DIGITALCOMMONS
@WAYNESTATE

Wayne State University

Wayne State University Dissertations

1-1-2015

Efficient Algorithms And Optimizations For Scientific Computing On Many-Core Processors

Kamel Rushaidat
Wayne State University,

Follow this and additional works at: https://digitalcommons.wayne.edu/oa_dissertations

 Part of the Computer Sciences Commons

Recommended Citation
Rushaidat, Kamel, "Efficient Algorithms And Optimizations For Scientific Computing On Many-Core Processors" (2015). *Wayne State University Dissertations*. 1408.
https://digitalcommons.wayne.edu/oa_dissertations/1408

This Open Access Dissertation is brought to you for free and open access by DigitalCommons@WayneState. It has been accepted for inclusion in Wayne State University Dissertations by an authorized administrator of DigitalCommons@WayneState.

EFFICIENT ALGORITHMS AND OPTIMIZATIONS FOR SCIENTIFIC COMPUTING ON MANY-CORE PROCESSORS

by

KAMEL RUSHAIDAT

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2015

MAJOR: COMPUTER SCIENCE

Approved By:

_____

Advisor Date

_____

_____

_____

_____

© COPYRIGHT BY

KAMEL RUSHAIDAT

2015

All Rights Reserved

DEDICATION

I dedicate this work to my parents, Ibrahim Rushaidat and Samar Al-Ajouz. To my wife, Areej, who helped through all hardship and provided me with inspiration and strength. Without her, I could not do this work. And to my kids, Naser and Judy, who have given me the motivation to finish this work.

ACKNOWLEDGMENTS

I would start with thanking God almighty for giving me the chance and strength to do my Ph.D. and for all the blessings that happened to me in my life. Also, I want to thank my advisor, Dr. Loren Schwiebert for all the inspiration and guidance during this trip. He was always there for me to provide mentorship through all these years.

Finally, I would like to thank Dr. Jeffrey Potoff, Dr. Robert Reynolds, Dr. Daniel Grosu, and Dr. Zhi-Feng Huang for serving on my dissertation defense committee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

This chapter will introduce coprocessors and their role in scientific research, focusing on GPUs as they are the main coprocessors used in this work. In addition, this chapter will give an introduction to the contributions presented in this work.

1.1 PC Coprocessors

For the last 40 years, microchip manufacturers have been developing computer processors that are designed to offload intensive calculations from the main processor to accelerate the system's performance. Those processors are called coprocessors or sometimes accelerators. Coprocessors can be used to help process large floating point arithmetic, encryption, signal processing, and graphics. Many of the early coprocessors were designed to accelerate floating point tasks, such as the Intel 8087 coprocessor, and the Motorola 68881/68882 coprocessors [83].

The high competition in the gaming and movie industries resulted in the introduction of high performance graphics cards that provide superior processing capabilities while being affordable at the same time. Graphics processors were considered as coprocessors for generating visual output; however, with all these processing capabilities, researchers have been interested in using them in applications that were infeasible in the past because of their long execution times and the unavailability of inexpensive supercomputers.

To render movies and game scenes, pixels are drawn in parallel by creating a multithreaded program that uses each thread to render different pixels [1]. This parallel architecture was the foundation for using the graphics processors for more general applications or what is commonly called GPGPU (General-Purpose Graphics Processing Unit) programming [1]. Along with the introduction of GPGPU programming came the development of specialized programming languages and APIs that provide a clear and flexible framework to write programs that run on graphics processors such as CUDA (Compute Unified Device Architecture), which was created by NVIDIA [2].

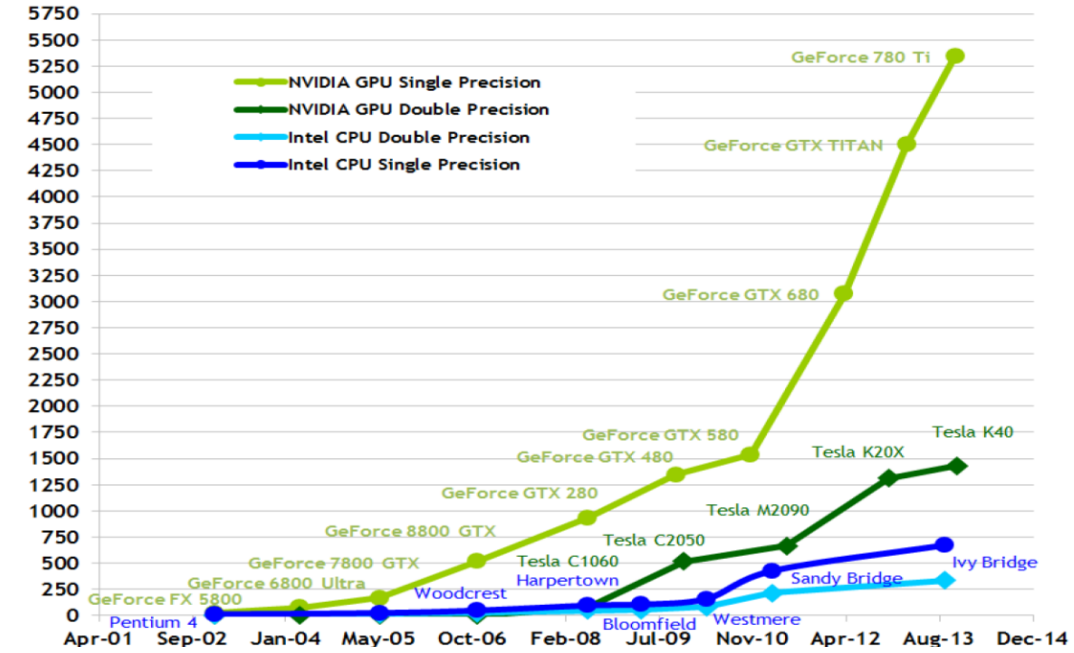


Figure 1: Floating point comparison between the GPU and CPU [6]

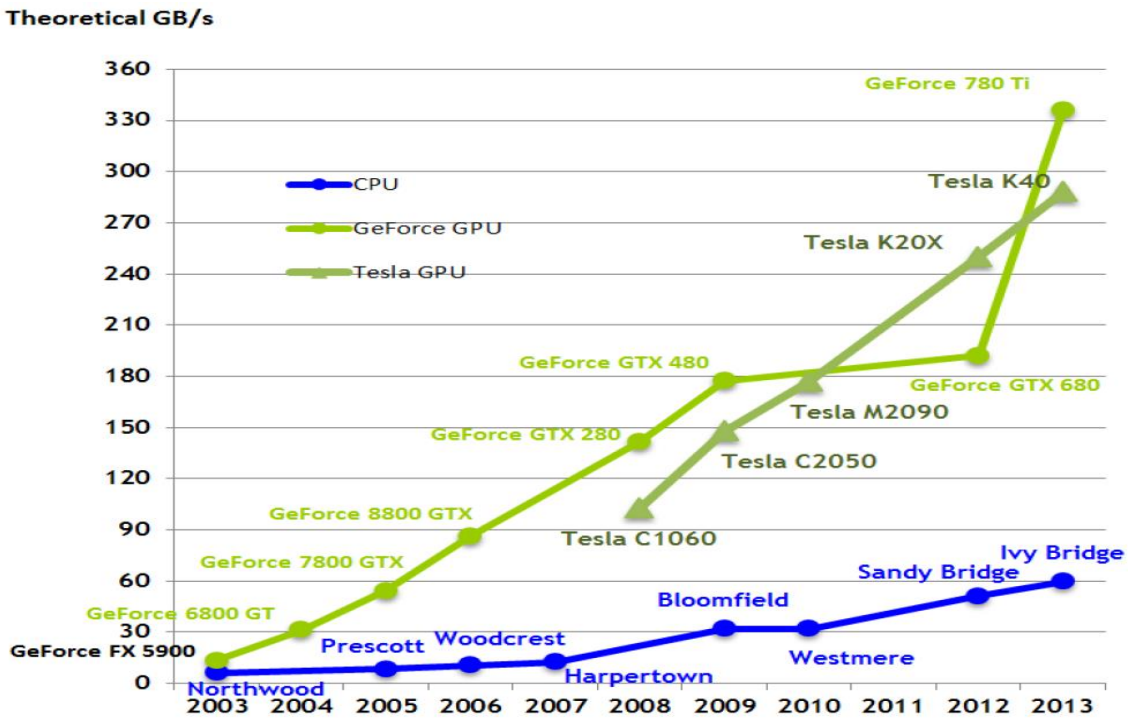


Figure 2: GPU and CPU memory bandwidth historical comparison [6]

GPUs offer unprecedented performance and they are designed to have high throughput. The GPU performance is rapidly increasing compared to CPU performance. Figure 1 shows a historical comparison between different types of CPUs and NVIDIA GPUs in terms of floating point operations per second (FLOPS/s) [6]. In addition, the memory bandwidth of GPUs is also increasing rapidly compared to CPUs as depicted in Figure 2 [6].

GPUs are now becoming a preferred choice to accelerate simulations in many fields of science as they are available and cheap relative to other types of high-performance computing options. GPUs have hundreds or thousands of processing cores compared to only a few in most CPUs, and this is why GPUs have high computational throughput [2]. Many GPU programming research projects have been conducted in different science fields [78]. GPUs are developing very quickly, as it can be noticed from Figures 1 and 2, and in the near future more features will be added to them to help in producing more energy efficient programs and to ease the conversion from sequential codes to parallel codes [107].

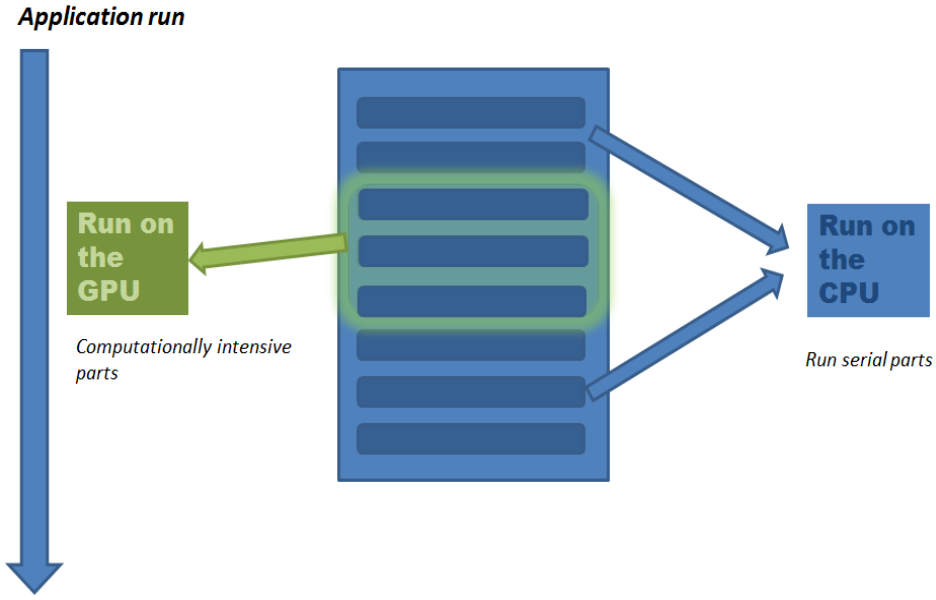


Figure 3: Program word division between the CPU and the GPU

Figure 4: High performance GPUs NVIDIA's Tesla (left) and AMD's FireStream (right) [77]

GPUs offer high performance if there is sufficient parallelism for them to be used to process the most computationally intensive portions of the application. CPU code is generally going to handle other portions of the application, such as I/O and program flow control because GPUs do not have direct access to I/O devices, and the CPU has more complicated cache management and control prediction than GPUs. Figure 3 shows how an application can be divided between the CPU and GPU.

While there are many hardware manufacturers for GPUs, the main two GPU manufacturers are NVIDIA and AMD. GPUs manufactured by those two companies are used now in all kinds of computers, from smartphones all the way up to supercomputers. Examples of supercomputers that are using GPUs are Titan at Oak Ridge National Laboratory, which has 18,688 NVIDIA K20X GPUs providing a theoretical peak performance of 27 petaflops [3]. Both companies produce high end GPUs for scientific applications. Figure 4 shows an Nvidia Tesla and an AMD FireStream GPUs, where both GPUs are designed to run high demanding scientific applications.

Early GPUs used programming languages such as OpenGL and Microsoft's DirectX [1]. However, many GPU programming languages were hard to learn and use, and lacked the representation of many operations that are needed, such as arithmetic operations. Modern GPUs are being programmed mainly by two programming frameworks, the open cross-platform OpenCL [4], and Compute Unified Device Architecture (CUDA) from NVIDIA [5].

In 2012, Intel introduced the Xeon Phi coprocessor. This coprocessor is an SMP (Symmetric multiprocessing) on a chip [83] that runs Linux OS. The Knights Corner Phi coprocessor has 61 cores, and 4 hardware threads per core. Phi coprocessors can run single and double precision calculations. The Cores have also L1 and L2 caches, vector processing unit, and scaler unit [83]. The super computer Tianhe-2 has 48,000 Xeon Phi 31S1P coprocessors [84]. Figure 5 shows an active Xeon Phi processor (left) and a passive one that is cooled externally (right).

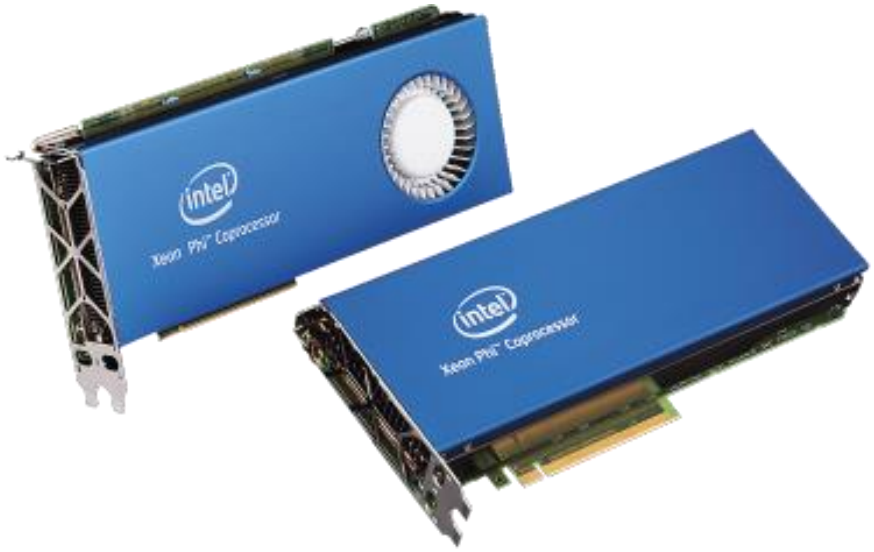


Figure 5: Intel's Xeon Phi coprocessor [83]

1.2 Programming Coprocessors

There are many frameworks that are used to program accelerators, such as OpenCL [4], OpenACC [86], CUDA [6], and OpenMP [85]. Message Passing Interface (MPI) is language independent standard that is used to pass data between connected coprocessors or CPUs. There are

many implementations of MPI standards, and some are free and open source [105]. MPI has many functions to do data collection and synchronization [105]. MPI can be used with other accelerator programming standards to program heterogeneous accelerator clusters [85, 105].

OpenCL can be used to program parallel applications that can run across heterogeneous systems. Many hardware manufacturers adopted this open standard, such as Nvidia, AMD, Apple, IBM, and Samsung [4]. OpenCL provides a low-level programming framework that can achieve good performance, but using OpenCL can produce less portable code when it is used to write applications for a specific hardware.

OpenMP (Open Multi-Processing) is another framework that provides a set of APIs to be used to parallelize programs [85]. A main thread usually forks into a number of sub threads that can be used to process the work load simultaneously. There are APIs to identify threads, sum data from threads, and synchronize threads.

OpenACC (Open Accelerator) [86] is a new programming standard that aims to provide more support for programming heterogeneous platforms. OpenACC has high-level directives that can be used to parallelize loops and optimize data locality.

CUDA (Compute Unified Device Architecture) [6] is a proprietary parallel programming framework that is developed by Nvidia. CUDA can only be used to program Nvidia's GPUs. Nvidia designed CUDA to work with C, C++, and FORTRAN, which makes it easier to use. As it can only run on Nvidia's GPUs, CUDA has many functions that could be used to exploit the hardware features of those GPUs.

1.3 GPU Technology Development

Before GPUs, many computer hardware manufacturers introduced graphics controllers that were used to accelerate graphics drawing. Some of the graphics controllers had general purpose languages that can be used to write general purpose programs; however, they were very hard to learn and use.

Throughout the 1980s and 1990s, graphics controllers continued to evolve, and 3D graphics cards were introduced to meet the increasing demand for more realistic and high resolution games and movies [3, 6]. In 1999, NVIDIA introduced the GeForce 256 graphics card, which is considered to be the first consumer-level graphics processing unit (GPU) [1, 2] that had integrated the capabilities of rendering 3D images in real time, and a programmable framework for parallel programming in a single chip.

With the introduction of the GeForce 256, the term GPU became popular and other manufacturers adopted the name or introduced similar terms. After that, GPU technologies continue to evolve, and new GPUs are consistently introduced that have better capabilities than before in terms of number of processing cores, memory, bus speed, and core clock rate.

1.3.1 GPU Architecture

The latest architecture introduced by NVIDIA is the Maxwell [79, 87] architecture, which was released in February 2014. However, many of the current GPUs are still built on the previous Kepler [46] and Fermi [45] architectures.

1.3.1.1 Fermi Architecture

The Fermi architecture was introduced in 2010 [45], and came with many major improvements over the earlier Tesla architecture. Figure 6 shows the main hardware components for the Fermi GPUs, while Figure 7 shows the Streaming Multiprocessor (SM) architecture.

The basic building blocks for a Fermi GPU are:

1- Streaming Multiprocessors (SMs):

The SMs are the main processing blocks on the Fermi GPU. Each SM has 32 CUDA cores in hardware revision 2.0, and 48 cores in the hardware revision 2.1. A CUDA core is a processor that is equipped with a pipelined arithmetic logic unit (ALU) and a floating-point unit (FPU).

