Each cell can be assigned to a thread block, which splits the computational load into natural units

in the GPU. This approach could lead to a high rate of unused threads in a block, driving down the performance. To avoid this, in [52] a proposed implementation worked by checking pair lists according to the cells that both of the beads belong to and determining if they are neighbors.

5.5 Hybrid Neighbor/Cell List Algorithm

The best performance was achieved via a combination of the Neighbor List Algorithm and the Cell List algorithm [52], depicted in the right side of Figure 5.4. This approach uses the pair list structures to create first a sublist based on the Cell List algorithm by checking if the cells are neighbors. This Neighbor List is updated every N timesteps. Then based on the Neighbor List, distances are computed and compared to the cutoff radius, they are added to the final pair list if the distance is within r_c . The Hybrid Neighbor and Cell List approaches have been found to perform about 10% better in all cases as compared to the Parallel Neighbor List algorithm [53, 54].

Chapter 6: Barnes-Hut Algorithm

N -Body problems such as planetary, stellar, galactic, and cosmological simulations are very good targets for parallelization. In some regions of these and related simulations, the level of detail required for an accurate representation can vary depending on how the system evolves. One can perform the entire simulation with a fixed fine level of detail to maintain accuracy, but this may require too much in terms of performance. Ideally, one could use fine grids in certain regions of the simulations while other regions could use coarse-grained grids when accuracy is not as important. Even though they use different granularity and force calculations, the basic principles remain and they are also calculated in $O(n^2)$ time, just as is the case with MD simulations.

A widely used force-calculation approximation algorithm used in N -Body simulations is the Barnes Hut algorithm [11]. In this approach, the algorithm hierarchically decomposes the space around the bodies into successively smaller grids that contain the summary information about the bodies contained within the grids and computes the interactions between the grids rather than between all of the bodies. Barnes and Hut created their algorithm based on hierarchical trees that represent square (or cubic) regions of the system space [11]. It works as an adaptive grid that adjusts the resolution according to the number of datapoints in each region. The nature of this algorithm inherently fits the Dynamic Parallelism scheme aforementioned in Chapter 3, since it dynamically determines the amount of computational work to be applied in each subregion of the problem. Although this method is mostly used for dynamics specific to astrophysics, it was designed to work with varying forms of the N -Body problem and scale very reasonably in systems with many particles that need force

calculations.

There are three important gains in performance that can stem from this technique. First, the construction of the tree scales on the order of $O(N \log(N))$, thus creating an auxiliary structure that is very efficient to build. Second, finding any bead in the tree is also very fast, as it takes $O(\log(N))$ time and even traversing the tree is in the order of $O(N)$. Lastly, all internal nodes in the tree can be used as approximations for the leaves that they contain; this is achieved by storing a center of mass, mass value, and width of the subregion in each one of the nodes. This creates a structure that is both capable of approximating distances without having to recompute distances for every bead and quickly locating the beads spatially. There is a tradeoff to be evaluated when using approximations in the tree, and it can be controlled via a value θ according to this equation:

$$\frac{l}{D} < \theta$$

where l corresponds to the length of the subregion represented in the node and D is the distance from the center of mass to the bead that is being calculated. θ is a fixed parameter for the algorithm that will determine the balance between precision and speed when traversing the tree. When the inequality holds, this means that the ratio of the size of the sub cube and the distance from its center of mass is enough to use this internal node as an approximation of the cells it contains. Otherwise, the algorithm recursively visits the children of the tree and performs the calculations. This allows the convenience of choosing a parameter that will give the simulation a balance depending on time and precision.

In the original paper [11], the tree construction was meant to be done at every timestep, and it starts with a root node that is assigned the whole virtual 3D space of the system. Each particle is inserted in the tree by determining a sub cube contained in the current virtual volume, then insert recursively following these rules:

1. If the particle falls in an intermediate node, find the next subcube and insert recursively into the appropriate subtree. This represents a step where the bead was assigned to a cell that is already subdivided, so it is necessary to find a smaller subcell.
2. If the particle finds an empty leaf, insert in place. Meaning, the bead is located in a cell that is empty and does not need further subdivision.
3. If the particle finds an occupied leaf, subdivide the virtual cube that this node represents and create eight new children for each new sub cube. Then recursively insert both particles. This happens when the bead is inserted in a cell that already contains one other bead, then the space needs to be partitioned.

The insertion process is repeated as many times as particles are in the system and the final form of the tree will have a particle in each leaf of the tree. Figure 6 shows on the left an initial 2D space with particles, the middle image shows the virtual partitioning of the space that produces an adaptive grid where every cell contains a single body. The image on the right is a quad tree used to represent the hierarchical partitions in the space.

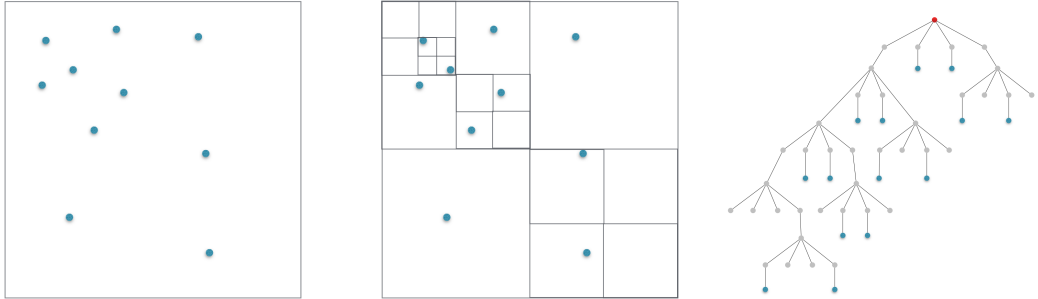


Figure 6.1: The space on the left represents the original state of a toy 2D system. In the middle, the representation of the virtual space partition and in the right the corresponding hierarchical structure for sub quadrants [28]

6.1 Parallel Barnes-Hut Tree

Parallel implementations of the Barnes-Hut algorithm have been previously implemented for N -body simulations. One implementation is shown in [28] and closely follows the steps explained in the original proposal of the algorithm, implementing a kernel for each one of the enumerated tasks as shown below:

```
read input data and transfer to GPU
for each timestep do
    1. compute bounding box around all bodies
    2. build hierarchical decomposition by inserting each body into the tree
    3. summarize body information in each internal octree node
    4. approximately sort the bodies by spatial distance
    5. compute forces acting on each body with help of the octree
    6. update body positions and velocities
end
transfer result to CPU
```

Each of these kernels can have memory optimizations, procuring coalesced accesses and thread coarseness techniques. The sort step helps with the coalescing memory accesses, the size of the blocks can be chosen to maximize the number of threads executed in parallel per each streaming multiprocessor. For instance, leaves and internal nodes of the tree are stored in the same array, with the beads in the first N portion of the array and the nodes in the remaining positions. The crucial kernel is the building of the tree, because the tree is the auxiliary structure that will speed up the next steps in the simulation and it performs faster than its serial counterpart ($\sim O(n/p \log(n/p))$). In his work [28], Hwu uses lightweight locking, thread throttling, and atomic operations to perform the insertion in each leaf as shown in Figure 6.2. The atomic operation (`atomicCAS()`) is used to compare and swap in this

implementation, and it works as an efficient lock on a leaf index and insert without risk of race conditions or inconsistencies in the tree building. The bodies in the tree are sorted to group beads that are spatially close together in the array. This helps in contiguous memory accesses, thus speeding up performance. To calculate the forces, bodies are assigned to parallel executing threads in a Round Robin fashion. Each of these threads traverse the tree, finding neighboring cells and adding the interaction to the force summation.