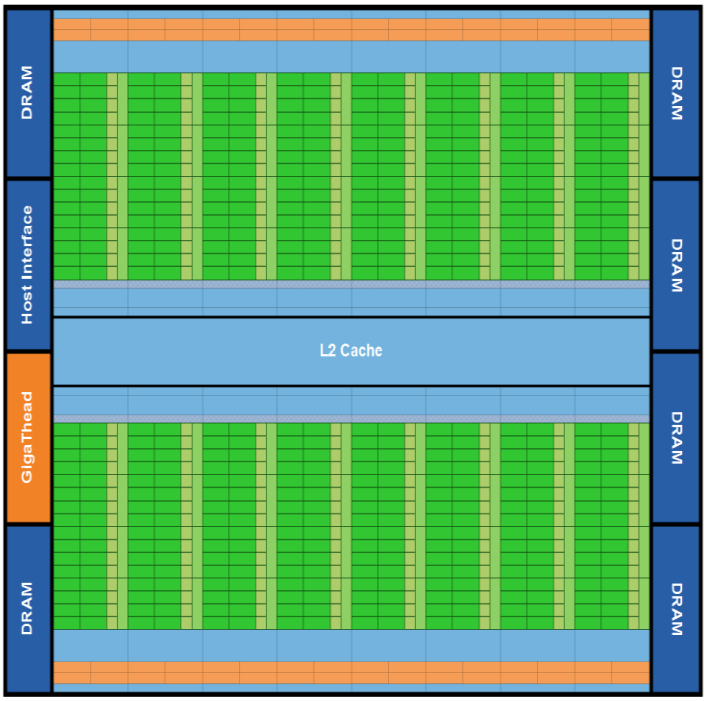


Figure 6: GPU hardware architecture for the Fermi platform . There are 16 SMs (vertical rectangular blocks) [45]

Figure 7: Fermi SM architecture [45]

Each SM can perform up to 16 double-precision operations per clock cycle, which is a considerable improvement over the previous architecture. This improvement helps in doing more accurate simulations.

Load/Store (LD/ST) units are responsible for calculating the source and destination addresses for the memory. Having sixteen of those (LD/ST) units on board a Fermi GPU will enable threads to do sixteen (load/store) address calculations per clock cycle [45]. In addition, each SM has four *special function units* (SFU) that can be used to execute some math functions such as sine, cosine, and square root [45].

To handle thousands of threads, the GPU follows the single instruction multiple data (SIMD) model. Threads are scheduled in batches of 32s called warps [3]. To organize the execution of thread warps, the Fermi GPU has a dual *warp scheduler*. At each clock cycle, each scheduler will select an instruction from a warp and assign it to a group of 16 processors or the four SFUs. Integer, float, load/store, and SFU instructions can be dual issued, while double precision instructions cannot be dual issued.

Another improvement over the previous architecture is having a full hierarchy memory, with shared memory and an L1 cache that share 64K on each SM.

2- Memory Hierarchy:

There are different types of memory that can be used in CUDA. Memory types differ in bandwidth, access rate, and size. The main types of memory in CUDA are:

a. Global memory: The global memory is the largest memory on the GPU; however, it is the slowest memory. The global memory can be used to share data between all threads on the device. To achieve efficient memory accesses, data reads and writes should be coalesced.

b. Shared memory: The shared memory is an on-chip memory that is faster than the global memory. Shared memory can be used to share data between threads in the same thread block. However, bank conflicts can decrease the access speed to data in

shared memory. Bank conflicts can be reduced by distributing the values in shared memory so that each thread in a warp either accesses the same shared memory value or values in different banks. Since memory is allocated among the banks in 4-byte increments, a double spans two adjacent shared memory banks.

c. Registers: Registers are the fastest memory type on the GPU; however, they are limited in number and size. Registers are used to store local variables in a kernel, but if there are not enough registers to store all the local variables, global memory will be used to store those variables.

d. Constant memory: Constant memory is a 64 KB read-only memory that is used to store constants. Only 8 KB is cached on an SM.

e. Texture memory: Texture memory is a cached read-only memory that is optimized for 2D spatial locality. Threads that belong to the same warp that access nearby texture memory locations will get better memory access performance.

3- Error Correcting Code (ECC):

Data inside memory can be altered by outer factors such as radiation, so the Fermi architecture added an ECC unit that detects and corrects such errors.

4- GigaThread Thread Scheduler:

At the chip level, Fermi schedules threads at a global level by distributing thread blocks to different SMs. Fermi GPUs also introduced many more improvements such as faster atomic operations, enhanced reductions, faster context switching, support for concurrent kernel execution, and improved branch prediction.

1.3.1.2 Kepler Architecture

Kepler came with many improvements over the Fermi architecture in terms of throughput, memory bandwidth, and power consumption [46]. Figure 8 shows the Kepler GPU architecture. New features of the Kepler GPUs include:

1- The new Streaming Multiprocessor (SMX) Architecture:

SMs in the Kepler architecture have far more cores and capabilities than the Fermi architecture. Figure 9 shows the Kepler SMX architecture. The first thing to be noticed is the number of CUDA cores per SMX has been increased to 192. A major improvement in the double-precision support is an increase in double-precision units. Now, there are 64 double-precision units in each SM. In addition, there is an 8-fold increase in SFU units, and a 4-fold increase in LD/ST units [46].

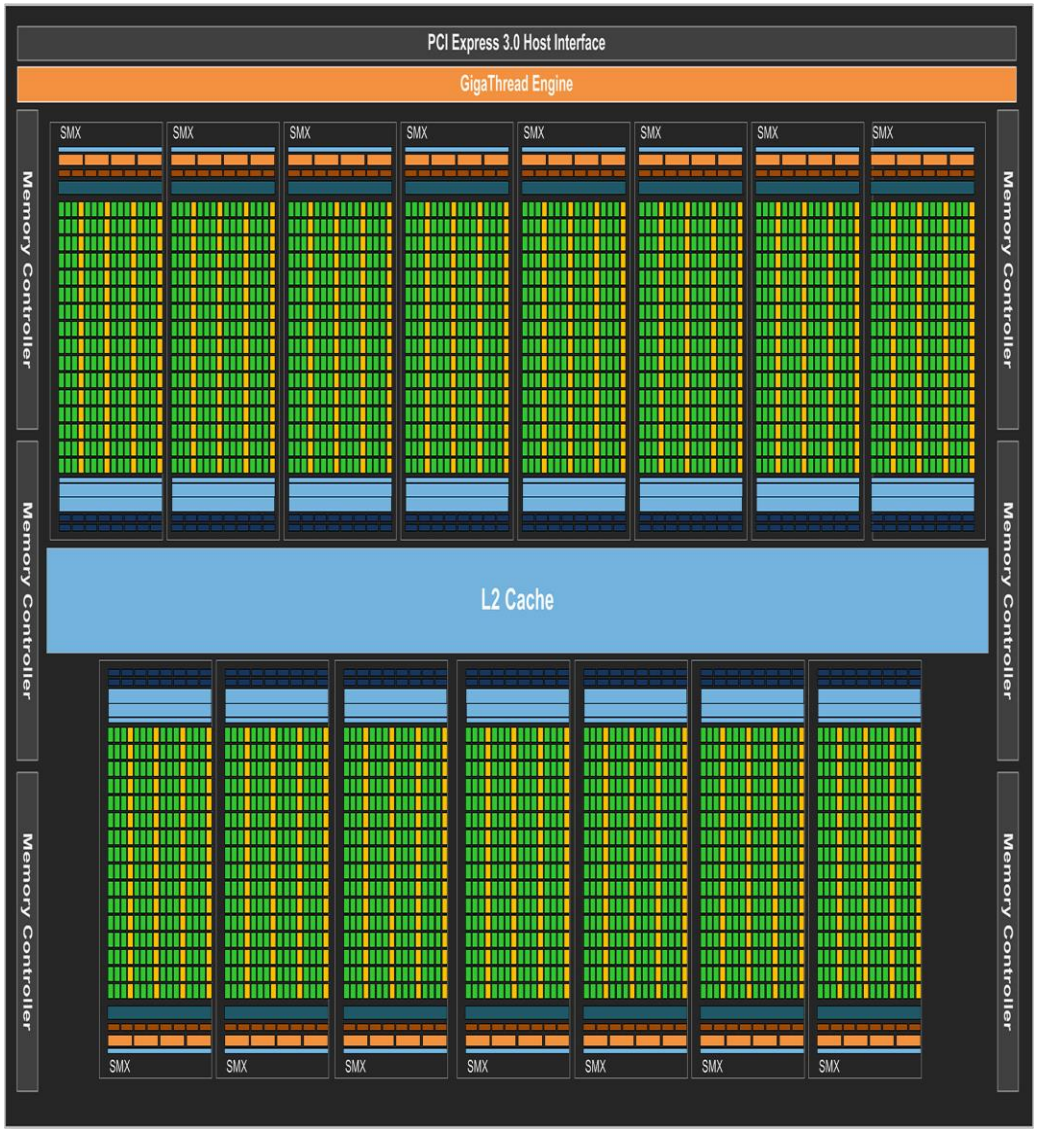


Figure 8: Kepler GPU hardware architecture [46]

Figure 9: Kepler SMX hardware architecture [46]

Another major improvement is the introduction of two more warp schedulers, which means that four warps can be issued and executed concurrently. Moreover, each Kepler warp scheduler is now equipped with two instruction dispatching units, allowing for more concurrent execution. In addition, double-precision instructions can now be dual issued.

Other improvements include the introduction of the shuffle instruction that allows threads belonging to the same warp to share registers, an increase in the number of registers per thread, the ability to configure shared memory for 8-byte banks for increased bandwidth and better support for double-precision numbers, the expansion and acceleration of atomic operations, and an increase of the GPU texture memory throughput.

2- Dynamic Parallelism:

In Fermi, the GPU cannot generate new work unless the CPU does that for it. In other words, all kernels are launched by the CPU. In Kepler, a new concept called dynamic parallelism was introduced to enable the GPU to launch kernels by itself, independent of the CPU. By using dynamic parallelism, the GPU can adapt the flow of the kernel executions and launch the required number of threads directly.

3- Memory Enhancements:

Kepler has a similar memory hierarchy to Fermi; however, Kepler enables the use of the read-only 48 KB data cache that was only accessible by the Texture Unit in Fermi. In addition, shared memory and L2 cache bandwidths are doubled.

Additional Kepler improvements include support for multiple CPU cores to launch work on the same GPU by introducing Hyper-Q, and direct GPU access through the network without going through the CPU memory by introducing GPUDirect [46].

1.3.1.3 Maxwell Architecture

The latest architecture from NVIDIA is the Maxwell architecture [79]. The main goals of introducing this new architecture are to develop GPUs for smaller computer platforms, and to increase performance while consuming less power. Figure 11 shows that the performance per Watt doubled compared to the previous Kepler architecture, and the performance per core is 35% more than in Kepler. To achieve those goals, NVIDIA introduced a new streaming multiprocessor architecture called SMM [79]. The new SMM is designed with more L2 cache and shared memory to improve performance; in addition to a group of architecture design changes that enable the Maxwell architecture to achieve double the performance for the same amount of power compared the Kepler architecture [79]. For instance, the new SMM uses four control logic units to dispatch the instructions, as shown in Figure 10, and the number of active threads per block increased from 16 in Kepler to 32 in Maxwell. In addition, new improved algorithms are designed to enhance the scheduling process. However, there are no high-end GPUs manufactured on the Maxwell architecture yet. More information on the Maxwell architecture can be found in [79, 87].

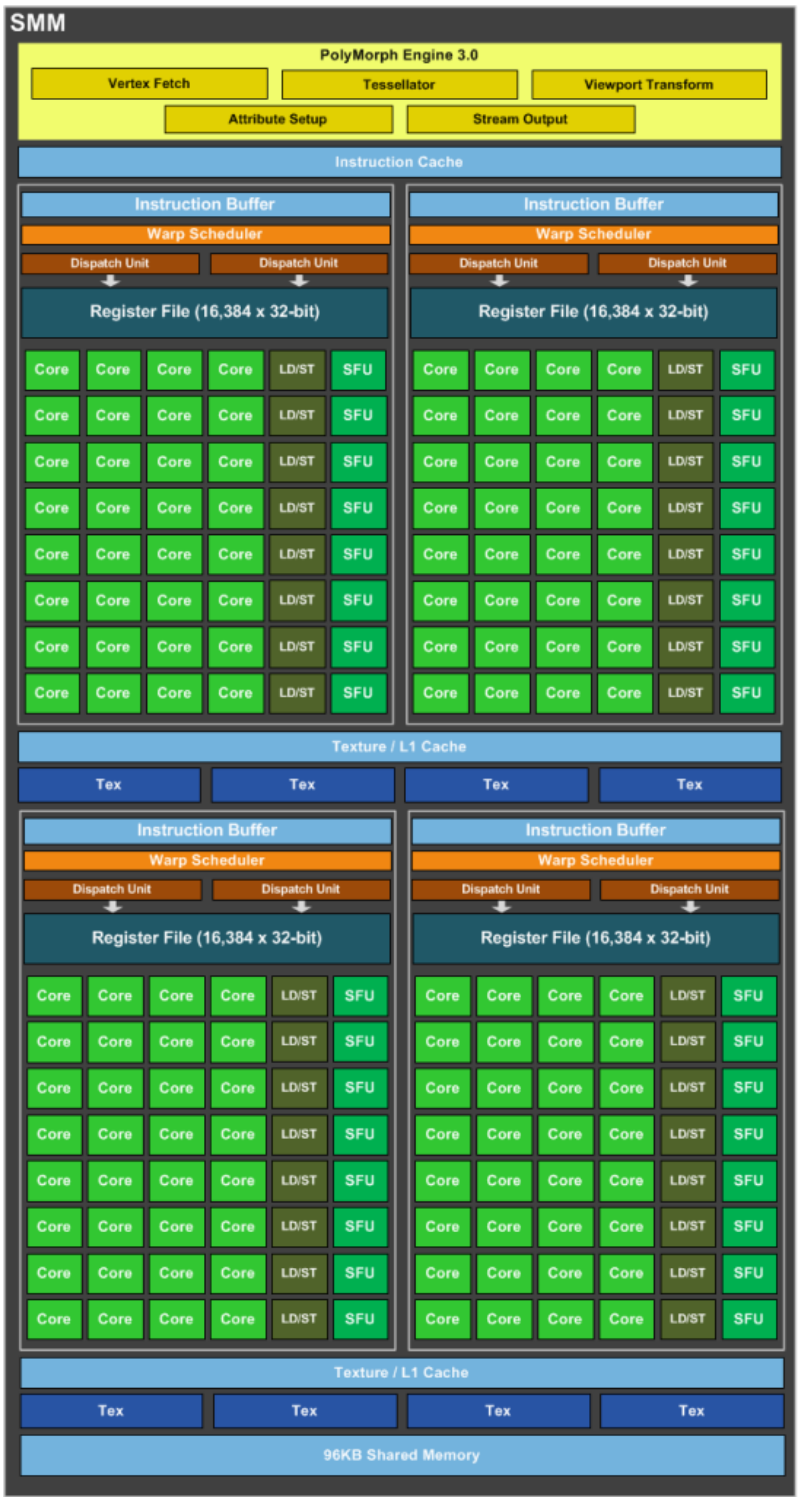


Figure 10: Maxwell SMM hardware architecture [79, 87]

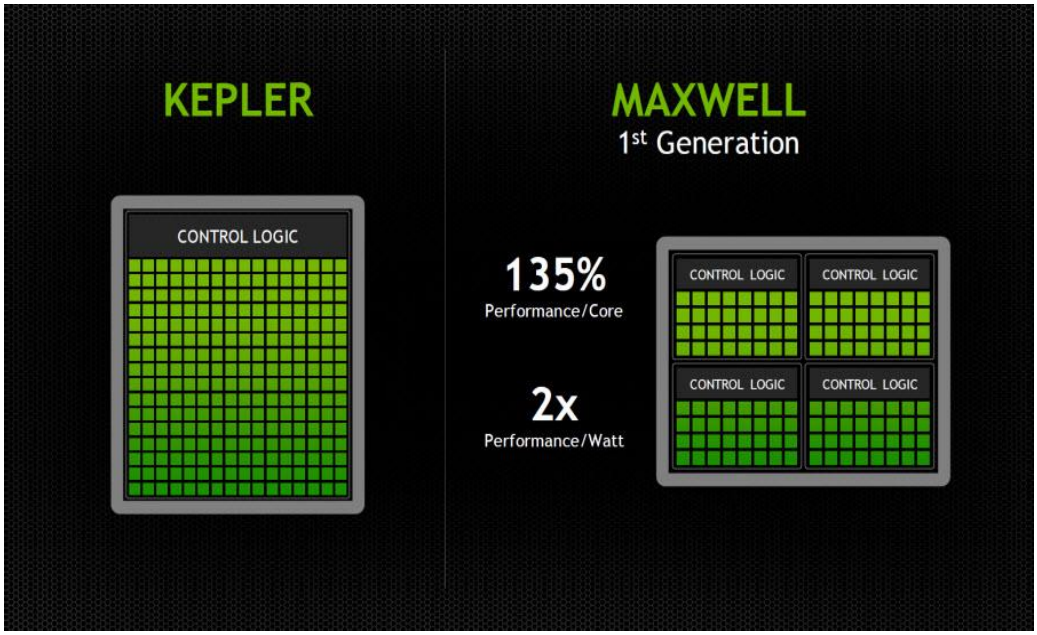


Figure 11: Nvidia's Kepler vs. Maxwell [87]

1.3.2 GPU Programming

While OpenCL is used to program different GPU architectures, CUDA runs only on NVIDIA's GPUs. CUDA also provides a large number of libraries [6] that are optimized for its GPUs such as:

- cuFFT: Library for Fast Fourier Transformations.
- cuBLAS: GPU accelerated BLAS library.
- cuSPARSE: GPU functions for sparse matrix operations.
- Thrust: Open source library of different data structures.
- cuRAND: GPU accelerated random number generator.

In addition, CUDA provides more built-in features and functions, supports templates, and has more support for developers. A showcase of CUDA libraries can be viewed at [7]. The main drawback of CUDA is that it is not an open standard. Since OpenCL is an open standard, it can be used on AMD GPUs, NVIDIA GPUs, and Intel Xeon Phi co-processors, along with other multi-core platforms.

CUDA simplifies many operations that were very hard to implement using earlier GPU programming languages, and provided a list of instructions to support parallel programming, thread management, synchronization, and memory management [6]. Figure 12 shows how CUDA compiles the CPU and GPU integrated codes.

1.3.2.1 Synchronization in CUDA

Synchronization is an important feature in any parallel programming framework. As threads execute in parallel, there is no guarantee on the order in which they will be executed. Hence, synchronization is needed to organize the execution of parallel programs. CUDA provides a set of synchronization tools for programmers.