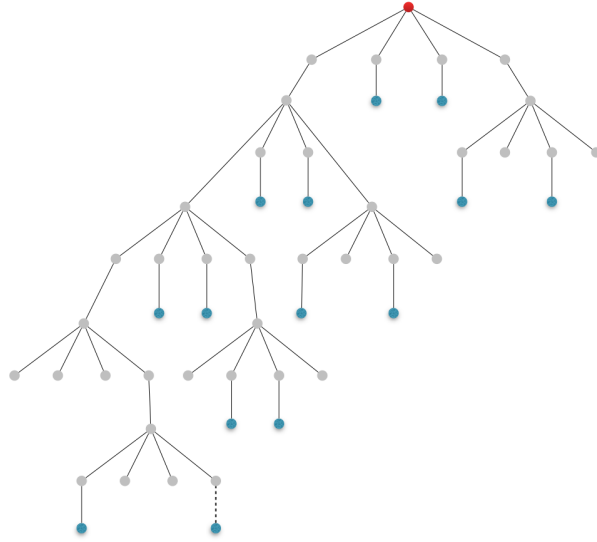


Figure 6.2: The dashed line represents a lightweight lock to insert a bead in the leaf on the Barnes Hut tree construction [28].

6.2 Dynamically Parallel Barnes Hut

In [5], a dynamically parallel implementation of a quadtree was shown partitioning 2D space into subregions and building a hierarchical structure to represent the scheme. This implementation launched an initial thread-block, representing the root node of the tree. One of the threads in the root thread-block launches four nested thread-blocks more if there are enough points to be processed in the area. Each of the new

child thread-blocks will also check the number of points in their areas and recursively launch more blocks when necessary, until each one is assigned a single point. The recursion stopping condition is met when the maximum depth is reached (natively 24 in the CUDA architecture) or when there is a single bead in the cell being processed by the current thread-block. In this quadtree algorithm, each block counts the points assigned to them, and reorders them moving them to its subregion of the array. It works with a minimum number of beads in each cell that can be set as a parameter. If the count of beads is less than the minimum or if the maximum recursion depth is reached, the recursive kernel stops and returns. The actions to be performed by each block are illustrated in figure 6.3. The depth and block launching process is shown in Figures 6.4 to 6.8.

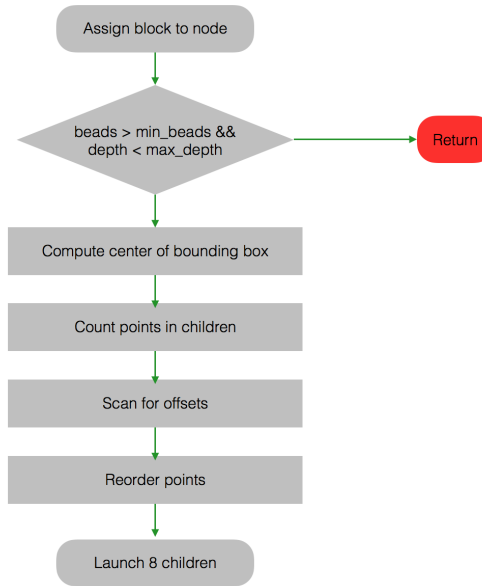


Figure 6.3: Work performed by each block in the parallel Barnes Hut tree construction.

Memory wise, the algorithm uses two buffers of size N that it uses to reorder, count, and swap beads. The final ordered beads should be in buffer 0 and buffer 1 where

they are used as an auxiliary memory spaces to temporarily store partially ordered data. The nodes are handled in a separate array, unlike the previously shown parallel Barnes Hut implementation. This tree approach suits the problem at hand and can be easily modified to work for octrees, as will be shown in subsequent chapters.

Both CPU and GPU Barnes Hut tree implementations are shown in [11, 28] and involve traversal per bead to calculate the accumulated force derived from the interactions. In this work however, it was adapted to work with the original interaction list used in previous SOP models. The traversal is changed in this case, as well as the CPU implementation explained above, because the pair lists dictate what sort of force computation is performed, whether the interaction is attractive or repulsive. As a result, the mass accumulation of the original Barnes Hut tree is not used, but instead the centers of mass are used to approximate distances.

To perform the force calculations with an octree structure and the pair list, an auxiliary tree structure is computed based on walks on the tree. Calculating the forces is a process that takes every pair (i, j) in both the attractive and the repulsive pair lists and calculates the forces derived from the interaction by finding j in the tree, and keeping an auxiliary tree of precomputed distances. For each pair (i, j) finding j on the tree can result in three possible options:

1. While traversing the tree, there is an internal node to which the distance from i has already been calculated, so this approximation is reused to calculate the resulting force of the interacting pair (i, j) .
2. While traversing the tree, there is an internal node found that is determined to be far away enough ($l/D < \theta$), so a distance calculation is performed and stored in the auxiliary tree.
3. Bead j is found in a leaf, meaning that it is close enough to i ($l/D > \theta$ for every

internal node in the path to j) and thus the interaction can not be approximated.

The bead-to-bead distance needs to be calculated and used to compute the derived force.

The recursive layout of the algorithm to calculate the distance between i and j looks like shown below:

```
if (current node == j) then
| calculate distance to i
else if (distance to current node already calculated) then
| return distance
else if (l/D less than  $\theta$ ) then
| calculate distance to this center of mass
| return distance
else
| find the child where j is
| recursively find j in child
end
```

The following series of images below illustrates the process of the parallel Barnes Hut tree being constructed via thread blocks that perform the operations described in this section of the chapter.

Depth = 0



block 0

buffer 1

a	b	c	d	e	f	g	h	i	j	k	l
---	---	---	---	---	---	---	---	---	---	---	---

Figure 6.4: The initial stage in depth 0, buffer 1 will have all the beads in their original positions and block zero will be launched and assigned the complete simulation space.

Depth = 1

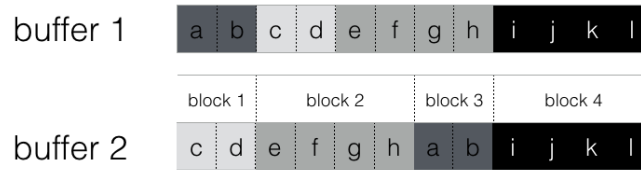
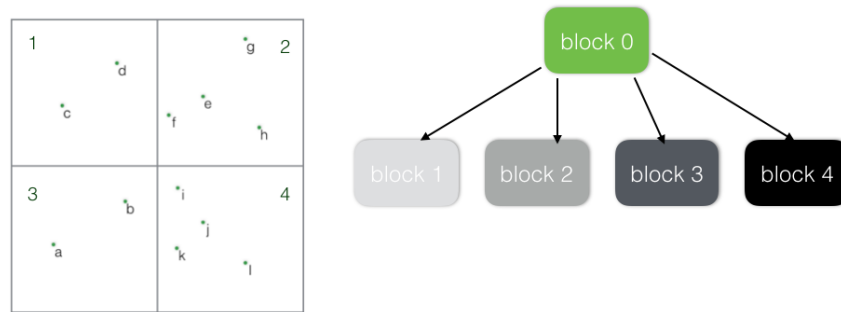


Figure 6.5: The blocks 1 to 4 will count their beads in buffer 1 and copy them in the appropriate reordered location in buffer 2. Then any block with more than one bead in its region will recursively launch four more thread blocks.

Depth = 2

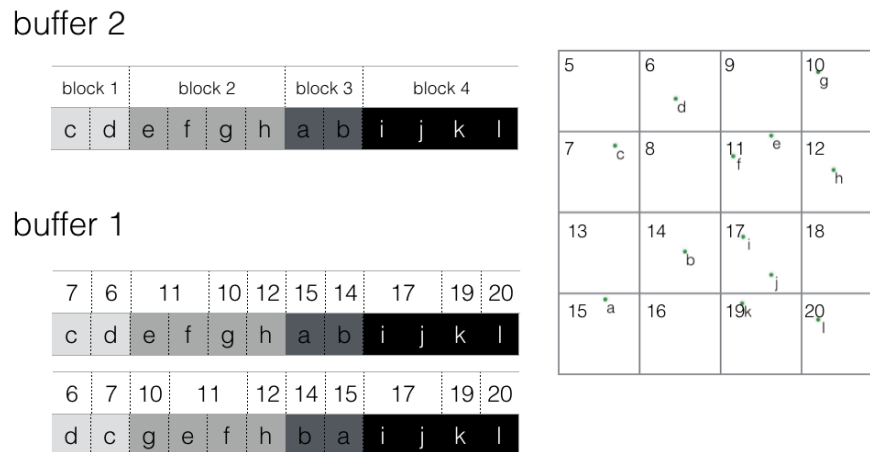


Figure 6.6: When the parent block finished, the beads were left in buffer 2, then at depth 2, and the process is repeated starting in buffer 2 and transferring to buffer 1.

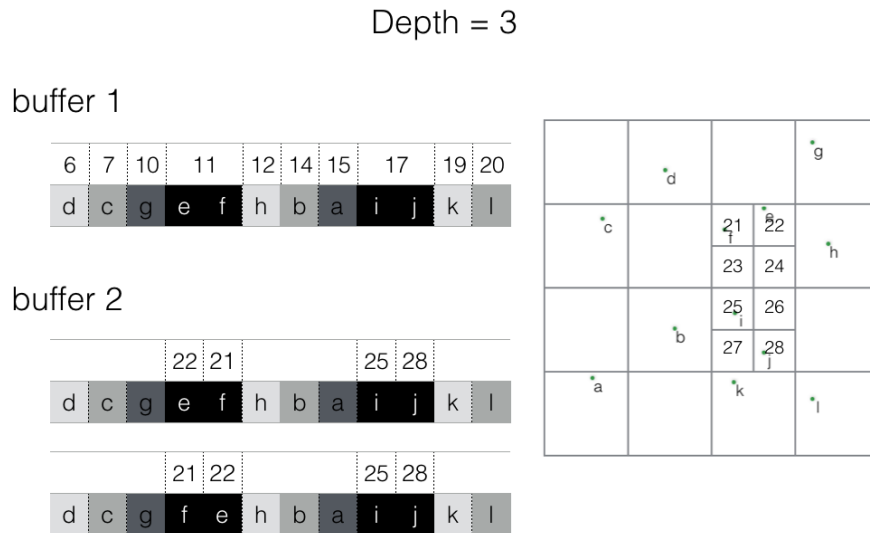


Figure 6.7: At a recursion depth of 3, most beads are already the only bead in their cell, and only thread blocks 11 and 17 will launch four more children blocks each.

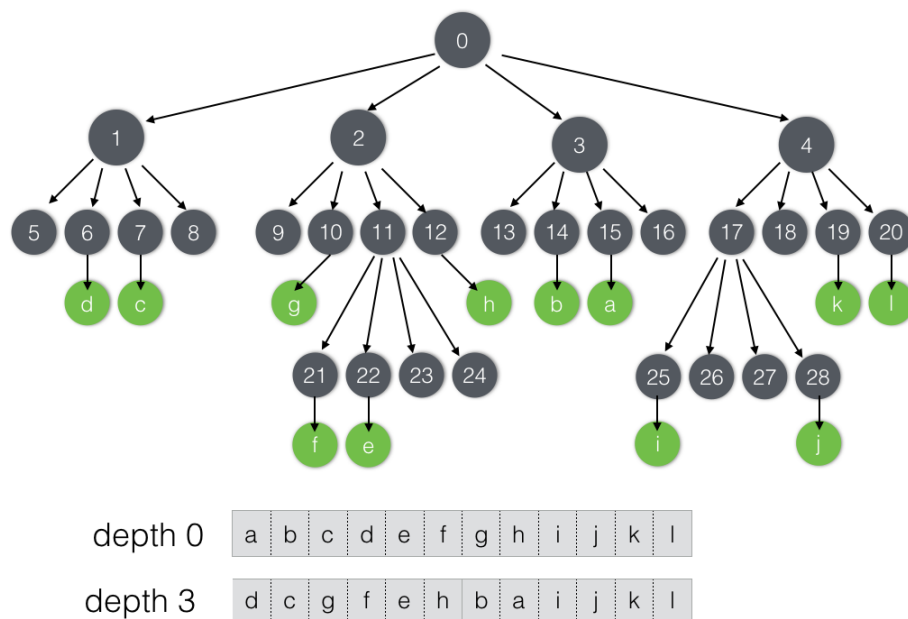


Figure 6.8: The tree constructed is the one showed in the upper portion of this image. The resulting ordering of beads is at the end of depth recursion level 3 the same as if the tree was traversed depth first to list the leaves.

Chapter 7: Performance of Barnes-Hut Algorithm in MD Simulations

7.1 CPU implementation

The construction of the Barnes-Hut tree for the SOP model was implemented as the original paper described for the CPU version [11]. In each tree update, it takes the current positions of each of the beads that form the biomolecule and inserts them into the tree one by one, either recursively creating more nodes or placing it in a currently empty leaf. The traversal part of the algorithm, however, needs to be modified for the SOP model due to the nature of the van der Waals forces and Lennard Jones coefficients. The force between bead i and the bead j in a leaf of the tree can be attractive or repulsive, and both cases are calculated differently (see terms V_{VDW}^{ATT} for the attractive interactions and V_{VDW}^{REP} for the repulsive interactions in Chapter 4, SOP model section). This means that the mass accumulated in a cell can not be used in the approximated calculation because not all the beads in that cell will contribute to the force in the same way. Additionally, the pair lists are still necessary, because they specify what pairs are attractive and repulsive interactions. But the position of the center of mass is useful and can be reutilized for the subset of pairs of the form (i, x) , where x is any other bead in the system different than i . These distances are kept while computing forces within the subset, and for beads x that belong in the same “pruned” branch of the BH tree, previous computations will be used in the force summation, thus saving operation time. The lists of attractive and repulsive forces are determined based on the type of bead that takes part of the interaction, a coefficient, and a cutoff for attractive interactions, and all of this information comes

from the input data. The way in which this list is conceptualized is shown in Figure 7.1, and it represents a particular challenge for the SOP model to work with the Barnes-Hut tree traversal.

i bead	j bead	q	i type	j type
1	3	13.29	1	2
...	...	...	...	...
1	5	25.13	3	1
2	4	21.69	2	2
...	...	...	...	...
2	8	12.42	1	2
3	1	15.65	2	2

Figure 7.1: Data input for the pair list that later splits into attractive and repulsive forces. Pairs are sorted first by the i bead and then by the j bead in the input data.

To take full advantage of the BH tree, an auxiliary tree was created that was updated with distances from a bead i to all other beads in the interaction list. Since the pair lists have blocks of interactions that start with bead i , the tree can be updated with distances and calculate approximations that can be reused in following pairs to be processed. This reduces the amount of floating point operations to be computed, depending on the θ configuration parameter, yet producing a significant increase in memory usage in the CPU version. For every pair, the relevant distances to be computed and used in the force update step are d^2 , d^6 , d^{12} , d_x , d_y and d_z . The distances in each axis represent the corrected positions based on periodic bounding conditions. All of the computations require floating point operations and are stored

in the auxiliary tree, so their approximations can be reused to reduce the number of float point operations. The tree traversal that computes these distances is shown in the pseudocode below:

```

forall pair  $[i,j]$  do
  | traverse tree starting in root
  | if found a calculation in auxiliary tree then
  |   | use calculation as approximation
  |   | add to pair list
  | else if found an internal node then
  |   | if  $l/D < \theta$  then
  |   |   | calculate distance to this internal node
  |   |   | record distance in auxiliary tree
  |   | else
  |   |   | recursively continue looking for  $j$ 
  |   | else if found a leaf then
  |   |   | calculate distance
  |   |   | add to pair list
end

```

The results of this implementation reduced calculated pairs in a constant factor for increasingly longer simulations, as shown in Figure 7.2. The percentage of approximations stayed for all the run cases very close to 60% for $\theta = 0.5$. For varying θ values, the percentage of bead-to-bead calculation and the percentage of approximations are shown in Figure 7.3. Their relationship is inverse and a loss of precision in the simulation is expected as θ increases, causing a tradeoff situation.