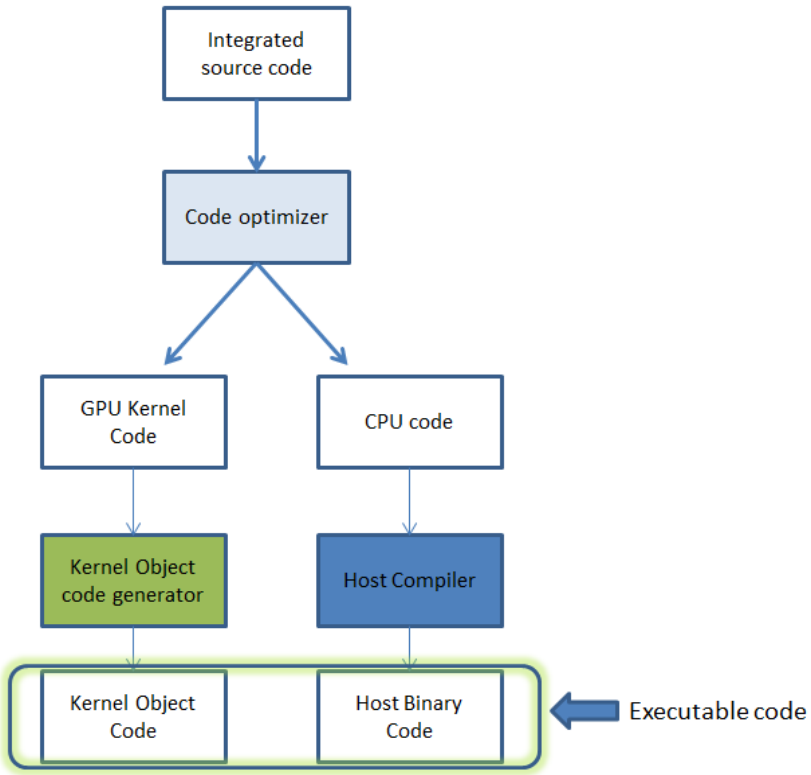


Figure 12: CUDA kernel compilation process

CUDA kernels are launched asynchronously; thus, after the host launches a kernel, the CPU will continue with the program execution. In some cases, results from the kernel are necessary for making decisions or generating output. As a result, CUDA has a statement called cudaDevic-

eSynchronize() [6]. cudaDeviceSynchronize will block the host until the kernel is finished executing. However, memory copy statements after the kernel launch will also block the host without requiring an explicit synchronization statement.

To synchronize threads in a thread block, CUDA has the __syncthreads() [6] statement. CUDA did not implement a function for cross-block synchronization because it can be costly in terms of performance. There are ways to do it programmatically by using atomic operations as locks, but again it can degrade performance.

CUDA also provides memory fence functions such as __threadfence() [6] and _threadfence_block() [6]. When a thread calls the __threadfence() function, it will block until all its previous writes to global memory and shared memory are visible to all other threads. The __threadfence_block() [6] instruction works the same way as __threadfence(), but on a block level.

1.3.2.2 Kernels and Device Functions

The main units of code execution on the GPU are called kernels. Kernels are created by putting the __global__ directive before the function definition. An example of a kernel function definition is:

__global__ void MyKernel(parameters)

Kernels cannot have a return type because they cannot return values directly. The only way of returning values is to use memory copy functions. To launch a kernel, the programmer should specify the number of threads per block and the number of blocks, and provide the function arguments. In some cases, kernels may have dynamic shared memory, so the programmer must also provide the size for that memory space. A kernel call would look like this:

MyKernel<<<Grid Size, Block Size>>> (arguments)

CUDA threads are organized into thread blocks, and blocks are organized into a grid. Threads inside a thread block can be organized into one, two, or three dimensions, with a limit of 1024 threads per block on most GPUs. Blocks within a grid can be organized into one, two, or three

dimensions. This flexibility in thread and block organization can be very useful in applications that have multidimensional data. Figure 13 gives a view of how threads, thread blocks, and grids relate to each other. By having threads divided into thread blocks, the hardware can scale the execution of the kernels to any GPU without the need to change the code.

When a kernel is launched, the grids are assigned to SMs to be executed. A thread block is assigned to one SM, and an SM can have more than one block assigned to it depending on how many threads are in that thread block. Registers and shared memory are also partitioned among threads and thread blocks.

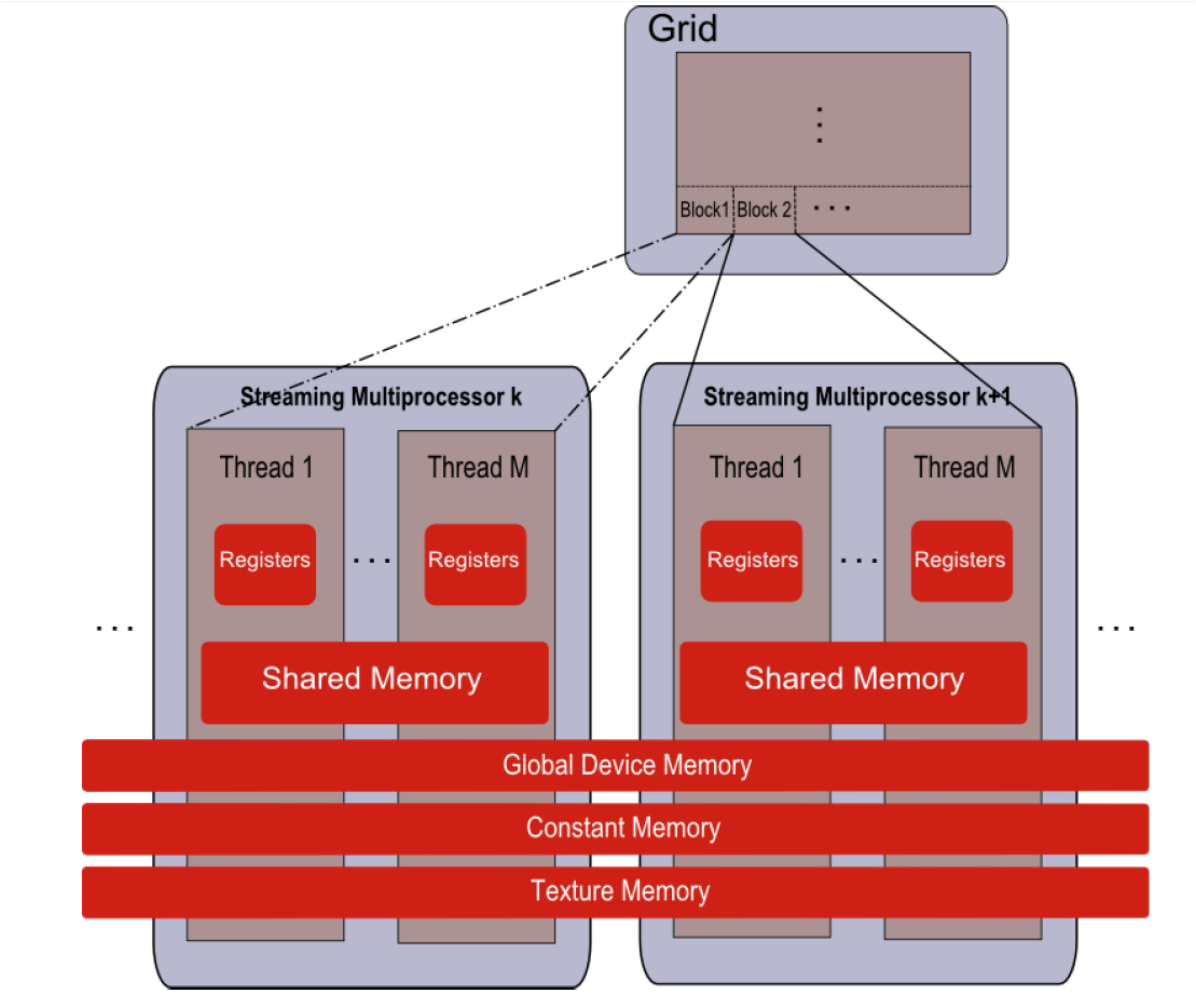


Figure 13: CUDA memory model [36]

1.3.2.3 CUDA Memory Model

Threads inside a thread block can communicate using shared memory. Shared memory provides a fast way for threads to share data. Each thread can store local variables inside registers. However, shared memory and registers are limited in size, so to store large data structures; the GPU uses the global DRAM memory, which is the slowest type of memory on the device. Careful planning of the use of the memory types and how data is partitioned among them can enhance performance. Figure 13 shows the CUDA memory model and how it is related to threads and thread blocks.

1.3.2.4 GPU-CPU Communication

Data is transferred into and out of the GPU by using memory calls [2]. Those memory calls can affect performance if not used carefully. Data can also be moved asynchronously between the GPU and CPU by using asynchronous memory calls. Data transfer between the CPU and the GPU is very time consuming, and thus should be reduced to a minimum.

1.3.2.5 Functions and Libraries

CUDA provides many libraries that are GPU optimized. In addition, CUDA provides a set of alternative math functions called intrinsic functions [28]. Intrinsic functions are faster than standard math functions in CUDA; however, they are less accurate. These functions may be used in calculations that can tolerate some loss in accuracy to gain more speedup. In addition, there is a set of atomic instructions that can be used to provide locks on data when it is modified. Examples of atomic functions are *atomicAdd*, *atomicSub*, *atomicDec*, and *atomicAnd* [28].

1.3.2.6 Compute Capability

In CUDA, the compute capability specifies the architecture of the GPU, described in terms of major and minor revision numbers. When two GPUs have the same major revision number, then this indicates that they have same architecture. The current major revision numbers are one, two, three, and five corresponding to Tesla, Fermi, Kepler, and Maxwell architectures, respectively.

The minor revision number specifies improvements that are made on the same architecture. More details on what capabilities each one of the CUDA compute capabilities have can be found in [6].

1.3.2.7 CUDA Streams

One of the powerful concurrency features of CUDA is CUDA streams. A stream is a sequence of instructions that are executed in the order that they are issued on the GPU [2, 28]. By default, there is one stream that the kernels are launched through. CUDA streams are used to achieve concurrency beyond the multithreading level. Instead of executing one kernel at a time on the device, CUDA streams can be used to execute a number of kernels concurrently on the device, which can be used to introduce more speedup. However, the ability to run multiple kernels concurrently depends on the device, which in this case should be of compute capability 2.0 and up. Another factor that is important is the availability of resources on the device. If each kernel uses a lot of hardware resources, then there will not be enough resources to run multiple kernels concurrently.

1.4 Research Contributions

The two research projects presented in this dissertation are: the development of an open-source Monte Carlo GPU code for thermodynamic ensemble interactions called GPU Optimized Monte Carlo (GOMC) [70], and the development of a GPU code for accelerating the computation of polygrain growth in the Phase-Field Crystal (PFC) [112, 113] model.

GOMC is an NSF-funded interdepartmental project with Professor Jeffrey Potoff's research group from the Department of Chemical Engineering and Materials Science. The GPU PFC project is also a joint project with Professor Zhi-Feng Huang from the Department of Physics and Astronomy. The enhancement of their software to run on the GPU is an important step for increasing the problem size and features of the systems, both of which allow deeper scientific understanding of the behavior of these systems.

Monte Carlo (MC) simulation has been used to study many problems in statistical physics and statistical mechanics that are not possible to simulate using Molecular Dynamics (MD) [8]. One

example is the simulation of adsorption of gases in porous materials [8]. While there are a fair number of well-known and widely used GPU Molecular Dynamics codes, such as LAMMPS [9], NAMD [10], AMBER [11], and HOOMD Blue [12], the existing Monte Carlo ensemble simulations are relatively slow and so are not practical for simulating large systems. In addition, those Monte Carlo codes are not yet ported to the GPU, which makes it almost impossible for researchers to run large systems in a reasonable amount of time. GOMC is created to address these shortcomings of existing Monte Carlo ensemble codes.

There are many challenges I faced when developing the GOMC GPU Monte Carlo molecular simulation. For instance, data structures need to be designed to enable memory coalescing for GPUs, the use of different techniques to optimize the ways to calculate energy interactions for the GPU, when and what data needs to be copied from and to the GPU, how to enable the code to scale when executed on more than one device, and the ability of the code to simulate systems with different types of molecules. Through this work, I introduced many optimizations that targeted those challenges.

The GPU PFC modeling is motivated by recent research efforts devoted to the understanding of the properties of crystalline materials, both their design and control. Recent developments include the introduction of new models to simulate system behavior, and novel properties that are of significant experimental and theoretical interest. One of those models is the Phase-Field Crystal (PFC) model [112, 113]. The PFC model has enabled researchers to simulate 2D and 3D crystal structures and study defects such as dislocations and grain boundaries. In this work, the Multicore Computing Lab carries out large-scale computer studies on GPUs to examine various dynamic properties of polycrystals in the 2D PFC model. Some properties, such as the Orientational Correlation function (g_6) [26, 35], require taking the circular average over different radii for every atom. This is very compute intensive when the system has hundreds of thousands of atoms.

This thesis reviews related work on both the GOMC and PFC projects in Chapter 2. Chapter 3 will describe the GOMC main components, flowchart, data structures, how energy interactions

are implemented and optimized using the cell list structure, and finally present and discuss the results. Chapter 4 will go over the PFC model, how the model was implemented and optimized for running on GPUs, the calculation of different PFC related properties, and finally present and analyze the performance of the PFC solver and different properties that run on the GPU. Finally, Chapter 5 will present the overall conclusions from this work, and what are the future contributions that I am planning for the two research projects.

CHAPTER 2 RELATED WORK

This chapter will present an overview of the Monte Carlo simulations, giving details on the ensembles programmed in this work, and how they are calculated. In addition, the chapter will go over grain growth, while focusing on the PFC model that is used in this work. The chapter will also present related work on some well-known Monte Carlo molecular simulation engines.

2.1 Monte Carlo (MC) Simulations

MC methods are a set of stochastic methods that use random numbers and probability statistics for problem investigation [16]. Through the use of repetitive random sampling on the input domain, then processing the selected inputs, MC methods try to converge to a steady-state solution. There are many applications of MC methods in the fields of physics, finance, artificial intelligence, and biology [8, 17, 19].

One of the applications of the MC method is the simulation of molecular systems. A popular MC model that is used to simulate such systems is called the Metropolis method [17]. The Metropolis method is used to evolve the system through multiple iterations that consist of selecting particles or molecules, performing a type of interaction with that selected particle or molecule, calculating the energy change, and then deciding based on a random value whether or not to accept that interaction. The Metropolis main steps are:

1- Generate initial system configuration.

2- Perform a move, such as particle displacement. The particle should be chosen and displaced randomly.

3- Calculate the energy change (ΔE) for the displaced coordinates.

4- Decide whether to accept the move or not:

   a. If $\Delta E < 0$, accept the move, save the new coordinates, then go to step 2.

b. Else, calculate $e^{\left(-\frac{\Delta E}{kT}\right)}$, and draw a random number *R* from the [0,1) range. If *R* > $e^{\left(-\frac{\Delta E}{kT}\right)}$ then accept the new coordinates. Else, reject the move and keep the old coordinates. In either case, return to step 2.

2.1.1 Statistical Thermodynamics Ensembles

One of the MC applications of molecular systems is the thermodynamic ensemble simulation [8, 19]. Ensembles represent the thermodynamic properties of a system. This work focused on the following three main ensembles, canonical ensemble, grand canonical ensemble, and Gibbs ensemble.

2.1.1.1 Canonical Ensemble

Canonical ensemble is one common ensemble in which the number of molecules or particles (N), box volume (V), and temperature (T) are fixed, so sometimes it is referred to as NVT [18]. NVT can simulate two moves, molecule or particle displacement, and molecule rotation.

Acceptance criteria are measured by using the Boltzmann factor given by:

$$e^{-\beta \Delta E} \tag{2.1}$$

where ΔE is the energy change between two states, and β is equal to $1/(k_B T)$, where k_B is the Boltzmann constant and T is the temperature in kelvin [18].

To calculate the Boltzmann factor for a move, ΔE needs to be calculated, which represents the change in energy between the old and new positions. After the Boltzmann factor is calculated, the result will be compared against a random number drawn uniformly from the [0,1) range. If the Boltzmann factor result is larger than the drawn random number, the move will be accepted, and the new coordinates are committed. NVT pseudo-code is shown in algorithm 1.

```
Canonical Ensemble Algorithm
input: steps, Number of particles , Volume, Temperature
// Calculate the system's initial energy
// Main Loop
for i := 1 to steps do
    // Randomly select a particle to move
    s ← rand()
    Old_particle_loc := particle_location(s)
    // Randomly move to a new location
    New_particle_loc := randCoords()
    // Calculate the selected particle's energy for the old and new locations
    for k := 1 to number of particles do
        if k!=s then
        old_energy_contrib += calculate_pairwise_energy(Old_particle_loc, k)
        new_energy_contrib  += calculate_pairwise_energy(New_particle_loc, k)
        end if
    end for
deltaE := new_energy_contrib–old_energy_contrib.
calculate_acceptance_rule()
if accepted then
    total_energy  += deltaE
    current_config := new_config
    update_system_status()
end if
updateMoveStatistics()
//Solve if the system in equilibrium state
// Periodically write system status to disk
end for

// End Algorithm
```

Algorithm 1: Canonical ensemble pseudo-code

2.1.1.2 Grand Canonical Ensemble

The grand canonical ensemble extends the canonical ensemble by defining temperature, volume, and the chemical potential as constants [18]. A reservoir is connected to the simulated system, allowing the particles and energy to be exchanged freely between them. Through this exchange of particles, the system and the reservoir will reach an equilibrium state, which can be determined by using the fixed values of the temperature and the chemical potential.

Figure 14 gives an example of a grand canonical simulation that has the simulated system with V volume (N particles), and the reservoir with V_0-V volume (M-V particles). Particles can interact with each other only when they exist inside the simulated system. The grand canonical pseudo-code is shown in algorithm 2.