This was designed to be recalculated every timestep of an N -Body simulation, although the methods used for MD simulations would use a fixed rate of update defined by every N timesteps. An advantage of the Pair List combined with the BH tree is that it is easy to detect when the beads have moved enough to invalidate the current tree. Whenever one of the beads is close enough to the boundaries of a cell and crosses over, the search of bead j in the current tree will fail to find the correct leaf, thus triggering a recomputation of the tree. This resulted in an adaptive form

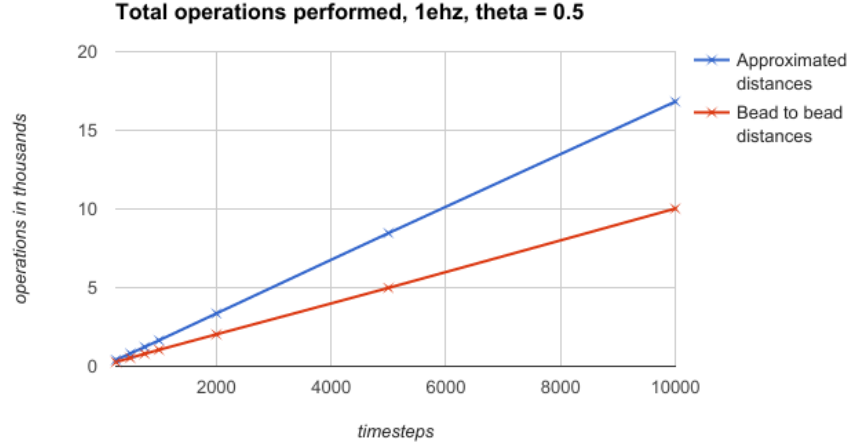


Figure 7.2: The lines show for a value of $\theta = 0.5$ how the calculations were reduced by the usage of the tree.

of tree recalculation that reduced the amount of tree recalculations in comparison to the parameters used to test in previous works [51]. How often this occurs, varies according to how dense the distribution of the beads is; the closer together they are the smaller the space has to be divided to create cells that contain only one bead, and, as a consequence, beads will cross over cell boundaries more often. Figure 7.4 shows the comparison between detection based reconstruction of the tree versus tree reconstruction every N steps, with N fixed as a parameter of the simulation. Although the difference is small due to the complexity of building the tree ($O(\log(n))$), it adds the convenience of having the tree reconstruct whenever the system has changed sufficiently, while still maintaining the accuracy.

Despite the convenience of the tree, there was little improvement observed in the CPU version, even varying the precision parameter as shown in Figure 7.5. There is a relationship between the ratio to which tree branches are approximated instead of calculated and how long it takes to calculate each force update. However, as seen in Table 7.1, the larger portion of the execution time break up is taken by the pair list

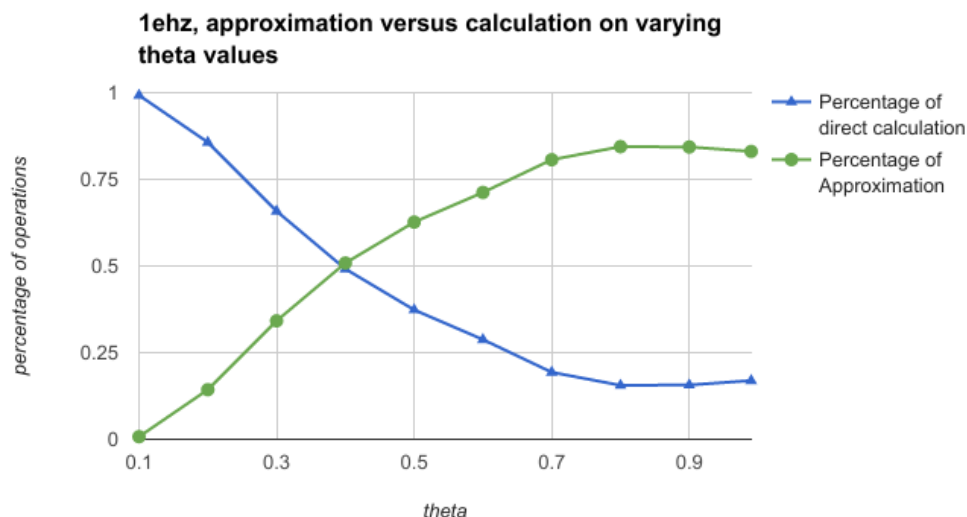


Figure 7.3: Theta determines the tradeoff between a very precise simulation or less computations used to calculate the same system.

	Whole simulation	Tree construction	Pair List construction	Force evaluation
seconds	1.946985	0.000548	1.803739	0.142698
average per timestep	0.001946985	0.000000548	0.001803739	0.000142698
percentage	100.00%	0.03%	92.64%	7.33%

Table 7.1: This table shows the time breakup among the most computationally expensive tasks of the simulation. It is run for $\theta = 1000$, with 1000 timesteps and biomolecule 1ehz (76 beads)

construction, while the tree construction and force evaluation are comparatively fast. This indicates that the problem seems to be memory bound in the CPU implementation. Even with a very fast tree computation and faster force evaluation, building a pair list based on the BH tree results in a memory constraint. Inspecting the memory in the system per timestep and comparing the approximated version to the all-pairs calculation was important to ensure the precision of the approximation. Figure 7.6 shows how the energy in the same system, in two simulations, one with $\theta = 0.0$ and one with $\theta = 0.5$ have a proportional relation. Pearson correlation for the energy in both simulations is about 0.95.

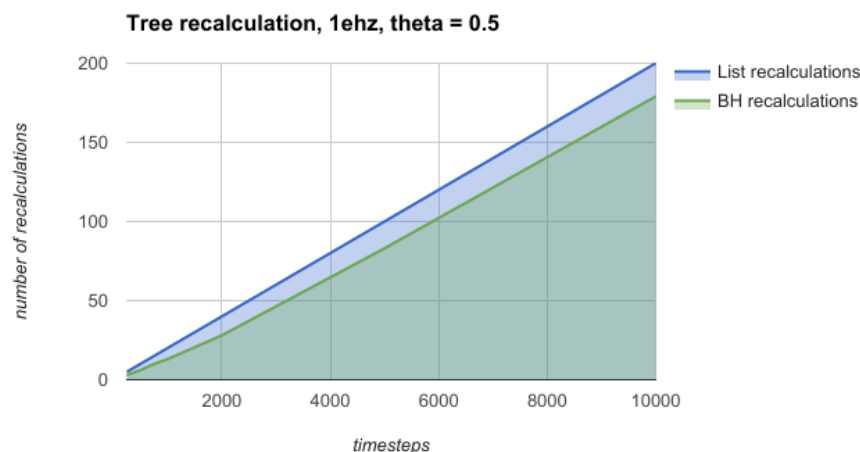


Figure 7.4: A small difference can be noticed in the amount of times the BH tree is calculated in comparison to purely pair based cutoff methods.