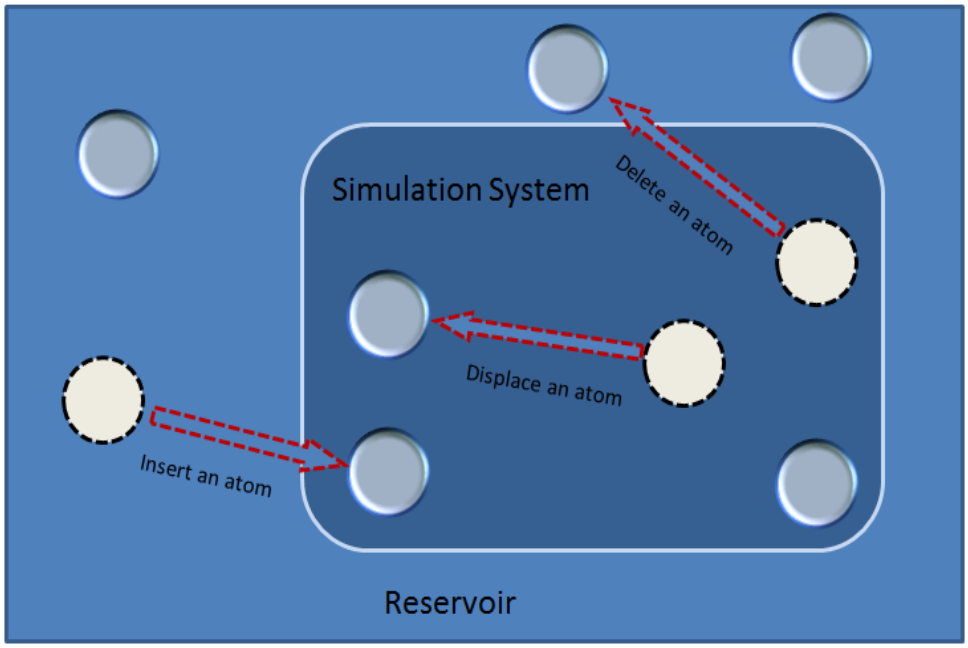


Figure 14: Particle exchange in the Grand Canonical Simulation

```
Grand Canonical Ensemble Algorithm
Input: steps, Number of particles, Volume, Temperature, Infinite Reservoir
//Initialize particles' coordinates inside the box randomly
// Calculate the system's initial energy
//Main simulation loop
for i= 1 to steps do
    //Randomly select a move type
    R ← rand()
    if (R < DisplacePercent) then
        //Attempt particle displacement
    else
        //Attempt particle transfer (Insertion/Deletion)
        //Choose a random source (Box or Reservoir)
        Source ← rand()
        if (Source < 0.5 ) then
            //Source box is the Box remove a random particle)
        else
            //Source box is the reservoir (Insertion a new particle to the box)
        end if
    end if
 //Solve if the system in equilibrium state
 //Periodically update system status to disk
 end for
// End Algorithm
```

Algorithm 2: Grand canonical ensemble pseudo-code

2.1.1.3 Gibbs Ensemble

Gibbs ensemble is used to simulate phase equilibria in vapor-liquid coexistence systems. In addition, Gibbs ensemble can be used to simulate many more systems such as solid-fluid equilibria, solid-vapor equilibria, adsorption equilibria, and membrane equilibria [36].

To model coexistence systems, we need to have two boxes. A series of moves can then be performed on those boxes, which include:

1. Particle or molecule displacement within a box:

This move is the same move used in canonical and grand canonical ensembles.

2. Volume transfer:

   Transfer an amount of volume from one box to the other.

3. Molecule or particle transfer:

   An particle or molecule can be transferred from one box to another. However, there are different ways to do this.

Figure 15 shows an illustration of the Gibbs ensemble moves. Algorithm 3 gives the Gibbs ensemble pseudo-code.

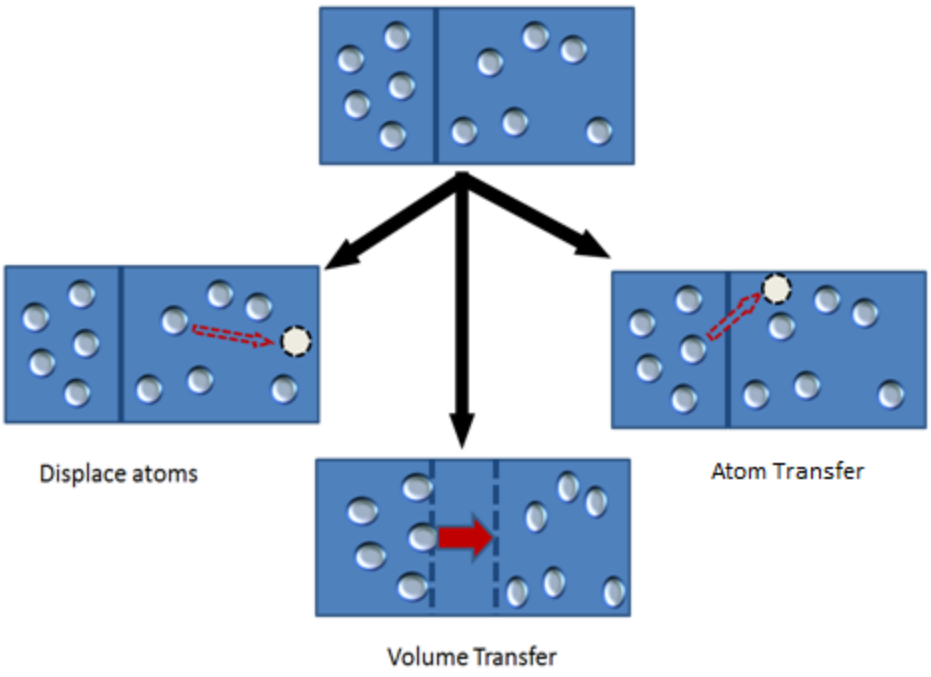


Figure 15: Gibbs ensemble moves

2.1.1.4 Configurational Bias

Sampling chain-molecules in MC simulations is a very important issue to achieve configurational equilibrium. A great deal of research has been devoted to the development of efficient methods to sample different structures for chain molecules; however, many of those methods will not work in dense systems [19]. One way to address the sampling problem is to completely re-

build the whole molecule or parts of it, while biasing the build toward preferable configurations. This method was proposed by Siepmann and Frenkel [19], and it was named configurational bias MC (CBMC). CBMC is based on the self-avoiding random walk algorithm that was proposed by Rosenbluth.

CBMC starts by choosing different random positions for the next particle to build. Those positions must not be occupied by any other existing particle in the system. For each generated trial position, one needs to calculate the Rosenbluth weight [19].

Gibbs Ensemble Algorithm

```
Input: steps, Temperature, Two boxes of volume (V1,V2) and number of particles (N1,N2)
//Main simulation loop
for i = 1 to steps do
    // Select a move type randomly
    R← rand()
    if (R < disp_percentage) then
            //Attempt particle displacement move
            //Select a box randomly
            selectedBox← rand()
            // Attempt to displace an particle in the selected box
    else if ( R < (disp_percentage + vol_percentage ) ) then
            // Attempt Volume Transfer
    else
             //Attempt particle transfer
             //Randomly select a source box
             sourceBox ← rand()
             // perform an particle transfer move to the other box
    end if
 //Solve if the system is in equilibrium state
 //Periodically write system status to disk
 end for

// End Algorithm
```

Algorithm 3: Gibbs ensemble pseudo-code

To achieve detailed balance [19], the trials are done at both boxes, where the trials at the source box are referred as old trials. As we build new sites, old and new trial weights are accumulated and used in the end to accept or reject the move. More on CBMC can be found in [19, 27].

2.1.2 Lennard-Jones Potential

The Lennard-Jones potential is a mathematical approximation used to compute the energy interaction between a pair of particles or molecules that incorporates the attractive and repulsive forces [8, 36]. The Lennard-Jones potential calculation is done using the following equation:

$$V_{Lj} = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \qquad (2.2)$$

where σ is the particle diameter, ϵ is the well depth, and r is the distance between the two particles.

2.1.3 Calculating Force Interactions

Other simulation techniques, such as molecular dynamics, typically require significantly more computation for each step of the simulation, and so are better-suited for parallel implementation. Even so, some previous simulations have used the GPU to implement the MC method [51]. Their implementation depends on an embarrassingly parallel algorithm that runs several concurrent simulations with small systems of 128 particles. Instead, this work uses the energy decomposition method (farm algorithm), which enables us to support configurations with over a million particles. In [61], a parallelization method for the canonical MC simulations via domain decomposition technique has been presented, where each domain can be assigned to a separate processor and multiple moves can be simulated in parallel. Interprocess communication is required only when moving particles near the edge of a domain, since this requires interactions between adjacent domains. To limit this communication, each domain is partitioned into three subdomains. The size of the middle subdomain is chosen as large as possible to minimize interprocess communication. Although well suited for a multicore CPU, this approach does not expose the fine-

grained parallelism required for an efficient GPU implementation. Each time a particle is displaced in, removed from, or inserted into the simulation region, energetic decomposition requires that pairwise energy be calculated between this particle and all other particles. A radial cutoff, denoted as r_{cut}, is typically chosen to reduce the execution time by limiting the calculation of inter-molecular forces to only those particles within the cutoff. The forces due to interactions with particles outside of the cutoff can be approximated using tail corrections [18]. Since interactions within only a small radius are considered, it is possible to create either a cell list or a neighbor list to organize particles based on their relative locations and ignore particles that are beyond the cutoff. In this way, not only are the energy and pressure computations of more distant pairwise interactions avoided, but also the calculation of distances between these particles.

Another approach to calculating force interactions between particles is based on reducing or eliminating the interactions with particles that are beyond the cutoff by constructing a neighbor list or a cell list. One common example of the neighbor list is the Verlet list [89]. Verlet lists maintain a list of neighboring particles for each particle, where those neighboring particles all fall within the cutoff. While this list reduces the number of interactions that must be computed, it requires more frequent updating. For MD, the Verlet list is a good option, as all particles move simultaneously and the system is closed in terms of adding or deleting particles [90]. In contrast to MD, the MC system is open, and particles can be displaced for relatively large distances, which may sometimes require rebuilding the whole Verlet list.

In the conventional cell list approach [91], the simulation box is divided into cells (squares in 2D, cubes in 3D) such that the dimensions of each cell are greater than or equal to the cutoff. Here, the cells will limit the number of interacting particles by only considering interactions across adjacent cells. However, adjacent cells may still have particles that are outside the cutoff, thus there is a need to check all pairwise interactions in adjacent cells. When compared to Verlet lists, cell lists require less effort to maintain, especially when displacing or deleting particles. To reduce the number of extraneous processed interactions in the cell list approach, cell dimensions

may be selected to be smaller than the cutoff, or in some cases, make the cell small enough to fit only a few particles [92]. However, this approach will generate many fine-grained cells that need to be examined, and in sparse boxes, many of those cells will be empty [92]. MC simulations perform much less computation at each step when compared to MD, so approaches that show good performance for MD simulations using cell lists and Verlet lists did not yield performance gains when simulating small systems for MC interactions [92].

There are many examples of using a cell list implementation for the MD simulations [18, 47, 65, 93, 94]. On early GPUs, an efficient implementation of cell list on the GPU was not viable due to the lack of atomic operations on the GPU [51]. Instead, implementations such as [65, 93, 94] use the CPU to construct the cell list and then copy it to the GPU. These cell lists are then used to construct a neighbor list. Note that in molecular dynamics simulations, all molecules are moved in each step, requiring the cell list to be updated after nearly every simulation step. The frequency of updates depends on how far a molecule moves in each step, how much extra distance beyond the cutoff is used in defining the neighbors, and how much inaccuracy can be tolerated in the computations. A state-of-the-art implementation is described in [12].

A third option is to use both a Verlet list and a cell list [95]. For instance, Proctor et al. [90] show cell lists on the GPU allow a fast approximation of whether or not two particles are within the cutoff, which performs better than immediately traversing the neighbor list. They do not create or maintain a cell list, but calculate the cell of each particle based on its coordinates, with cell dimensions larger than the cutoff, and use this calculation to determine whether or not two particles are in neighboring cells.

2.1.4 Molecular Simulation Engines

Molecular Dynamics (MD) and Monte Carlo (MC) computer simulations are the most widely used simulations in materials science.MD codes have been considered as better candidates for parallel implementation because each simulation step in a MD simulation requires considerably huge computation effort when compared to MC simulations. For this reason, many molecular dy-

namics codes have been developed, some of which have been modified to utilize the GPU, including LAMMPS [9], NAMD [10], AMBER [11], and HOOMD-Blue [12]. On the other hand, there is a class of problems that cannot be simulated using the current methodologies. For example, adsorption in porous materials is the sort of problem that requires the simulation of an open system, which requires a methodology that allows for fluctuation in the number of molecules in the system.

While MD simulations have been studied well by other researchers, other systems are impossible to simulate using these MD codes, such as the simulation of multicomponent adsorption in porous solids [97], which will open the door for solutions such as the development of novel porous materials for the sequestration of CO_2 and the filtration of toxic industrial chemicals. In particular, molecular dynamics (MD) codes cannot be used to simulate an open system without using a hybrid MC-MD approach [89, 99] because of the fluctuation property of MC that MD does not utilize.

Another general purpose molecular simulation is the HOOMD-Blue (Highly Optimized Object-Oriented Many-Particle Dynamics) simulation engine that is developed in Michigan State University. HOOMD-Blue is programmed to use GPUs to accelerate MD simulations, and it can scale up to thousands of GPUs, thus enabling it to perform very large simulations [12]. There is MC extension for HOOMD-Blue that is called Simpatico [100] that supports some MC algorithms. Another extension for HOOMD-Blue is called Hard Particle MC (HPMC) [41], which supports doing MC hard particle simulations [41].

Sandia National Labs started developing an open source simulation engine for MD simulation in 1995, which has the capability to run on parallel processors. The simulation engine is called LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [9]. LAMMPS can be run on many modern parallel accelerators, such as GPUs and Phi coprocessors. To improve efficiency and enhance performance, LAMMPS uses neighbor lists to track of close particles [9].

Loyens et al. [57] developed a parallel MC Gibbs ensemble simulation that specifies algorithms to parallelize each movement type of the MC simulation. For displacement, the system can be divided into regions, provided that the range of the interactions is short, so that the displacement of particles in one region does not affect other regions' energy interactions. Using the regions scheme, different processors can be responsible for calculating energy interactions in different regions. However, this scheme will fail if the system has long range interactions, or molecules that can span more than one region. For the volume move, each processor can calculate the energy interactions for a group of particles. Particle exchange can be parallelized by having different processors calculate a number of the trials that are used to select the best position when building the new molecule in the destination box [57].

Monte Carlo for Complex Chemical Systems (MCCCS) Towhee [102] is an open source simulation engine for Gibbs MC simulations. However, the code does not support running on modern accelerators. There is a parallel version that uses MPI to distribute the work load, which cannot guarantee to achieve huge speedups. Another MC molecular simulation engine is Cassandra that is developed by the Maginn Group [116]. Cassandra uses OpenMP to accelerate the simulation.

Another MC simulation engine has been created by the research group that developed HOOMD [41]. In this simulation, the simulation box is divided into cells. Particles are represented by circle disks. In a trial move, a disk is displaced to a random place. If the disk does not overlap with another one, the move gets accepted [41]. When compared to other MC molecular simulations, this simulation requires less computation as it does not have to check if the two particles fall within the radial cutoff.

2.2 Grain Growth

Polycrystalline materials are composed of grains of different crystal orientation. Those materials can be found everywhere around us such as in metals, alloys, and ceramics [14]. The behavior and properties of polycrystalline materials are determined by the shape, arrangement, and size of the grains [16]. Thus, a great deal of research is devoted to the understanding of those materials.

Grain boundaries are regions that separate two crystal structures with different orientations [13]. There could be different types of grain boundaries depending on how much misorientation there is between two grains. One type is called low-angle grain boundary, in which the misorientation is only a few degrees, at most ten degrees. If the misorientation is more than that, the boundary is called a high-angle grain boundary [14]. Figure 16 shows an illustration of the previously mentioned grain boundary types.

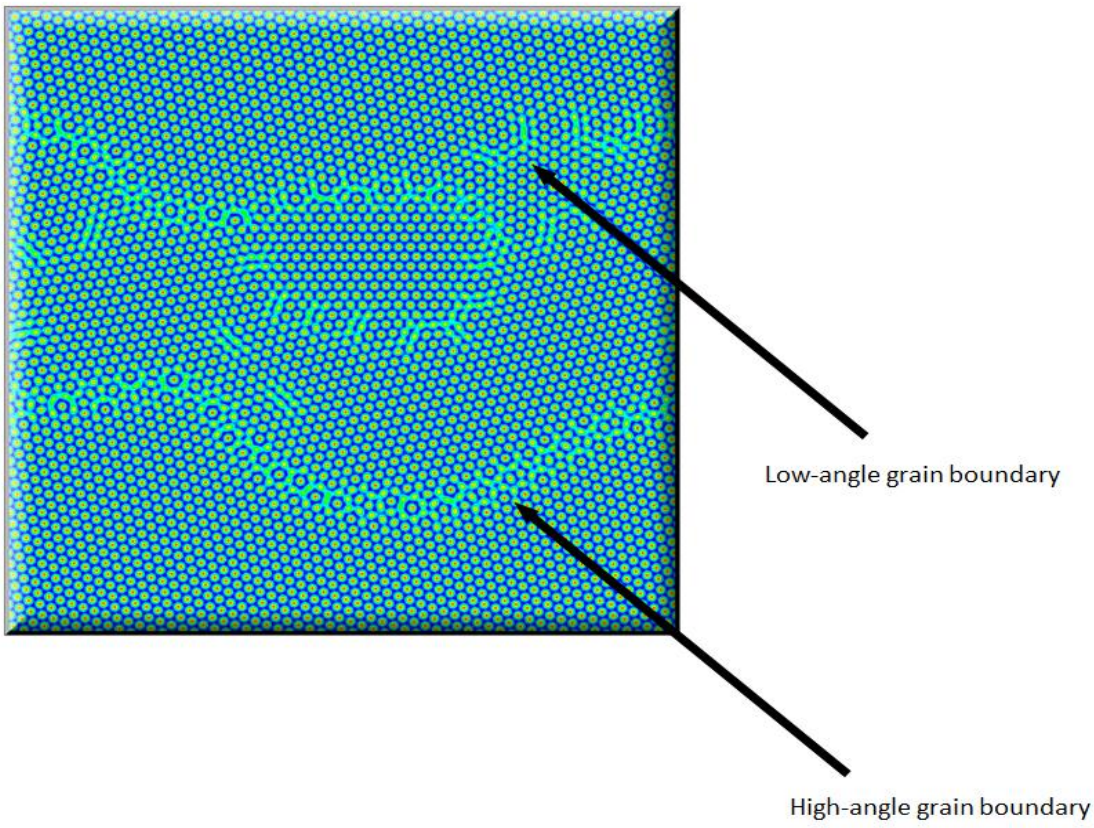


Figure 16: High and low angle grain boundaries

State of the art imaging technology enables scientist to take images of materials at the atomic level. Those images can show defects in the crystalline structure of those materials. However, those images can be large, and it is not easy to use them to detect defects just by looking at them. In addition, there can be a huge number of images that are generated for a material that is studied for a time period. As a result, a number of simulation models were developed to simulate the crystalline materials' behavior.

To simulate and model grain growth, the simulation model should include features for simulating multiple crystal orientations and simulating deformations. One way to simulate such systems is the use of Molecular Dynamics [14]; however, MD has some limitations regarding time scaling and the size of the simulation [19, 34].

Another model used to simulate grain growth is the Phase-Field Crystal (PFC) model. The PFC model can be used to simulate 2D and 3D grain growth simulations. In addition, the PFC model can be used to study defects, elasticity, and grain boundaries [16]. There are different equations that are used to describe the dynamics of atom movement and grain boundary migration.