7.2 GPU implementation

Identifying the bottlenecks in the CPU version influenced the optimizations applied to the GPU version. It was implemented with dynamic parallelism to build the Barnes-Hut tree, with results twice as fast in the smallest test system. In terms of the pair list, the task was to reduce the memory bottleneck, which was achieved through shared memory. When the beads are first read in, an index of each of the blocks is created for both attractive and repulsive van Der Waals forces. Each block of ordered interaction pairs as shown in figure 7.1, was assigned to a thread block, so that several auxiliary trees could be computed simultaneously and in shared memory, thus reducing the memory bandwidth occupation and using a GPU on chip memory as shown in Figure 7.7. No extended pair list was computed for the GPU case so that global memory usage could be reduced. The traversal of the tree and computation of forces is done in a single step, accumulating the forces of each bead as the pairs are processed in the tree. A kernel that performs this traversal was used to compute the forces used by the attractive van Der Waals and it reduced the computation time by

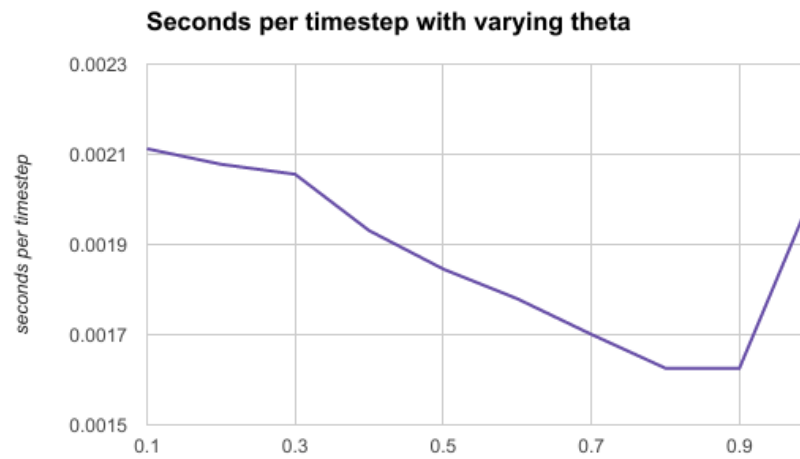


Figure 7.5: The amount of float point operations to be performed does decrease on increasing theta values.

half when compared to the traditional all-pairs simulation.

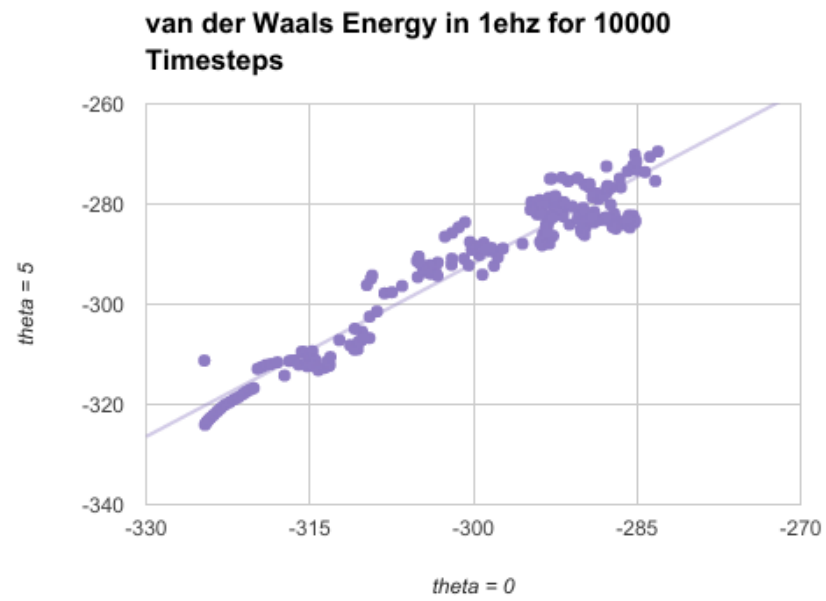


Figure 7.6: The approximation of the simulation will impact the energy but they should remain related variables. Pearson correlation for this experiment is 0.9579909477.

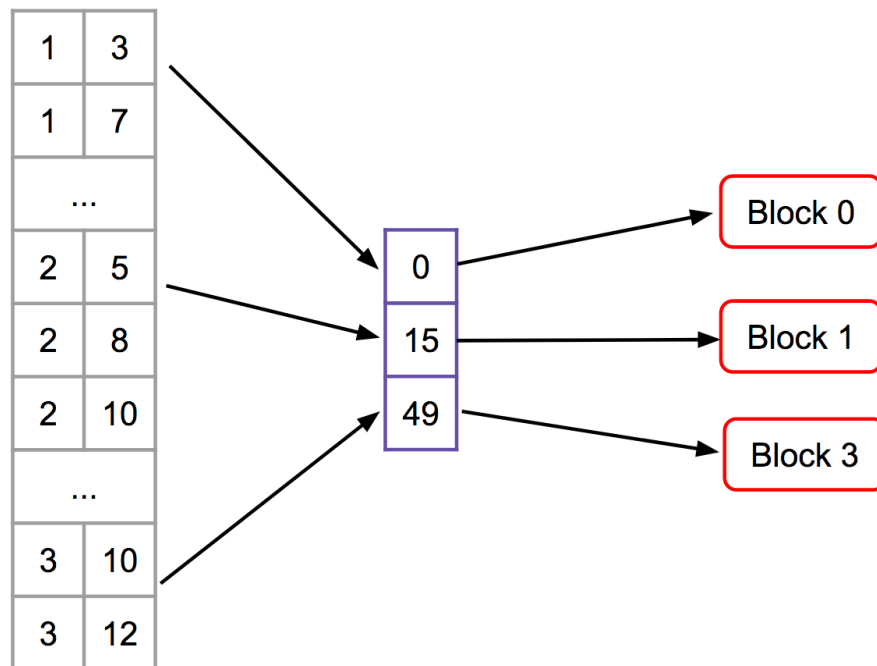


Figure 7.7: The pairs in each of the lists of interaction were divided in groups and indexed, each thread block processes one of these groups of pairs.

Chapter 8: Conclusions and future work

MD simulations continued to improve over time, as has the hardware that they are run in. Nvidia GPUs have advanced and will continue to do so in the foreseeable future, providing every time more sophisticated and capable features for developers and scientists. This work aimed to make use of a new feature implemented after CUDA computing capability 3.5 and combine it with the *N*-Body approach called Barnes-Hut tree of hierarchical space decomposition. It works with the computing of a virtual tree that represents the space partition to efficiently manipulate and perform calculations and searches on points distributed in the simulation space. Despite the extensive use of parallelized Barnes-Hut implementations in astrophysics, the parallel recursive implementation of a Barnes-Hut octree, via the method presented hereby was not found in previous literature. Neither has the application of tree methods in MD simulations.

With this work and the attempt of utilizing the aforementioned methods, we can conclude that Barnes-Hut is suited for simpler *N*-Body simulation models where the force calculations are not as divergent as they can be in the SOP coarse grained model. However, it is still possible to achieve such implementation with promising results in terms of accuracy, configurability, adaptability and speedup that require further and extensive testing and optimization. It was not without obstacles that these conclusions were reached, the main challenges were related to dealing with the CUDA architecture, particularly the specificity of devices necessary to test, and the intricacy of the pre existing codebase as well as the newly added code.

Future work for the SOP Model-Barnes Hut implementation is abundant. A natural next step is benchmarking against other common cutoff methods such as the Verlet

neighbor list, Cell list, Hybrid list, and parallel versions of all of them. Additionally, reviewing and implementing the newer version of the Barnes-Hut modified tree [55] based on tree walks that are shared among several leaves in the tree, thus creating path prefixes that can be taken advantage of. Rethinking the data structures that are currently used as well as memory patterns to create coalescing memory access could highly impact the current results, as could the interleaving of memory loads and computations (streaming).