The PFC model is an extension to the phase-field model [112, 113, 114]. In this extension, the system's atomic density evolution is described by the dissipative dynamics [112, 113]. In addition, the atomic density in the PFC model is periodic, thus minimizing the solid's phase free energy functional denoted by $\mathcal{F}$ [112, 113]. The periodic atomic density also allows the model to show elastic effects and crystal orientations [112, 113]. To minimize $\mathcal{F}$, we need to calculate:

$$\partial\psi/\partial t = \nabla^2[-\epsilon\,\psi + (\nabla^2 + q_0^2)^2 - g\psi^2 + \psi^3] \qquad (2.3)$$

where $= (3/B^S)^{1/2}/2$, q_0 is equal to 1 [114] and ψ is the atomic number density field.

As mentioned before, one of the applications of the PFC model is the study of grain growth and grain boundaries. There are many ways to detect grain boundaries, and one way is by detecting defects in the hexagonal lattices in the PFC simulation. A hexagonal lattice represents an atom and its six neighbors [14]. If the lattice has five or seven neighboring atoms, then it is called a disclination [15]. A pair of five and seven disclinations forms a dislocation. In some cases, a disclination can be identified as free and not bonded with another disclination. Figure 17 gives an illustration of a hexagonal lattice and two disclinations.

There are different properties that can be measured to study the grain size such as the density of correlation lengths and moments. As for grain growth, there is a different group of properties that are examined such as triple junctions, velocity of grain boundaries, and curvature [109, 110].

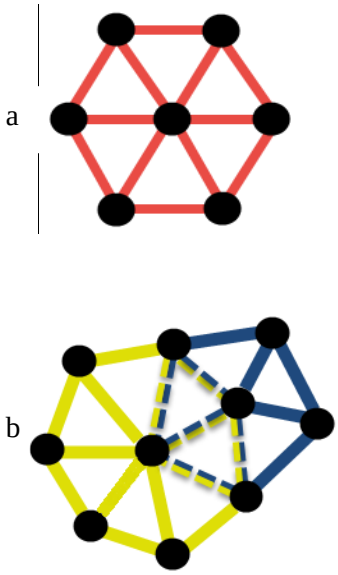


Figure 17: (a) A hexagonal lattice (b) A pentagonal and heptagonal lattice; each forms a disclination

CHAPTER 3 GPU OPTIMIZED MONTE CARLO (GOMC)

This chapter will present an overall description of the GOMC serial and GPU implementations, including the approaches, procedures, software, and hardware. In this chapter, the serial code will also be referred to as the host or CPU code, while the GPU code can be referred to sometimes as the device or parallel code.

3.1 System Description

GOMC is a Gibbs ensemble Monte Carlo simulation engine developed specifically for the simulation of phase equilibria for systems that contain 10,000-100,000+ interaction sites. This simulation engine is designed to simulate different types of molecules that may have different sizes and shapes. Chapter 2 presented Gibbs ensemble and described its structure and simulation flow. To design an open-source framework that can be expanded to simulate more complicated systems, accommodate new I/O formats, and introduce new move types, GOMC is designed using software engineering concepts, such as classes, inheritance, and polymorphism.

3.1.1 GOMC Simulation Flowchart

The flowchart of the GOMC execution pipeline is shown in Figure 18. As seen in the flowchart, some parts are done on the CPU and other parts on the GPU. Mainly, the CPU code is responsible for I/O, molecule selection and move acceptance, initialization, and data communication between the CPU and the GPU. The GPU is responsible for the computationally demanding parts, especially the energy interactions.

3.1.2 Data Structures and System Classes

The GOMC simulation engine architecture follows object-oriented principles, and all main functions and variables are enclosed in classes. CUDA does not support enclosing the global functions in classes, so the GPU functions are written outside the program classes.

To ease the process of data copying from and into the GPU, and to make the threads access data in a coalesced way, the data was stored in arrays in which each entry has no complex struc-

tures like structure or class objects. In other words, data is stored as structures or classes of arrays. For example, to store the X, Y, and Z coordinates of the molecules' particles, three arrays are used to represent the corresponding X, Y, and Z coordinates of each particle in each molecule.

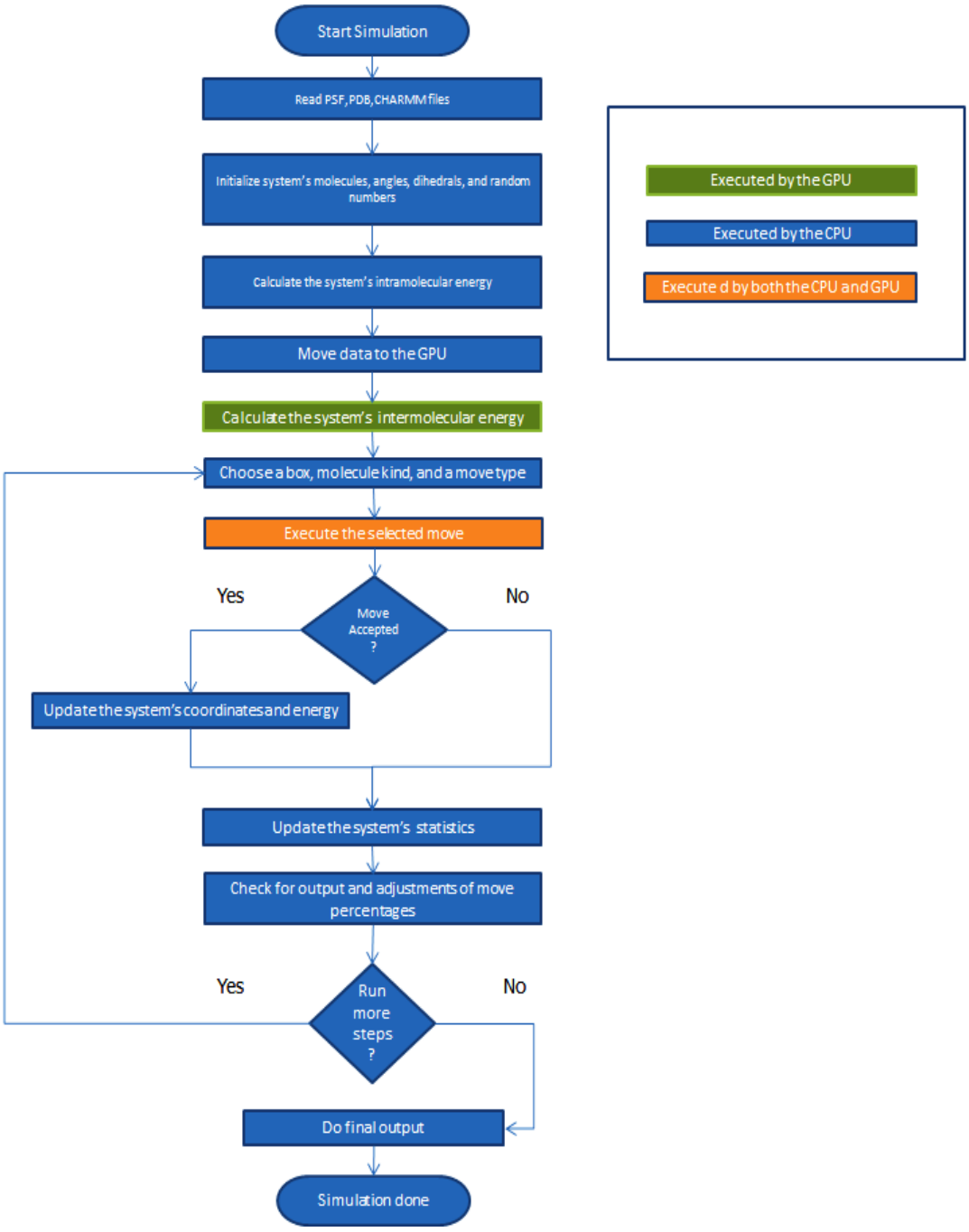


Figure 18: GOMC flowchart

3.1.3 I/O

As an open source software engine, GOMC is designed to use standardized input and output file formats, allowing users to work seamlessly between GOMC and other simulation engines such as NAMD [10], and analysis and visualization tools such as VMD [74]. Figure 19 shows the compatibility between GOMC, NAMD, and VMD.

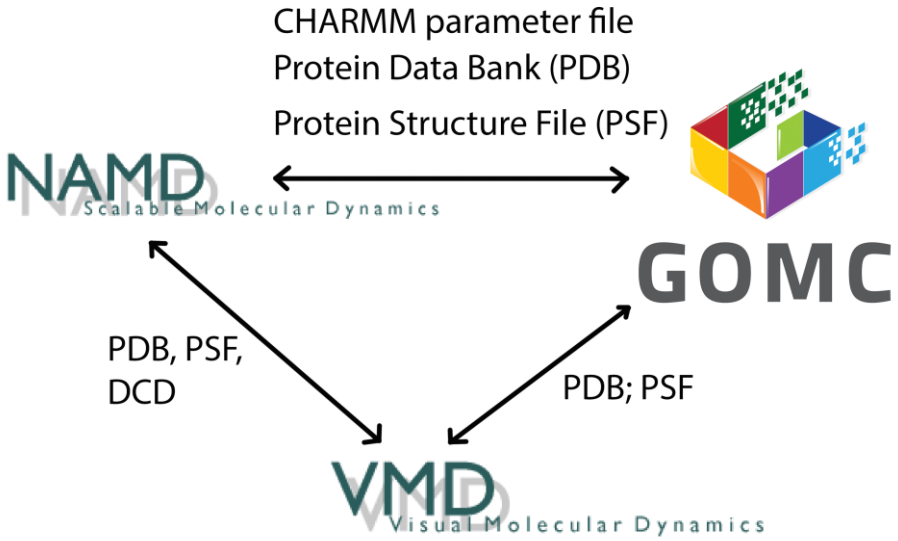


Figure 19: GOMC I/O compatibility with file formats used by other simulation engines

For input, the Protein Structure File (PSF) [75] format describes the structure of the system molecules, such as the bonds, angles, and dihedrals that make up each molecule. Protein Data Bank (PDB) [76] file formats are used to describe the 3D structure of molecules, such as the particles of the molecules and the coordinates of those particles. Application-specific file types are used to specify simulation parameters such as temperature, volume, and number of steps.

3.1.4 Initialization

Molecules' coordinates, angles, dihedrals, random number generators, and system parameters are initialized at the start of the simulation. The coordinates are initialized by reading the PDB, while the PSF files are used to initialize the structure of each molecule, including the angles and dihedrals.

System parameters are read from the input configuration file. The configuration file specifies most of the system variables such as initial box dimensions, move percentages, number of simulation steps, input and output file names, output frequency, temperature, cutoff distance, and random number seed specification.

3.1.5 Random Number Generation

Random numbers have an essential role in the MC method. Random numbers are used to select moves and determine the acceptance of them. There are many algorithms to generate uniform pseudorandom numbers. GOMC uses Mersenne Twister algorithm to generate the different random sequences used in GOMC. Mersenne Twister is one of the most commonly used pseudorandom number generators due to its long period ($2^{19937} – 1$), fast random number generation, and its statistical randomness [33].

In the GPU version of GOMC, the random numbers are also generated on the CPU. When calling functions on the GPU, the required random numbers are passed to the GPU as parameters. In addition, if the random numbers are generated and moved to the GPU, there will be overhead of tracking how many random numbers are used, then when that stream is consumed, the CPU must generate another sequence and move it to the GPU. Although the cuRand package can be used to generate random numbers, this will not generate the same random stream of random numbers on both the serial and GPU versions of GOMC.

3.2 Main System Functionality

This section will focus on describing how energy calculations are done in GOMC and how different Monte Carlo ensembles work in GOMC.

3.2.1 Energy Interactions

Energy interactions are the main functions in the simulation, as they are a key factor in determining the acceptance of moves. Energy interactions may involve all the system molecules, such as when calculating the system's total energy, or a certain molecule interaction, or even a single

particle energy interaction. Figure 20 shows a particle interaction move, where the energy is calculated for the old position within the radial cutoff, and the new energy is calculated for the new position in the radial cutoff. The most computationally intensive energy function is the total system energy function, as it calculates energy interactions of each molecule with all other molecules in the same box within a radial cutoff.

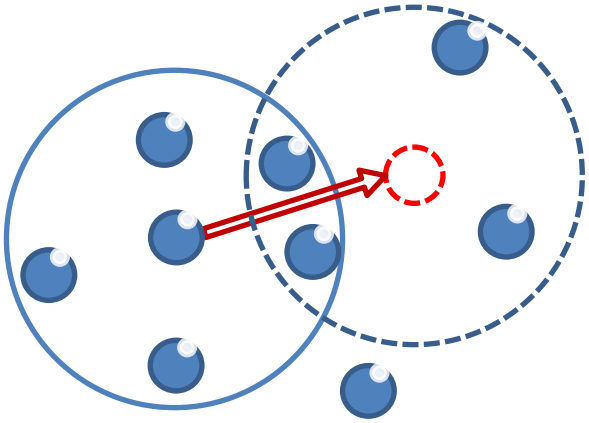


Figure 20: Radial cutoff (r_{cut}) in a displacement move

At the start of the simulation, the system's total energy is calculated by combining the energy of inter-molecular interactions, intra-molecular interactions, and tail corrections. Intra-molecular interactions involve only interactions among particles within the same molecule. This energy calculation depends on the number of particles in the molecule, angles, and dihedrals within the molecule. Because the molecule structure does not change throughout the simulation, unless the molecule is moved to another box and re-grown there, there will be no need to update or calculate the intra-molecular energy frequently.

Inter-molecular energy interactions are the most computationally demanding part of the total energy calculations as it is of order $O(N^2)$. Unique pairwise energy interactions of particles from different molecules are examined by determining first if the pair falls within the cutoff before calculating the Lenard Jones potential. All unique pairwise energy interactions are then added up for each box, then each boxes' energies are combined to give the final system inter-molecular energy.

Inter-molecular interactions are also used in determining the acceptance of the volume inter-change moves, as it is recalculated for the system after scaling the molecules.

When displacing or rotating a molecule within the same box, the old and new inter-molecular energy interactions need to be calculated for that molecule. This energy interaction involves only the selected molecule and other molecules in the same box. Single particle energy interactions are calculated when a molecule is re-grown in the destination box. As the molecule is re-grown one particle at a time, particle interactions are used to decide whether to accept this new location or not.

3.2.2 Ensemble Moves

The Gibbs ensemble in GOMC includes the simulation of four main move types. Based on the specified move percentages, the simulation selects a move at each step. In the displacement move, a box, a molecule kind, and then a molecule of the chosen kind are selected at random. After that, the new location is generated by shifting the particles and the center of mass to a new location within the box. Periodic boundary checks are used to handle coordinates that cross box boundaries.

The rotation move is handled almost the same way as the displacement move, except that when a molecule has only one particle, the rotation move is replaced with a displacement move. The rotation of the molecule coordinates is done by using the center of mass as the pivot point, where rotation matrices are used to do the transformation.

The volume move is the most computationally demanding move as it involves calculating the total system inter-molecular energy interactions for the scaled coordinates. The first step of this move is to select source and destination boxes, and then calculate the new temporary dimensions for the boxes. After that, the coordinates and center of masses are scaled depending on the amount of volume exchanged. Upon the energy calculations for the new volumes, the old and new inter-molecular energies are used to determine the move acceptance. If the move is accepted, the new coordinates are committed as the current coordinates of the system molecules.

The last move type is the molecule transfer. As in displacement and rotation, a box, a molecule kind and a molecule are selected. However, the transfer is done using the configurational bias method to re-grow the molecule in the destination box. The linear CBMC implementation in GOMC is designed to re-grow linear alkanes in the destination box.

The linear CBMC starts by growing the first particle in the destination box. The number of trials for the first particle is specified in the input file. In the source box, the first particle trials will have the first particle location of the moved molecule as one of the trials. For each trial, the weight is calculated for each trial and then used to choose the winning trial position.

After the first particle location is chosen, the second particle trial position will depend on the location of that first particle. The trial positions are found by generating random positions on a sphere that surrounds the first particle. The radius of the sphere is the bond length between the two particles. After the second particle, angles and dihedrals are included in choosing trial positions for the remaining particles in the molecule. After the re-growth is done, the weights of the old and new molecule are used to determine the acceptance of the whole move.

3.3 Brute Force GPU Implementation and Optimizations

The GPU implementation for GOMC is focused on the code parts that are the most computationally intensive. Some parts of the code will remain on the CPU, mainly those parts related to I/O, adjustments, program flow control, and decision making.

3.3.1 Data Load and Movement

To process the data on the GPU, it should be first allocated on the GPU and then copied to that allocated place. At the start of the simulation, the simulation loads all the data necessary to perform the different GPU kernels. Data structures allocated and moved to the GPU include:

1- Molecule coordinates and centers of mass in each box.

2- Force field arrays used in energy interactions.

3- Molecule and particle kinds.

4- Temporary arrays used in scaling for volume moves.

5- Molecule start indices and lookup arrays.

Some data is just passed as parameters when the kernels are launched, such as the random numbers used to shift molecules. Some temporary arrays are necessary to hold the scaled coordinates of molecules when doing a volume move. Data needs to be moved back to the CPU for output depending on the output frequency set in the input file.

The molecule lookup arrays are not moved to the GPU because they introduce a second level of indirection when accessing coordinates, which will slow the performance. The only move that requires the shifting of the coordinates is the molecule transfer move.

3.3.2 Calculating the Total System Inter-Molecular Energy

Energy interactions are the places where the GPU can be used to achieve significant speedups, especially for calculating the total system inter-molecular energy. For the initial version of the total system inter-molecular energy, each thread is responsible for calculating a unique pair of molecule interactions. Although the total number of possible interactions is N^2, there are only $N \times (N-1)/2$ unique pairwise interactions. If N^2 threads are launched, some blocks will have more skipped threads than the other, which can cause an imbalanced workload. To achieve workload balance, the pairwise interactions are re-mapped so that the number of skipped threads will be reduced. Figure 21 shows the re-mapping method.

In the remapping process, a thread goes over each unique pair of particles, and decides if they fall within the radial cutoff. If two particles fall within the cutoff, the Lenard Jones potential is used to calculate the energy and then it is stored in shared memory. The use of shared memory will speed the reduction operation later.

Threads in a thread block are synchronized so that they wait for each other to finish calculating the inter-molecular energy for their assigned pair. Then, the reduction method begins [36]. The reduction process works by having threads in a thread block sum the values from other threads in the same block. Here, half of the threads in the thread block will do the summation, and

then when they are done, half of the threads used in the previous step will do the next step of the summation, and so on. This reduction will continue until there is 64 values to add up. After that, loop unrolling is used to sum the rest. Special cases such as having an odd number of threads are resolved in the code. Atomic operations are used to synchronize this summation across thread blocks. Figure 22 shows how the loop unrolling is done.