Bibliography

- [1] Herman H. Goldstine. *The Computer from Pascal to von Neumann New Ed Edition*. Princeton University Press; New Ed edition (1980)
- [2] Martin Davis. *The Universal Computer: The Road from Leibniz to Turing*. W. W. Norton Company; 1st edition (October 2000).
- [3] Transistors to Transformations. From Sand to Circuits - How Intel makes chips.
[Consulted on 03/2017]
- [4] CUDA C Programming Guide v8.0. January 2017.[Consulted on 03/2017]
- [5] Barry Wilkinson, Michael Allen. *Parallel Programming. Techniques and applications using networked workstations and parallel computers*. Prentice Hall, New Jersey 1999.
- [6] van Gunsteren, Wilfred F. and Berendsen, Herman J. C. *Computer Simulation of Molecular Dynamics: Methodology, Applications, and Perspectives in Chemistry*. Angewandte Chemie International Edition in English, 1990.
- [7] H.M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T.N. Bhat, H. Weissig, I.N. Shindyalov, P.E. Bourne. *The Protein Data Bank Nucleic Acids Research* 2000.
- [8] L. Verlet. *Computer "Experiments" on classical fluids i. thermodynamical properties of Lennard-Jones molecules..* Physical Review, 1967
- [9] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, USA, 1989.

- [10] Andrew J. Proctor, Cody A. Stevens, Samuel S. Cho *GPU-Optimized Hybrid Neighbor/Cell List Algorithm for Coarse-Grained MD Simulations of Protein and RNA Folding and Assembly* Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine, 2013.
- [11] Josh Barnes and Piet Hut. *A hierarchical $O(N \log N)$ force calculation algorithm*. Nature, December 1986.
- [12] David B. Kirk, Wen-Mei W. Hwu. *Programming Massively Parallel Processors* Third Edition, Morgan Kaufmann, 2017.
- [13] T. J. Bergin, ed. *50 Years of Army Computing: From ENIAC to MSRC*. 2000. url: http://www.arl.army.mil/www/pages/shared/documents/50_years_of_army_computing.pdf. [Consulted on 03/2017]
- [14] Gordon E. Moore. *Cramming More Components onto Integrated Circuits*. Proceedings of the IEEE, 1998.
- [15] ASCI Red. Top 500. url: <https://www.top500.org/>. [Consulted on 03/2017]
- [16] Jack Dongarra. *Report on the Sunway TaihuLight System* Oak Ridge National Laboratory. June 2016.
- [17] John M. Shalf, Robert Leland. *Computing beyond Moore's Law*. IEEE Computer Society, 2015.
- [18] Karl Rupp. url: <https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/> [Consulted on 03/2017]
- [19] Michael J. Flynn. *Very High-speed Computing Systems*. Proceedings of the IEEE, December 1966.

- [20] Nvidia Whitepaper. *NVIDIA's Next Generation Cuda Compute Architecture Fermi*. V1.1
- [21] Khronos Group. *The open standard for parallel programming of heterogeneous systems*. url: <https://www.khronos.org/opencv/>. [Consulted on 03/2017]
- [22] NVIDIA. *NVIDIA accelerated computing*. url: <https://developer.nvidia.com/cuda-toolkit> [Consulted on 03/2017]
- [23] OpenACC. *The OpenACC Application Programming Interface*. url: http://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5_0.pdf
- [24] OpenmP. *http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf*. url: `OpenMPApplicationProgrammingInterface`
- [25] Edgar Gabriel and Graham E. Fagg and George Bosilca and Thara Angskun and Jack J. Dongarra et al. *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*. Proceedings, 11th European PVM/MPI Users' Group Meeting. Budapest, Hungary, 2004.
- [26] Fung, James and Mann, Steve. *OpenVIDIA: Parallel GPU Computer Vision*. Proceedings of the 13th Annual ACM International Conference on Multimedia. New York, 2005.
- [27] Andrew Adinetz *Adaptive Parallel Computation with CUDA Dynamic Parallelism*. url: <https://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/> [Consulted on 03/2017].
- [28] Wen-Mei W. Hwu. *GPU Computing Gems Emerald Edition (Applications of GPU Computing Series)*. Morgan Kaufmann, 2011.

- [29] Bruce Alberts, Alexander Johnson, Julian Lewis, David Morgan, Martin Raff, Keith Roberts, Peter Walter. *Molecular Biology of the Cell*. Garland Science, 6th edition, 2014.
- [30] Alan Fersht. *Structure and Mechanism in Protein Science: A Guide to Enzyme Catalysis and Protein Folding*. W. H. Freeman and Company, 1999.
- [31] C. B. Anfinsen. *Principles that govern the folding of protein chains*. Science, 1973.
- [32] J. A. Doudna and T. R. Cech. *The chemical repertoire of natural ribozymes*. Nature, 2002.
- [33] Leuchter, Jessica D. and Green, Adam T. and Gilyard, Julian and Rambarat, Cecilia G. and Cho, Samuel S. *Coarse-Grained and Atomistic MD Simulations of RNA and DNA Folding*. Israel Journal of Chemistry. 2014.
- [34] K. A. Dill et al. *Principles of protein folding—a perspective from simple exact models*. Protein Sci 4.4, 1995.
- [35] C. M. Dobson. *Protein misfolding, evolution and disease*. Trends in biochemical sciences 24.9, 1999.
- [36] W. A. Eaton et al. *Fast kinetics and mechanisms in protein folding*. Annual Review of Biophysics and Biomolecular Structure 29, 2000.
- [37] A. R. Fersht and S. Sato. *Phi-Value analysis and the nature of protein-folding transition states*. Proceedings of the National Academy of Sciences 101.21, 2004.
- [38] P. G. Wolynes, J. N. Onuchic, and D. Thirumalai. *Navigating the folding routes*. Science 267.5204, 1995.

- [39] B. Kuhlman et al. *Design of a novel globular protein fold with atomic-level accuracy*. Science 302.5649, 2003.
- [40] C. Hyeon, R. I. Dima, and D. Thirumalai. *Pathways and kinetic barriers in mechanical unfolding and refolding of RNA and proteins*. Structure 14.11, 2006.
- [41] D. L. Pincus et al. *Minimal models for proteins and RNA from folding to function*. Prog Mol Biol Transl Sci 84, 2008.
- [42] R. D. Hills and C. L. Brooks. *Insights from Coarse-Grained Go Models for Protein Folding and Dynamics*. International Journal of Molecular Sciences, 10.3, 2009.
- [43] J. E. Stone et al. *GPU-accelerated molecular modeling coming of age*. Journal of Molecular Graphics and Modeling 29.2, 2010.
- [44] J. A. Anderson, C. D. Lorenz, and A. Travesset. *General purpose molecular dynamics simulations fully implemented on graphics processing units*. Journal of Computational Physics 227.10, 2008.
- [45] S. Plimpton and B. Hendrickson. *A new parallel method for molecular dynamics simulation of macromolecular systems*. Journal of Computational Chemistry 17.3, 1996.
- [46] C. Clementi, H. Nymeyer, and J. N. Onuchic. *Topological and energetic factors: what determines the structural details of the transition state ensemble and "en-route" intermediates for protein folding? An investigation for small globular proteins*. J Mol Biol 298.5, 2000.
- [47] M. Guthold and S. S. Cho. *Fibrinogen Unfolding Mechanisms Are Not Too Much of a Stretch*. Structure 19.11, 2011.