The single molecule and single particle energy calculations are done in the same fashion as the total energy interactions, however, they are less computationally intensive as there are O(N) interactions. Other GPU functions that are used include functions for scaling molecule coordinates and calculating the Boltzmann factor on the GPU.

3.3.3 Ensemble Moves

The different moves of the Gibbs ensemble have some or all parts done on the GPU. The selection of the move parameters is done mostly on the CPU, and then they are moved to the GPU when the kernels are launched.

For the displacement move, the kernel has three main tasks; the first one is to do the shifting for the old molecule position. All threads that are launching the kernel will do the energy calculations depending on that shifted molecule, so the coordinates should be stored and be available to each and every thread. To provide fast access to the shifted coordinates, shared memory is used to store them. For each block, the first N threads are used to process the shift of the selected molecule particles, where N is equal to the molecule number. After that, those threads will store the shifted coordinates in shared memory to be used later. While those N threads are shifting coordinates, the rest of the threads in the block will be waiting for the process to complete before moving forward to calculate the energy interactions.

After the shifted coordinates are stored in shared memory, each thread will calculate the pairwise energy interactions for the old and shifted molecules. Next, the energy is summed across all thread blocks using reduction and loop unrolling. Finally, the acceptance phase is done by the first thread in the last executed block. The rotate move is done in the same fashion as the dis-

placement move, except that it uses a different procedure to generate the new molecule trial position.

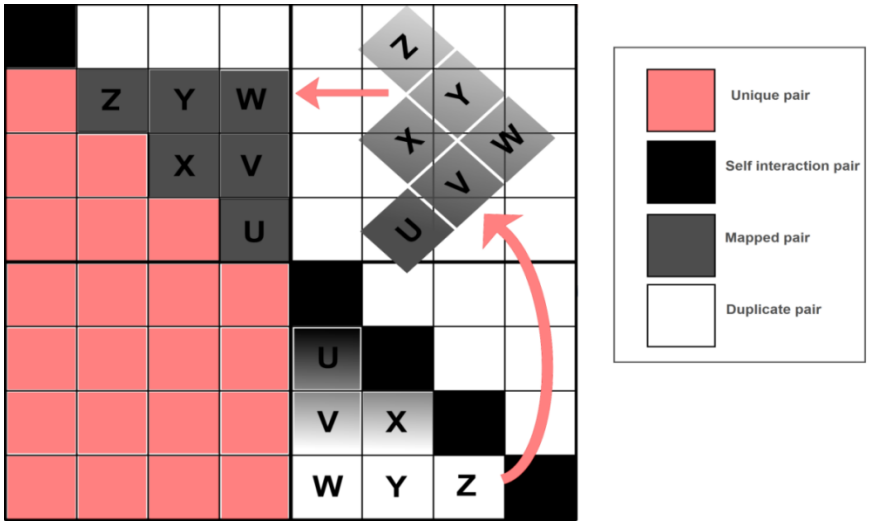


Figure 21: Energy calculation mapping algorithm across threads

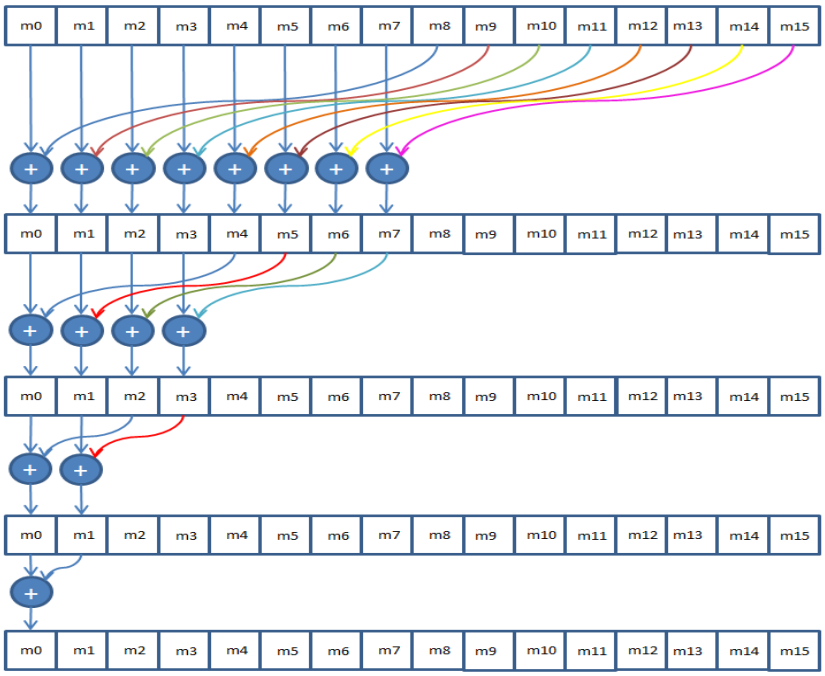


Figure 22: Reduction algorithm for partial summation of the energy in the shared memory

The volume moves is done in a different way than the displacement and rotate moves as it will involve all the system's molecules. The first step of the volume move is the scaling of the mole-

cules in both boxes. Here, the scaling is done on the GPU by launching a kernel for each box and passing the scale to each kernel. As the scaling is independent for each box, each kernel is launched on a separate stream so they can be executed in parallel if there are enough resources on the GPU.

Because the system inter-molecular energy interactions are of order $O(N^2)$, they need more threads than the scaling kernels, which are of order O(N). For that reason, the scaling and the energy interactions are done on a separate kernel. For each box, the inter-molecular energy interactions are calculated by launching a kernel for each one. Using the same technique as in the scaling part, the kernels will be launched on two separate streams.

The last step of determining the volume move acceptance is done on the CPU, after copying back the source and destination box energy interactions. If the move is accepted, the new coordinates are committed by performing a device-to-device copy, where the new coordinates are copied from the temporary arrays to the current coordinates array.

In the molecule transfer move, the CBMC method is used to re-grow the new molecule. Because the molecule is grown particle by particle, the GPU will be used only to calculate the particle energy interactions for the trial position at each stage. To generate the trial positions, the coordinates are copied back from the GPU to the CPU because the CPU does not have the current coordinates of the system, as they are not copied back after performing a displacement, rotate, or a volume move. After the CPU generates the trial positions, the GPU is used to calculate the energy interactions. After finishing the re-growth process, if the move is accepted, the shift is done on the CPU, and then the coordinates are moved to the GPU.

3.4 Cell List Implementations and Optimizations

This section will present the design and implementations for the conventional cell list method and the modified cell list method, called the microcell list, using OpenMP and CUDA. The focus here is on doing single molecule energy calculations and the entire system's energy calculations,

as they are the main overhead in the simulation. The implementation and results are also shown in [96, 101].

3.4.1 Conventional Cell List

As shown in Figure 23, the simulation box is partitioned into square cells, where the cell length of each dimension (S) is greater than or equal to r_{cut}. Coordinates are used to assign particles to cells. A particle can have interactions with other particles that fall within the volume of interest, which will be the current cell and the adjacent 26 neighboring cells. The cell dimension is calculated by maximizing the integer (L/S), where L is the box's dimension length. For instance, if r_{cut} equals 2.5 and L equals 23.9, then S is selected to be 2.656 with 9 cells per dimension. If L < 3 r_{cut}, S will be set to L/3. Cell construction is done at the start of the simulation, or when a volume changes in the volume transfer, as molecules' positions will change.

3.4.1.1 OpenMP Implementation

In the OpenMP implementation of the cell list, the simulation uses the default scheduling. Here, if the program runs with *T* threads and the loop has *N* iterations, thread 0 will process the first *N/T* iterations, and then thread 1 will process the second *N/T* iterations and so on. OpenMP has other modes of scheduling, such as the static mode, where the loop iterations are divided into specified chucks of equal sizes [85, 115]. Threads will process this specified number of iterations until all iterations are done. In the dynamic scheduling mode, iterations are assigned to threads in chunks, and when a thread is done processing the assigned chunk, it will take another one and start processing it [115].

Cell Construction

To provide fast access, the neighboring 26 cells of each cell are cached in a list that can be used later to find particles in the volume of interest. Linked lists are used to store the particle indices of each cell. The reason for using linked lists is that they provide flexibility in terms of adding and removing particles from cells.

Intermolecular force interactions

The calculation of intermolecular force of a particle starts by finding to which cell that particle belongs. Next, the cached list of neighboring particles is accessed to get the indices of all the neighboring 26 cells. After accessing the list of neighboring cells, a list of all particles that belong to those 26 cells, along with the particles in the cell that has the particle of interest, is constructed. This will give us a neighbor list of all particles in the volume of interest. Then, each thread will process the energy interactions of one or more particles in the list with the particle of interest. Each thread will process almost the same number of particles. Before doing the summation of final energy interactions, the threads are synchronized, and then a reduction operation is used to sum up the energy interactions calculated by the OpenMP threads.

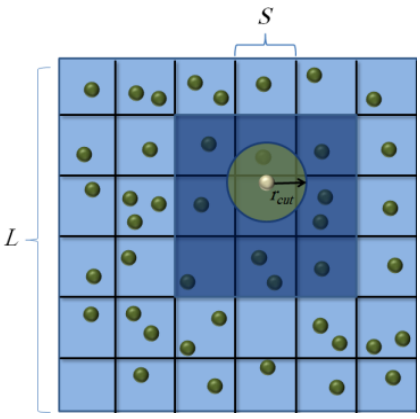


Figure 23: 2D View of a Conventional Cell List

Calculating the system's total intermolecular energy interactions has more overhead, so here each thread will be responsible for calculating the force interactions of particles in one or more cells. For each cell, an OpenMP thread will first construct the neighbor list of particles for a cell, as done when calculating the interactions of a single particle, then use the same list to calculate the interactions for each particle in the selected cell, as they all have the same list.

3.4.1.2 GPU Implementation

Memory access is a major design factor in writing code for GPUs. Using linked lists can lead to an increase of memory reads, thus limiting the speedup. As a result, the arrays are used to store

the adjacency list of the neighboring cells. In addition to limiting the memory access, arrays can provide coalesced memory access, thus reducing memory access overhead.

The size of the arrays that store the list of neighboring cells can be determined, as it will be the number of cells in the box multiplied by 26. However, the size of the array used to store the particle indices of each cell is harder to guess. Therefore, the array's size should be selected to be large enough to accommodate all particles, but some factors such as the radial cutoff, the minimum distance between particles, and the density can help in selecting an appropriate size. In addition, the simulation tests for overflow when assigning particles to cells.

The process of assigning particles to cells, or binning, starts first with creating an adjacency list of neighboring cells. A GPU kernel will be responsible for the creation of this list by assigning one thread to find the 26 adjacent cells for each cell, then storing them. This scheme will eliminate the need for any atomic operations.

Binning of particles is also done on the GPU, where each thread will be responsible for binning one or more particles. Atomic increment is used to increment the counter of particles in the cell to prevent race conditions. Since the use of these atomic operations is limited to the cell initialization operation, and the fact that those atomic operations have become faster with the newer GPU architectures, the overhead is minimal.

Assigning work to blocks and threads depends on many factors, including the move type. To calculate the particle's intermolecular energy, there can be different ways for assigning cells to blocks, such as using one block to process all the 27 cells in the volume of interest, or using 3 blocks, where each block processes 9 adjacent cells, or using 9 blocks, where each block processes 3 cells, and finally, using 27 blocks, where each block processes one cell. Previous experiments [88] show that the best performance is obtained by using 27 blocks. Here, each block will process the interactions of all particles in the assigned cell with the target particle. To calculate the system's intermolecular energy, each block will process the interactions of all particles in its

designated cell with all particles in the center cell. The coordinates of the particles in the center cell are stored in shared memory to reduce memory access time.

The same optimized summation method is used throughout all of GPU implementations in this work [88]. The first phase will sum all the interactions of the threads in that block, which are stored in shared memory, then store them in a global memory location so that they can be accessed by threads from other blocks. Atomic operations are used to achieve synchronization, as there is no explicit synchronization function in CUDA to synchronize thread blocks. The last executing thread block will be responsible for the second phase of summation, which is summing the energy interaction across the thread blocks. Finally, the total sum for the energy interactions is stored in the first thread of the last executing thread block.

3.4.2 Proposed Cell List Algorithm and Optimizations

As many researchers have shown, a conventional cell list outperforms brute force energy calculations [57]; however, there are some drawbacks to the conventional cell list approach, such as encompassing more volume than the volume of interest and the cost of maintaining this list.

In this work, I evaluate a modified cell list approach that divides the simulation box into what is called microcells, where the dimension of each axis of each cell is equal to 1σ, except for boundary cells, which can be smaller. For example, if the volume of the simulation is equal to $60.34\sigma^3$, then the boundary cell along an axis will have a length of 0.34σ. An illustration of the microcell list is shown in Figure 24.

The use of this microcell is based in part on the fact that a cell with a smaller size can accommodate only a few particles. In addition, the use of this fine-grained approach will reduce the processed volume significantly. Another advantage is that the microcell list structure allows a more efficient mapping of the computation to the many-core architecture of the GPU, leading to better load balancing.

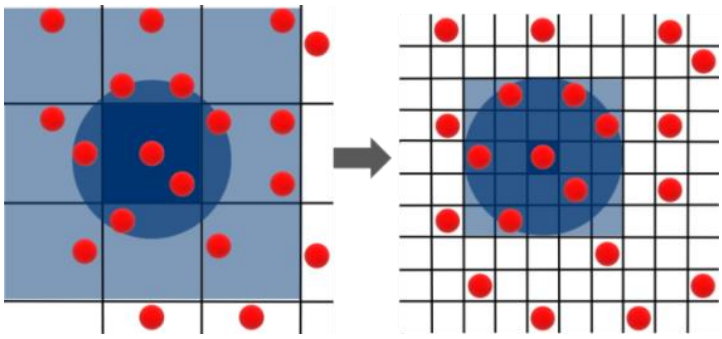


Figure 24: Using a microcell data structure reduces the total volume being processed

3.4.2.1 OpenMP Implementation

The OpenMP implementation of the microcell list uses the same method used for the conventional cell list as described in section 3.4.1.1, as it has good load balancing properties. The main advantage of using the microcell list is that it results in fewer particles in the constructed neighbor list, which should lead to fewer computations.

3.4.2.2 GPU Implementation

The initialization of the microcells is required at the start of the simulation and after a volume transfer is executed. Resizing the cells is one option to address the volume change; however, the GPU is used to initialize the cells, which takes almost negligible time (less than 200 microseconds for a system of 131072 methane molecules using NVIDIA's Tesla K40c). This is achieved by taking directly the integer portion of each coordinate and using it to determine the appropriate cell. For example, if a particle is centered at location [23.5, 12.3, 14.9], then it will be placed in cell [23, 12, 14].

Tracking the contents of each cell requires two arrays; one stores the number of particles in each cell and the other holds the particle indices of the particles in each cell. Instead of storing the particle indices of a cell in consecutive memory locations, they are organized such that the first particle of each cell is stored in the array, followed by the second particle of each cell and so on. This scheme will improve memory coalescing.

In the microcell list implementation, an adjacency list for cell neighbors is not constructed, as it will be huge due to the large number of small cells. Instead, each thread will calculate which cell it is accessing. To map the threads of a thread block to the neighboring cells of a particle, a 3D thread block is defined to ease the mapping process and make it more efficient. Here, threadIdx.x, threadIdx.y, and threadIdx.z are used to map threads to a unique cell in the cube of microcells. An offset is needed to further map the threads to the actual cells in the system. For instance, for a cube of microcells of size 7^3, when processing a particle in cell (7, 6, 5), neighboring cell (4, 3, 2) is assigned to thread (0, 0, 0) and cell (10, 9, 8) is assigned to thread (6, 6, 6). A block of 512 threads is set to be the maximum block size, which can cover up to 8^3 unique microcells for a block. In the case of larger cutoffs, threads will need to iterate over more than one microcell to process all the microcells in the cell cube.

To process the system's intermolecular energy interactions of a box, this work extended the microcell algorithm [88] used to process molecule energy interactions by launching a number of thread blocks equal to the number of single particles to process. However, for large systems, this may result in launching a large number of thread blocks. To overcome this, the system can be configured so that each thread block is responsible for processing the energy interactions of more than one particle. Throughout all this work's experiments, it has been found that the best results are achieved when each thread block processes the cell energy interactions of four particles. The algorithm will only calculate the energy interactions of unique pairs of particles.

3.5 Hybrid Cell List Implementations

This section will present the implementations for the conventional cell list algorithm using two parallel implementations, a hybrid MPI+OpenMP code and a hybrid MPI+CUDA code, and evaluate their scalability across nodes of multicore CPUs, Intel Xeon Phi coprocessors, and GPUs. In addition, the section will present a modified cell list implementation for calculating the system's total intermolecular force interactions that is based on reducing the dimensions of the

cell in order to reduce the whole volume of interest in which the energy interactions are evaluated. The MPI+CUDA version runs on GPU nodes, while the MPI+OpenMP version runs on multicore CPU and Phi processors.

3.5.1 Hybrid MPI+OpenMP Cell List Implementations

This section will go over the hybrid MPI+OpenMP Cell list implementations, showing how the work is distributed among different processing nodes.

3.5.1.1 Conventional Cell List

The master MPI process (id=0) is responsible for constructing the cell list and distributing the workload among the other MPI processes (id≠0). After completing the cell list construction, other MPI processes will start processing their assigned cells, where each node will process approximately the same number of cells. In this work, the total energy calculation functions are calculated on multiple processes, which is mainly used in volume moves, as other moves have much less computation to benefit from distributing the work on other processors.

At each MPI process, which will be running one parallel device or CPU, the calculation of the system's total intermolecular energy interactions of its assigned cells is done using OpenMP. Here, each OpenMP thread will be responsible for calculating the interactions of particles in one or more cells. As all particles in a cell share the same adjacent cells, a neighbor list of all particles in adjacent cells and the current cell is constructed first by combining the particle lists of all those cells in the volume of interest that has 27 cells. After constructing the list, the thread will then iterate over the interactions of each particle in the center cell with all particles in the neighbor list. After all threads finish processing all cells, a reduction operation is used to sum the partial intermolecular energies.

After all MPI processes are done calculating the energy interactions of their assigned cells, an MPI_Gather [105] operation is executed to compute the system's total intermolecular energy. At this point, the master MPI process will sum those results and then use the final sum to determine the acceptance of the volume move.