- [48] A. Zhmurov et al. *Sop-GPU: Accelerating biomolecular simulations in the centisecond timescale using graphics processors*. Proteins: Structure, Function, and Bioinformatics 78.14, 2010.
- [49] A. Zhmurov et al. *Mechanism of Fibrin(ogen) Forced Unfolding*. Structure 19.11, 2011.
- [50] I. Newton. Principia: Mathematical Principles of Natural Philosophy, 1687.
- [51] Andrew J. Proctor, Tyson J. Lipscomb, Anqi Zou, Joshua A. Anderson, Samuel S. Cho. *Performance Analyses of a Parallel Verlet Neighbor List Algorithm for GPU-Optimized MD Simulations*. International Conference on BioMedical Computing, 2012.
- [52] Andrew J. Proctor, Cody A. Stevens, Samuel S. Cho. *GPU-Optimized Hybrid Neighbor/Cell List Algorithm for Coarse-Grained MD Simulations of Protein and RNA Folding and Assembly*. ACM-BCB, 2013.
- [53] W. Mattson and B. M. Rice. *Near-neighbor calculations using a modified cell-linked list method*. Computer Physics Communications, 1999.
- [54] D. B. Wang, F. B. Hsiao, C. H. Chuang, and L. Y. C. *Algorithm optimization in molecular dynamics simulation*. Computer Physics Communications, 2007.
- [55] Barnes, Joshua E. *A Modified Tree Code: Don'T Laugh; It Runs*. J. Comput. Phys. San Diego, California 1990.

Appendix A: CPU Barnes Hut Tree Construction

```
int get_next_tree_index(){
    int old = next_tree_index + 1;
    next_tree_index += 8; //creates space for 8 new cells
    return old;
}

coord * find_subtree_coord(double width, coord *center, int bead_index){
    coord * subtree = new coord();

    subtree->x = unc_pos[bead_index].x < center->x ? 0 : 1;
    subtree->y = unc_pos[bead_index].y < center->y ? 0 : 1;
    subtree->z = unc_pos[bead_index].z < center->z ? 0 : 1;

    return subtree;
}

int get_subtree_index(coord *sub_block, int base_index){
    int tree_child = base_index + (sub_block->x * 4) +
                      (sub_block->y * 2) + sub_block->z;
    return tree_child;
}
```

```

void update_center_mass(int current_node, int bead_index){
    octet_center_mass[current_node].x += unc_pos[bead_index].x;
    octet_center_mass[current_node].y += unc_pos[bead_index].y;
    octet_center_mass[current_node].z += unc_pos[bead_index].z;
}

coord * get_udpated_center(coord *boxCenter, coord* subtree, double width){
    double quarter = width/4.0;
    coord * newCenter = new coord();

    newCenter->x = subtree->x == 1.0 ? boxCenter->x + quarter :
                    boxCenter->x - quarter;
    newCenter->y = subtree->y == 1.0 ? boxCenter->y + quarter :
                    boxCenter->y - quarter;
    newCenter->z = subtree->z == 1.0 ? boxCenter->z + quarter :
                    boxCenter->z - quarter;

    return newCenter;
}

void insert_bead_bhtree(int tree_index, int bead_index,
coord *boxCenter, double width){
    if (indices_bhtree[tree_index] == empty_cell){
        /* Inseart bead with negative index */
        indices_bhtree[tree_index] = -bead_index;
        octet_count_bhtree[tree_index] = 1;
    }
}

```

```

    octet_center_mass[tree_index].x = boxCenter->x;
    octet_center_mass[tree_index].y = boxCenter->y;
    octet_center_mass[tree_index].z = boxCenter->z;

} else if (indices_bhtree[tree_index] < 0) {
    /* Index is negative if there's a node in the cell,
    new positive index will be added for new cell */

    int a_bead_index = -indices_bhtree[tree_index];
    indices_bhtree[tree_index] = get_next_tree_index();
    octet_count_bhtree[tree_index] = 2;
    update_center_mass(tree_index, bead_index);

    coord *subtreeA = find_subtree_coord(width, boxCenter, a_bead_index);
    coord *subtreeB = find_subtree_coord(width, boxCenter, bead_index);
    int subtree_indexA = get_subtree_index(subtreeA,
                                           indices_bhtree[tree_index]);
    int subtree_indexB = get_subtree_index(subtreeB,
                                           indices_bhtree[tree_index]);
    coord *newCenterA = get_udpated_center(boxCenter, subtreeA, width);
    coord *newCenterB = get_udpated_center(boxCenter, subtreeB, width);
    insert_bead_bhtree(subtree_indexA, a_bead_index, newCenterA, width/2.0);
    insert_bead_bhtree(subtree_indexB, bead_index, newCenterB, width/2.0);
    delete newCenterA; delete newCenterB;

} else {

```

```

/* Positive index, internal node found */
octet_count_bhtree[tree_index] = octet_count_bhtree[tree_index]+1;
update_center_mass(tree_index, bead_index);

coord *subtree = find_subtree_coord(width, boxCenter, bead_index);
int subtree_index = get_subtree_index(subtree,
                                     indices_bhtree[tree_index]);
coord *newCenter = get_updated_center(boxCenter, subtree, width);
insert_bead_bhtree(subtree_index, bead_index, newCenter, width/2.0);
delete newCenter;
}
}

double set_initial_width(){
    double width = 0.0;
    double x,y,z;
    for (int i = 0; i <= nbead; i++) {
        x = fabs(unc_pos[i].x);
        y = fabs(unc_pos[i].y);
        z = fabs(unc_pos[i].z);
        width = x > width ? x : width;
        width = y > width ? y : width;
        width = z > width ? z : width;
    }
    rootWidth = ceil(2*width);
}

```

```

void build_bh_tree(){
    reinserted++;
    /* reset tree */
    std::fill_n(indices_bhtree, 16*nbead, empty_cell);
    next_tree_index = 0;
    int tree_index = 0;
    set_initial_width();
    coord *boxCenter = new coord();
    boxCenter->x = 0.0;
    boxCenter->y = 0.0;
    boxCenter->z = 0.0;

    for (int i = 1; i <= nbead; i++) {
        insert_bead_bhtree(tree_index, i, boxCenter, rootWidth);
    }

    /* Tree is marked as rebuilt */
    rebuild = 0;
    delete boxCenter;
}

```

Appendix B: GPU Barnes Hut Tree Construction

```
__device__ bool check_points(Octree_node &node, int *points1,
int *points2, int num_points, Parameters params){
    if(params.depth >= params.max_depth || num_points <= params.min_points){
        if(params.point_selector == 1){
            int it = node.points_begin(); int end = node.points_end();
            for(it += threadIdx.x; it < end; it += blockDim.x){
                points1[it] = points2[it];
            }
        }
        return true;
    }
    return false;
}
```

```
__device__ void count_points(const int *in_points, FLOAT3* dev_unc_pos,
int *smem, int range_begin, int range_end, FLOAT3 center){
    if(threadIdx.x < 8) smem[threadIdx.x] = 0;
    __syncthreads();

    for(int iter=range_begin+threadIdx.x; iter<range_end; iter+=blockDim.x){
        FLOAT3 p = dev_unc_pos[in_points[iter]];
    }
}
```

```

    int x = p.x < center.x ? 0 : 1;
    int y = p.y < center.y ? 0 : 1;
    int z = p.z < center.z ? 0 : 1;

    int i = x*4 + y*2 + z;

    atomicAdd(&smem[i], 1);
}
__syncthreads();
}

__device__ void scan_offsets(int node_points_begin, int* smem){
    int *smem2 = &smem[8];
    if(threadIdx.x == 0){
        for(int i = 0; i < 8; i++){
            smem2[i] = i == 0 ? 0 : smem2[i-1] + smem[i-1];
        }
        for (int i = 0; i < 8; i++){
            smem2[i] += node_points_begin;
        }
    }
    __syncthreads();
}

__device__ void reorder_points(int *out_points, const int *in_points,