The initialization and maintenance of the cell list causes major overhead that needs to be addressed in order to use the cell list efficiently. For small systems, this overhead may eclipse the advantages of using the cell list in the first place. In MD, the large volume of computation required for each move may mask this overhead, but in MC, this is still a major concern, especially for small systems.

3.5.1.2 Microcell List

For the OpenMP implementation of the microcell list, the same method is used to calculate the total energy as described in section 3.5.1.1, as it has good load balancing properties. The main advantage of using the microcell list is that it results in fewer particles in the constructed neighbor list, which should lead to less computation.

3.5.2 Hybrid MPI+CUDA Cell List Implementations

As some nodes may have more than one GPU, CUDA streams [6] are used to distribute work among those GPUs, where each stream can execute a kernel asynchronously on different GPUs. The master MPI process is responsible for distributing the work to other MPI processes.

CUDA 5.0 and later releases introduced GPUDirect Remote Direct Memory Access (RDMA) [6], which allows network devices to access data in the GPU device memory without going through the host's memory. After initializing the cells, the master MPI process will distribute the workload on other GPUs in other MPI processes, where each GPU will processes almost the same number of cells.

3.6 Testing and results

In all the results presented in this dissertation, the standard speedup metric is used for performance comparisons of the GPU codes and the serial codes. The total elapsed execution time measurement starts with the input of the parameters, and ends with the final output. The following well-known formula is used to calculate the speedup ratio (S):

$$S = \frac{T_S}{T_P} \tag{3.1}$$

where T_s is the serial code execution time, and T_p is the GPU code execution time.

3.6.1 Cell list testing

In this section, performance evaluations of the cell list implementations are presented on different parallel architectures. The base for all comparisons will be the single core CPU runs that use the brute force method. The brute force CPU code does not use OpenMP. All reported results are the average of three trial runs; the difference in performance among these three runs was negligible. All the energy results from the two cell list methods match on all parallel processors. Table 1 shows the specifications for the CPU, GPU, and the Intel Xeon Phi coprocessor (MIC) parallel hardware used in the experiments. All interactions are done using double precision. The Intel® compiler (icpc v15) for Linux is used to compile the OpenMP code for the MC CPU and the MIC, while CUDA 7.0 was used to compile the CUDA GPU code. All codes are compiled with the 64-bit and the full optimization (-O3) flags.

Table 1: List of major specifications of the parallel processors for the experiments

	CPU	**MIC**	**GPU**
Model	Intel® Xeon ® E5-2680 v2	Intel® Xeon Phi™ 7120P	Nvidia Tesla K40
Micro Architecture	Ivy Bridge (EP)	Knights Corner	Kepler
Number of cores	10 (**Hyper threaded**)	61	2880 (15 ×192)
Clock Frequency	2.85 GHz	1.238 GHz	0.745 GHz

3.6.1.1 Molecule Intermolecular Energy Evaluations

This set of experiments evaluates the performance of the molecule intermolecular Lennard Jones force interactions. For small systems, it can be observed that there is slight or almost no improvement by using the cell list over a single core CPU brute force approach because of the overhead of processing the cells, data reduction, and the low utilization of the parallel hardware.

For large systems, and when testing with different cutoffs, the cell list on all parallel platforms outperforms the single core CPU implementation for both conventional and microcell list as shown in Table 2 and Figure 25. It can also be noticed that the microcell list outperforms the conventional cell list on all parallel platforms, with the multicore CPU achieving the highest speedups while the MIC achieving the lowest. The best results on the multicore CPU and MIC were achieved using 8 threads. Figure 26 shows that the GPU had the highest speedup gain of the microcell list over the conventional cell list approach compared to the other parallel platforms. The reason for this is that only one thread block is needed to do all the calculations, so there is no need for inter-block communication to do data reduction.

Table 2: Runtime results for cell list implementations for molecule intermolecular energy interactions in microseconds for a simulation box size of 65536 methane molecules (one interaction site in each molecule)

	Brute force	**Conv. Cell List**			**Microcell List**		
Radial cutoff	Single CPU core	Multicore CPU	MIC	GPU	Multicore CPU	MIC	GPU
2.5σ	1025	31	456	63	30	430	35
3.0σ	1033	34	680	104	28	433	36
3.5σ	1047	42	747	154	29	440	65
4.0σ	1059	60	757	241	40	442	72

3.6.1.2 System Intermolecular Energy Evaluations

I first experimented with the conventional cell list and the microcell list using a cutoff of 2.5σ. Table 3 shows the execution time in milliseconds for each of the parallel architectures for calculating the system's intermolecular function using the two cell list methods. Figure 27 shows speedup gains over the single core CPU brute force code. The best performance was achieved using 16 threads for the CPU (by using hyper threading) and 128 for the MIC. The density is 0.0177 particles per σ^3 for all systems.

For the multicore CPU implementation, it can be notice from Figure 28 that the microcell list was faster than the conventional up to a box size of 1024 octane molecules. For larger systems, the overhead of having many small cells in the microcell list makes the conventional cell list

more efficient. As for the MIC, it can be noticed that the microcell list performed better than the conventional cell list for all sizes, but less speedup is gained for larger sizes.

Table 3: Runtime results for cell list implementations for system intermolecular energy interactions in milliseconds for octane systems (8 interaction sites, $r_{cut} = 2.5\sigma$)

	Brute force	**Conv. Cell List**			**Microcell List**		
Number Of Mols.	Single CPU core	Multicore CPU	MIC	GPU	Multicore CPU	MIC	GPU
128	9.61	1.323	7.131	0.597	1.055	2.284	1.816
256	29.07	2.871	7.743	0.956	2.175	3.646	3.92
512	97.94	3.61	9.597	1.885	3.579	7.431	7.72
1024	350.99	7.059	12.691	3.48	6.778	9.215	15.063
2048	1330.21	14.12	26.048	6.894	14.684	23.032	29.014
4096	5170.72	22.05	40.506	14.394	26.801	36.447	56.788
8192	20282.57	42.83	71.371	28.5	48.342	63.214	113.04

In the GPU evaluations, the conventional cell list performed better than the microcell list. The reason for this is that the conventional cell list uses far fewer threads to process the volume of interest. In the microcell list GPU code, each block will calculate the intermolecular energy for one or more particles, while in the conventional cell list, each of the 27 blocks processes the interactions of one or more cell. This scheme works fine with the conventional cell list as most of the cells will not be empty.

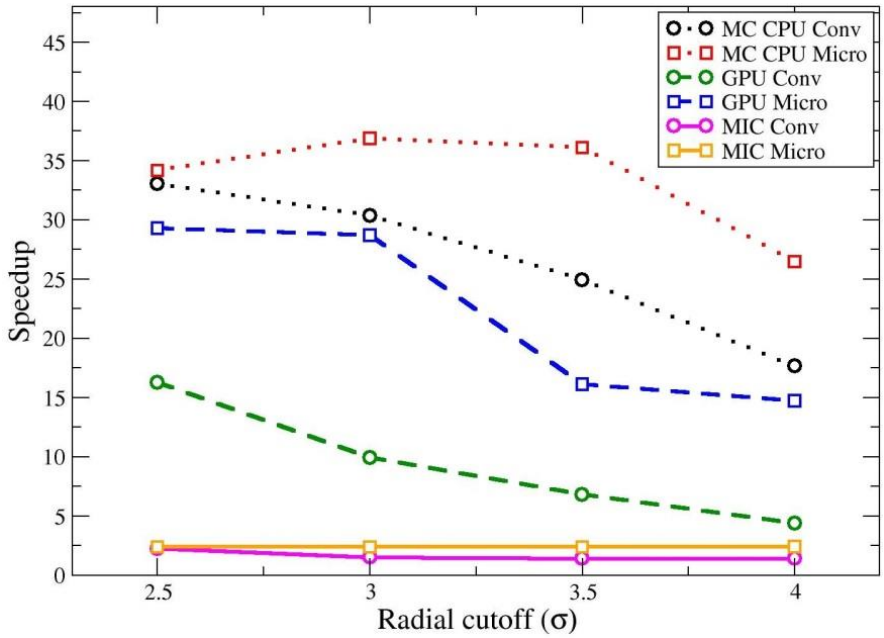


Figure 25: Speedup of cell list implementations over the 1 core CPU brute force implementation

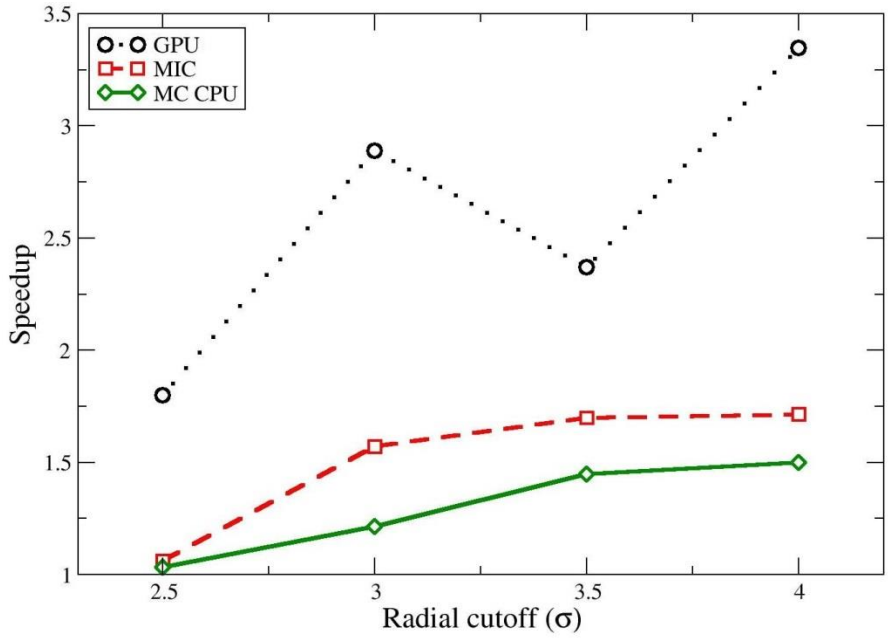


Figure 26: Speedup of microcell list over conv. cell list for molecule intermolecular interactions

I also ran the same tests with a relatively large cutoff of 4σ. Table 4 shows the runtime results in milliseconds for each of the parallel architectures for the system's intermolecular function using the two cell list methods. Figure 29 shows the speedup of the cell list codes on the different parallel platforms over the CPU for a radial cutoff of 4.0σ. Figure 30 shows the speedup of the Microcell list code over the conventional cell list code for all parallel platforms used in the experiments.

Table 4: Runtime results for cell list implementations for System intermolecular energy interactions in milliseconds for octane systems (8 interaction sites). (r_{cut} = 4.0σ)

	Brute force	**Conv. Cell List**			**Microcell List**		
Number Of Mols.	Single CPU core	Multicore CPU	MIC	GPU	Multicore CPU	MIC	GPU
128	21.946	2.879	10.998	1.866	2.125	3.296	3.946
256	58.613	9.24	36.903	2.377	4.398	5.974	8.485
512	150.447	24.228	51.542	3.579	8.705	9.782	16.996
1024	436.459	28.738	77.121	7.776	17.405	15.68	33.717
2048	1455.622	53.862	135.137	16.938	32.427	36.368	64.8
4096	5163.027	110.678	232.186	31.719	59.881	58.21	125.54
8192	19093.304	185.628	351.178	62.678	109.566	84.077	251.27

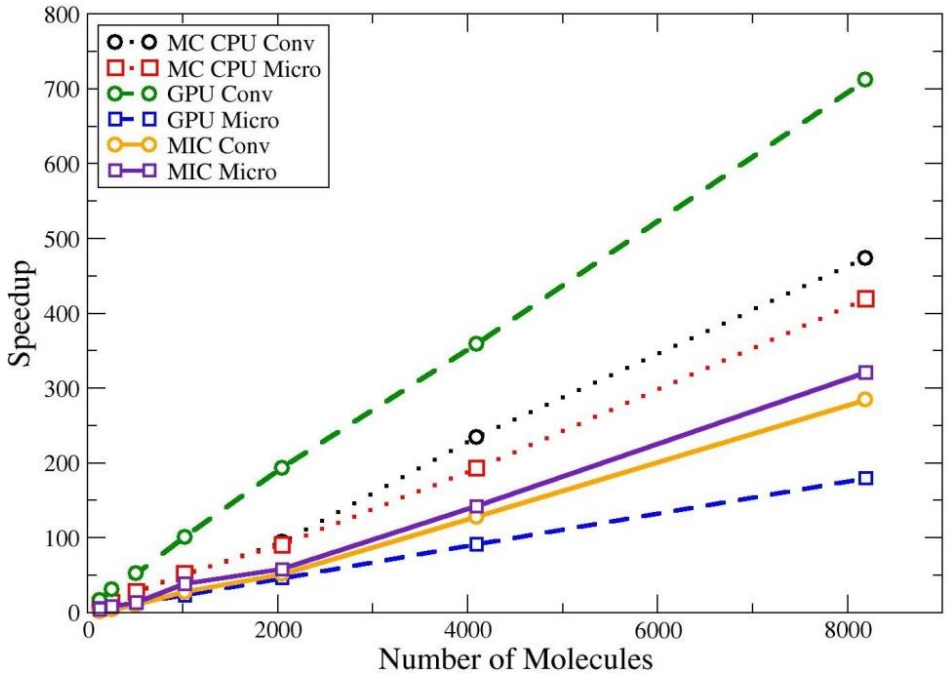


Figure 27: Speedup for cell list implementations for system intermolecular energy interactions over 1 core CPU brute force for octane systems (8 interaction sites, $r_{cut} = 2.5\sigma$)

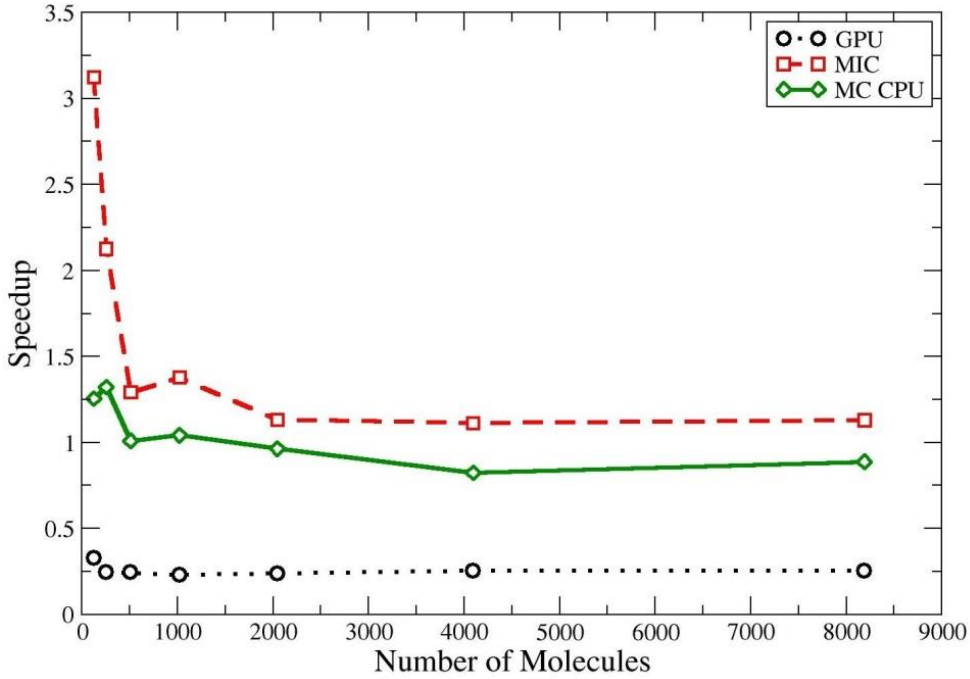


Figure 28: Speedup for microcell list for system intermolecular energy interactions over conventional cell list for octane systems. ($r_{cut} = 2.5\sigma$)

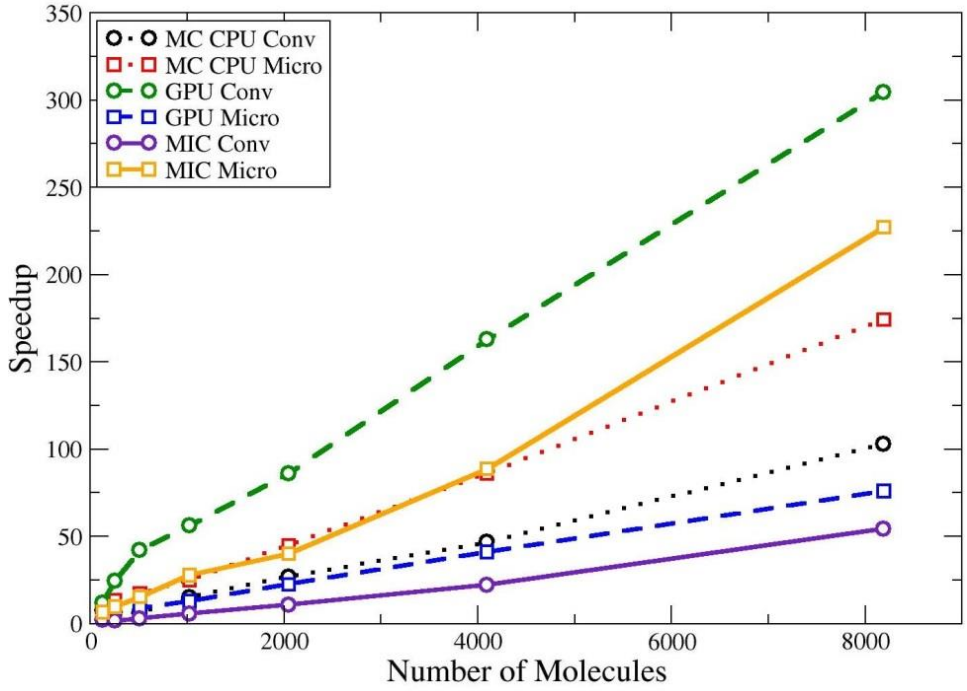


Figure 29: Speedup for cell list codes for system intermolecular energy interactions over CPU brute force for octane systems (8 interaction sites, $r_{cut} = 4.0\sigma$)

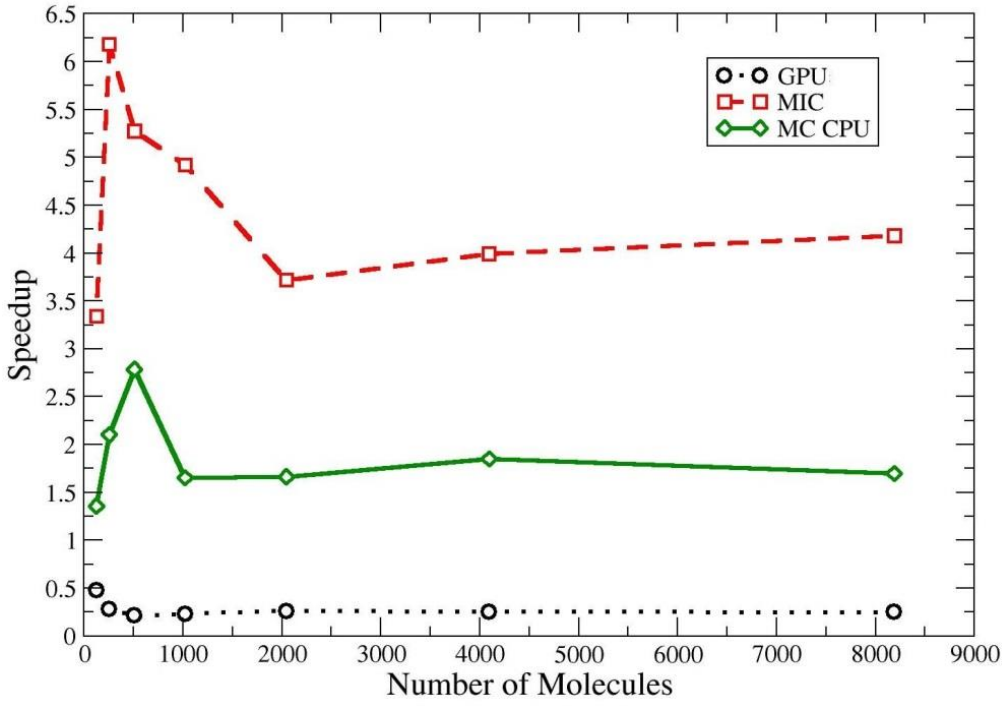


Figure 30: Speedup for microcell list for system intermolecular energy interactions over conventional cell list for octane systems (8 interaction sites, $r_{cut} = 4.0\sigma$)

As observed from Figure 29, the microcell list performs better than the conventional cell list for all box sizes on the multicore CPU and the MIC, and achieves more speedup compared to runs with smaller cutoffs. For the GPU, the conventional cell list still performs better than the microcell list. It can be observed from Figures 27 and 29 that the GPU conventional cell list performed the best compared to the other parallel architectures.