```

```

FLOAT3 *dev_unc_pos, int *smem, int range_begin, int range_end,
FLOAT3 center){
    int *smem2 = &smem[8];

    for(int iter = range_begin+threadIdx.x; iter<range_end; iter+=blockDim.x){
        FLOAT3 p = dev_unc_pos[in_points[iter]];

        int x = p.x < center.x ? 0 : 1;
        int y = p.y < center.y ? 0 : 1;
        int z = p.z < center.z ? 0 : 1;

        int i = x*4 + y*2 + z;

        int dest = atomicAdd(&smem2[i], 1);
        out_points[dest] = in_points[iter];
    }

    __syncthreads();
}

__device__ void prepare_children(Octree_node *children,
Octree_node &node, int *smem){

    const FLOAT3 center = node.center();
    float half = node.width() / 2.0f;
    float quarter = half / 2.0f;

```

```

smem[7] = node.points_begin();

for(int i = 0; i < 8; i++){
    int xf, yf, zf;
    zf = i % 2;
    if (zf == 0) zf = -1;

    yf = (i / 2) % 2;
    if (yf == 0) yf = -1;

    xf = i / 4;
    if (xf == 0) xf = -1;

    children[i].set_center(center.x + quarter * xf,
                           center.y + quarter * yf,
                           center.z + quarter * zf);
    children[i].set_width(half);
    children[i].set_range(smem[7+i], smem[8+i]);
}
}

__global__ void build_octree_kernel(Octree_node *nodes,
Octree_node *all_nodes, int *num_nodes,
int *points1, int *points2, FLOAT3 *dev_unc_pos, Parameters params){
__shared__ int smem[16];

```

```

Octree_node &node = nodes[blockIdx.x];
int num_points = node.num_points();

bool exit = check_points(node, points1, points2, num_points, params);
if(exit) return;

float3 center = node.center();

int range_begin = node.points_begin();
int range_end = node.points_end();

const int* in_points = params.point_selector == 0 ? points1 : points2;
int *out_points = params.point_selector == 0 ? points2 : points1;

count_points(in_points, dev_unc_pos, smem, range_begin, range_end, center);

scan_offsets(node.points_begin(), smem);

reorder_points(out_points, in_points, dev_unc_pos, smem,
               range_begin, range_end, center);

if(threadIdx.x == blockDim.x-1){
    int next_node = atomicAdd(&num_nodes, 8);
    node.set_children(next_node);
    Octree_node *children = &all_nodes[next_node];
    prepare_children(children, node, smem);
}

```

```

        build_octree_kernel<<<8, blockDim.x, 16*sizeof(int)>>>
            (children, all_nodes, num_nodes, points1, points2,
             dev_unc_pos, Parameters(params, true));
    }
}

__global__ void init_nodes(int *points, int num_points){
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    for(int iter = threadIdx.x; iter<num_points; iter+=blockDim.x){
        points[iter] = i;
    }
}

__global__ void calculate_centers_mass(Octree_node *all_nodes,
int *num_nodes, int *points, FLOAT3 *dev_unc_pos){
    FLOAT x = 0.0, y = 0.0, z = 0.0;
    Octree_node node;
    for(int iter = threadIdx.x; iter<num_nodes[0]; iter+=blockDim.x){
        node = all_nodes[iter];

        int begin = node.points_begin(); int end = node.points_end();
        for (int i = begin; i < end; i++) {
            x += dev_unc_pos[points[i]].x;
            y += dev_unc_pos[points[i]].y;
            z += dev_unc_pos[points[i]].z;
        }
    }
}

```

```

        x /= node.num_points();
        y /= node.num_points();
        z /= node.num_points();

        node.set_center_mass(x/node.num_points(),
                             y/node.num_points(),
                             z/node.num_points());
    }
}

void build_bh_tree(){
    dim3 threads(BLOCK_DIM, 1, 1);
    dim3 grid((int) ceil((nbead + 1.0) / (float) threads.x), 1, 1);
    FLOAT *dev_result; FLOAT result[BLOCK_DIM];
    cudaMalloc((void **)&dev_result, BLOCK_DIM*sizeof(FLOAT));
    maxCoord<<<grid, threads, BLOCK_DIM*sizeof(FLOAT)>>>
        (dev_unc_pos, dev_result, nbead);
    cudaMemcpy(result, dev_result, BLOCK_DIM*sizeof(FLOAT),
               cudaMemcpyDeviceToHost);

    Octree_node root;
    root.set_range(0, nbead);
    root.set_width(2*ceil(result[0]));
    cudaMemcpy(all_nodes, &root, sizeof(Octree_node), cudaMemcpyHostToDevice);

    init_nodes<<<grid, threads>>>(dev_buffer1, nbead);

```

```

Parameters params(nbead);
num_nodes_host[0] = 1;
cudaMemcpy(num_nodes, num_nodes_host, sizeof(int),
           cudaMemcpyHostToDevice);
build_octree_kernel<<<1, BLOCK_DIM, shared_mem>>>(all_nodes,
           all_nodes, num_nodes, dev_buffer1, dev_buffer2, dev_unc_pos, params);

calculate_centers_mass<<<grid, threads>>>(all_nodes,
           num_nodes, dev_buffer1, dev_unc_pos);
cudaMemcpy(num_nodes_host, num_nodes, sizeof(int), cudaMemcpyDeviceToHost);
}

```

Curriculum VITAE

PERSONAL DATA

NAME: Melisa Carranza Zúñiga
PLACE OF BIRTH: San José, Costa Rica
EMAIL: melisacz@gmail.com

WORK EXPERIENCE

Current AUGUST 2016	Research Assistant at WAKE FOREST UNIVERSITY, North Carolina <i>Professor Jennifer Erway</i> Provide assistance and coding in Linear Algebra projects. NSF funded research.
SUMMER 2016	Software Engineering Intern at INMAR, <i>Promotions Network</i> Completed a summer project from the Software Engineering team in Digital Promotions.
AUGUST 2016	Teaching Assistant at WAKE FOREST UNIVERSITY, North Carolina <i>Graduate School of Arts and Sciences</i> Help professors grade and assist students with course projects.
2013-2015	Mobile Developer at KONRAD GROUP, San José <i>Digital production</i> Participation in several iOS, Web, and Android client and internal projects.

EDUCATION

2015-2017	Wake Forest University, North Carolina Master Degree in Computer Science. GPA 3.9 Thesis track, working in GPU Optimized Molecular Dynamics Simulations.
2014	Instituto Tecnológico de Costa Rica Master Degree in Computer Science Not finished
2010-2014	Instituto Tecnológico de Costa Rica Bachelor Degree in Computing Engineering
2005-2010	Liceo Experimental Bilingüe José Figueres Ferrer, Costa Rica, General Certificate of Secondary Education

SCHOLARSHIPS

2010-2014	Scholarship for academic excellence and admission scores Instituto Tecnológico de Costa Rica
2015	Scholarship for the Southeast Women in Computing Conference. Atlanta, Georgia.
2015	Scholarship for the Grad Cohort Workshop by the Computing Research Association. San Diego, California.
2016	Scholarship the Grace Hopper Conference by the Anita Borg Institute. Houston, Texas.

LANGUAGES

SPANISH: Mothertongue
ENGLISH: Fluent
FRENCH: Basic Knowledge

COMPUTER SKILLS

Programming languages: C, C#, C++, Python, Perl, Java, Swift, SQL and NoSQL
Operating Systems: GNU-Linux, Windows, Windows Server, MacOS, Android, iOS

INTERESTS AND ACTIVITIES

- Quantum computing, internet of things, augmented reality, big data, artificial intelligence, mobile development, web technologies, information retrieval, machine learning, computer vision, high performance computing, interdisciplinary research, bioinformatics, operating systems, social work.
- Member of Upsilon Pi Epsilon, International Honor Society for the Computing and Information Disciplines.
- Member of Association of Computing Machinery (ACM).