3.6.2 Testing of Hybrid Implementations

This section will show the assessment the performance of the conventional cell list and the microcell list implementations on three different parallel cluster platforms. Three trial runs are made for each presented result; the difference in performance among these three runs is negligible. In addition, the difference between energy results from the two cell list algorithms on all parallel devices is negligible.

The clusters' hardware specifications for the multicore CPU, GPU, and the Phi coprocessor are shown in Table 1. All energy calculations are done using double precision. The cluster used for all the experiments has a total of 8 nodes, where 4 of them have a total of 8 GPUs (2 GPUs on each node). Two of the 8 GPUs are NVIDIA's Tesla K40X, and the other 6 are NVIDIA's Tesla K20X. The other 4 nodes have a total of 8 Phi coprocessors (2 Phi devices on each node). All 8 nodes have the same CPU described in Table 1. In the set of experiments for the GPU and Phi, the difference in execution time of running the codes on devices that are on the same node or different nodes is negligible, such as in the case of using 4 GPUs on two nodes versus using 4 GPUs on 4 different nodes. The cluster used in all experiments has InfiniBand and Gigabit Ethernet networking.

The hybrid MPI+OpenMP implementations are compiled using the Intel compiler (icpc v15) for Linux, while CUDA 6.0 is used to compile the MPI+CUDA GPU code. The MVAPICH2 2.0 [105] CUDA-Aware version of MPI is used to compile the MPI codes, as it has support for GPUDirect. Tests are done using different numbers of devices that will each run an individual MPI process. For the multicore CPU and Phi coprocessor experiments, I also experimented with

different numbers of OpenMP threads. The Phi codes are compiled in native mode. All tests are done using systems of octane molecules; in which each molecule has 8 interaction sites (particles). A radial cutoff of 2.5σ is used in all experiments, and the density for all boxes is 0.0105 particles per σ^3, which equilibrated systems may have.

3.6.2.1 Hybrid MPI+OpenMP

The first set of experiments is designed to evaluate the performance of the hybrid MPI+OpenMP implementations of the conventional cell list and the microcell list using different numbers of devices and OpenMP threads. First, a review the results of running the total intermolecular energy calculations on the multicore CPU nodes will be presented. Figure 32 and Tables 5 and 6 show runtime results using different numbers of OpenMP threads and MPI processes, highlighting the configurations that achieved the best performance for each problem size. OpenMP threads will be referred to as "threads" throughout the results' section.

For the conventional cell list results, observe that increasing the number of devices does not always give the best result, as can be seen from the performance results of problem sizes less than 2048 molecules. This is due mainly to the communication overhead. As the problem size grows, the best performance is achieved when using 8 CPUs with 8 threads running on each CPU. For a problem size of 8192 molecules, using 8 CPUs with 8 threads each achieves more than 26 times speedup over using a single CPU running one thread.

The microcell list results show that for problem sizes that are less than 1024, the best performance is achieved when using 8 CPUs and 4 threads per CPU. Simulation runs on boxes of size 1024 and more show that the best results are achieved using 8 CPUs with 8 threads per CPU. It can be observed that running more than 8 threads per CPU results in slower execution times for most problem sizes and MPI process counts.

Table 5: Multicore CPU runtime results (in milliseconds) for the conventional cell List method

		Number of Devices (MPI Processes)			
# of Mols	**# of threads**	**1**	**2**	**4**	**8**
128	1	10.197	6.431	4.174	2.667
	4	4.832	3.288	2.866	4.695
	8	4.54	4.168	7.128	8.921
256	1	16.54	9.63	6.485	3.705
	4	7.44	4.541	4.414	3.564
	8	9.442	8.799	7.707	10.151
512	1	24.903	13.478	9.09	5.063
	4	9.478	5.553	4.669	3.763
	8	7.59	9.106	6.266	6.616
1024	1	54.26	28.408	16.937	9.04
	4	17.855	9.477	7.06	6.791
	8	10.191	7.707	6.883	7.08
2048	1	107.757	56.221	35.566	19.456
	4	37.051	20.76	12.133	9.633
	8	22.052	17.1	10.112	8.714
4096	1	195.793	100.577	57.98	29.39
	4	58.384	30.727	17.627	10.571
	8	32.992	21.378	20.89	9.681
8192	1	392.983	200.762	111.023	58.115
	4	114.901	58.528	32.116	18.746
	8	60.034	48.159	32.739	15.462

Table 6: Multicore CPU runtime results (in milliseconds) for the microcell list method

		Number of Devices (MPI Processes)			
# of Mols	**# of threads**	**1**	**2**	**4**	**8**
128	1	8.298	5.31	3.75	2.327
	4	4.452	2.94	2.635	2.121
	8	4.413	4.559	8.056	13.249
256	1	15.365	8.718	6.665	3.876
	4	6.975	4.504	3.644	3.041
	8	9.589	8.414	7.424	11.231
512	1	31.051	17.158	11.275	6.474
	4	11.594	7.178	4.458	3.381
	8	13.481	8.612	8.356	6.613
1024	1	59.013	31.789	19.557	11.151
	4	19.59	11.814	6.917	8.768
	8	25.631	20.576	7.574	5.702
2048	1	123.268	63.033	42.033	22.326
	4	41.585	23.591	12.927	10.096
	8	24.168	19.392	12.377	9.537
4096	1	250.367	128.164	75.386	39.086
	4	74.298	40.07	22.711	13.839
	8	45.582	25.972	19.515	12.073
8192	1	502.473	258.901	143.795	74.067
	4	142.367	74.569	38.715	21.429
	8	79.063	44.528	25.352	20.277

Table 7: Intel Xeon Phi runtime results (in milliseconds) for the conventional cell list method

		Number of Devices (MPI Processes)			
# of Mols	**# of threads**	**1**	**2**	**4**	**8**
128	1	110.514	79.09	56.459	46.914
	4	64.478	53.651	45.999	43.25
	8	61.667	54.702	50.14	48.032
	16	60.608	60.968	58.853	54.072
	32	73.494	69.583	68.842	63.571
	64	74.534	74.616	73.549	71.244
256	1	183.572	158.621	84.375	59.888
	4	89.748	105.442	59.493	52.059
	8	70.578	111.739	93.232	60.293
	16	70.893	66.481	63.41	60.708
	32	89.956	101.026	101.563	97.759
	64	97.248	103.352	102.198	170.623
512	1	275.703	199.897	170.566	79.199
	4	121.437	123.406	110.726	93.432
	8	74.242	67.565	62.664	62.277
	16	76.592	68.365	68.478	62.414
	32	84.697	78.897	119.987	104.293
	64	103.157	104.162	170.538	179.84
1024	1	561.946	297.994	192.096	154.121
	4	198.89	115.059	87.585	77.371
	8	166.839	136.731	110.579	73.029
	16	101.129	82.57	72.644	71.217
	32	103.352	92.573	89.081	85.604
	64	109.335	103.768	102.774	99.197
2048	1	1141.71	626.289	389.655	268.059
	4	448.743	229.902	148.137	140.078
	8	236.496	150.271	149.252	87.213
	16	158.216	148.057	131.691	75.855
	32	155.927	137.513	133.755	112.034
	64	131.12	117.991	109.036	173.266
4096	1	2067.47	1125.947	674.253	330.087
	4	688.341	383.991	240.949	170.527
	8	342.847	244.267	177.639	108.021
	16	262.245	196.752	151.39	131.148
	32	206.152	160.072	149.609	129.24
	64	240.78	200.075	193.123	187.18
8192	1	3956.38	2204.27	1243.334	650.784
	4	1215.15	702.108	418.35	248.169
	8	694.48	404.519	268.099	159.101
	16	423.201	295.124	195.426	161.073
	32	305.056	220.267	183.715	131.12
	64	279.508	245.362	199.214	189.868

Table 8: Intel Xeon Phi results (in milliseconds) for the microcell list method

		Number of Devices (MPI Processes)			
# of Mols	**# of threads**	**1**	**2**	**4**	**8**
128	1	78.57	99.786	87.075	40.906
	4	94.487	86.56	81.672	35.149
	8	95.222	92.194	90.022	58.966
	16	96.265	94.495	98.564	102.464
	32	107.951	107.15	116.172	112.156
	64	134.045	139.146	162.684	166.415
256	1	130.368	69.975	50.451	92.286
	4	122.332	96.257	90.551	86.808
	8	62.476	55.808	50.094	60.656
	16	97.698	99.423	103.945	101.238
	32	114.801	114.925	110.845	103.939
	64	139.236	131.478	205.249	164.484
512	1	337.043	213.603	123.167	118.289
	4	162.57	123.609	102.465	93.155
	8	131.634	107.247	99.404	64.758
	16	115.731	107.772	105.751	106.643
	32	121.008	115.252	110.186	108.974
	64	144.65	142.099	169.601	171.258
1024	1	455.556	304.862	226.032	150.338
	4	223.035	155.237	127.497	101.342
	8	163.451	123.899	113.641	95.872
	16	144.317	118.828	115.728	103.704
	32	133.591	119.158	116.312	115.485
	64	143.779	160.709	178.434	150.984
2048	1	1075.143	575.527	409.94	250.004
	4	404.38	266.414	181.949	135.412
	8	253.697	182.867	129.88	120.353
	16	188.236	146.471	130.888	119.556
	32	160.616	151.538	143.406	114.82
	64	167.689	188.399	184.024	172.811
4096	1	1989.131	1063.011	642.327	373.816
	4	645.778	373.279	238.608	160.693
	8	392.485	246.706	180.403	152.292
	16	266.783	201.959	156.7	142.564
	32	228.128	197.778	144.881	139.714
	64	215.566	231.837	210.317	219.453
8192	1	3957.847	2114.066	1189.278	664.164
	4	1181.601	669.13	388.684	239.494
	8	656.358	390.26	244.342	151.171
	16	385.959	265.922	199.024	162.898
	32	263.511	243.613	195.732	155.327
	64	284.107	252.832	249.387	240.962

When compared to the conventional cell list results, it can observed that the best configurations of MPI processes and threads for the microcell list are faster than the best configurations for the conventional cell list for problem sizes up to 1024. However, the difference in performance is slight, achieving at most a speedup of 19% for the problem size of 1024 molecules per box. For larger problem sizes, the overhead of maintaining and processing many small empty cells makes conventional cell lists perform better.

The second set of the hybrid MPI+OpenMP cell list implementations is done on Xeon Phi accelerators. There are many ways to program those accelerators. Here, I tried running the MPI+OpenMP code on those accelerators to observe the performance compared to multi core CPUs. Figure 31 and Tables 7 and 8 show the execution times for these two cell list algorithms.

For the conventional cell list, note that the best performance for the first two problem sizes is achieved when using 8 Phi processors with 4 threads each. As the problem size grows, the best performance is achieved when using 8 Phi devices and 8 and 16 threads for problem sizes 1024 and 2048 respectively, then for the remaining sizes, the best performance is achieved when using 8 Phi devices and 32 threads. Also note that for the largest problem size, 8192, the simulation achieved more than 30 times speedup for the best performing configuration over using one Phi coprocessor with 1 thread.

Microcell list results show that for the first two problem sizes, using 8 Phi devices running 8 threads each will actually be slower than just using one Phi running one thread. For problem sizes that are larger than 1024, the best performance is achieved using 8 Phi coprocessors with 32 threads each. For the largest problem size of 8192, the best configuration achieved more than 25 times speedup over using a single Phi device running one thread.

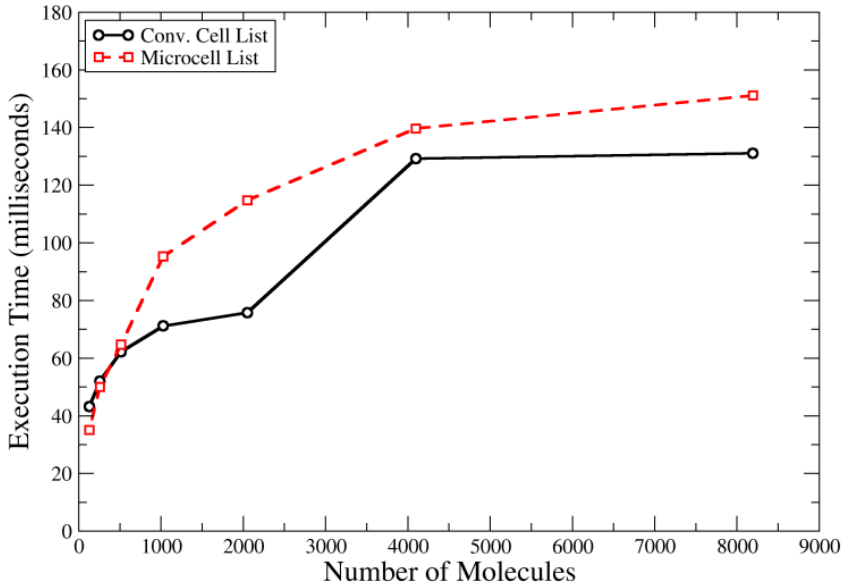


Figure 31: Execution time in milliseconds for the best configurations of the conventional and microcell lists when running on a Phi cluster

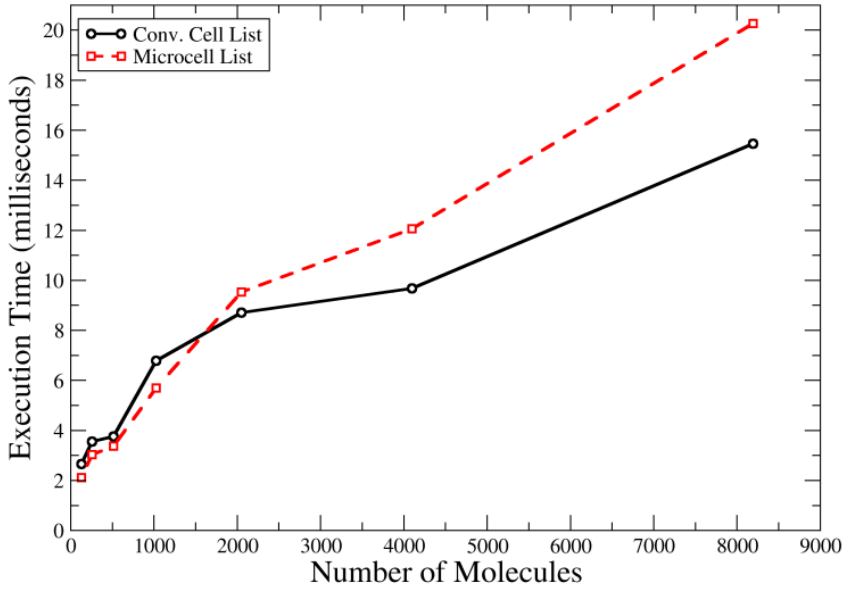


Figure 32: Execution Time in milliseconds for the best configurations of the conventional and microcell lists when running on a multicore CPU cluster

3.6.2.2 Hybrid MPI+CUDA

This section presents the results of running the two different cell list implementations on GPUs. Table 9 shows the conventional cell list runtime results. Note that for problem sizes of 128 and 256, using two GPUs gave the best performance results. As the problem size grows, the best execution time was achieved when using all 8 GPUs. For the largest problem size of 8192, using

eight GPUs achieved almost 3.45 times speedup over using a single GPU. Table 10 shows the runtime results for the microcell list. It can be seen that with the exception of the smallest problem size, the best performance results are achieved using all 8 GPUs. It can also be noticed that for problem sizes 128, 256, and 512, the microcell list code is faster than the execution times of the best configurations in the conventional cell list runs. As the problem size grows, having many small cells will have more overhead compared to the overhead of having larger cells in the conventional cell list approach. Figure 33 shows run time for the best configurations for the two cell list algorithms on the GPU.

Table 9: GPU runtime results (in milliseconds) for the conventional cell list method

	Number of Devices (MPI Processes)			
Number of Mols	**1**	**2**	**4**	**8**
128	1.165	1.022	1.149	1.557
256	1.658	1.258	1.395	1.495
512	2.974	1.94	1.8	1.736
1024	5.291	3.046	2.763	2.291
2048	10.268	5.872	4.266	4.018
4096	20.913	11.269	7.574	6.548
8192	41.249	21.317	13.408	11.88

Table 10: GPU runtime results (in milliseconds) for the microcell list method

	Number of Devices (MPI Processes)			
Number of Mols	**1**	**2**	**4**	**8**
128	1.605	1.445	0.75	0.91
256	3.279	2.021	1.795	1.209
512	6.469	2.768	1.931	1.53
1024	12.652	5.368	3.076	2.591
2048	23.439	9.873	5.667	4.404
4096	46.066	19.412	16.49	10.384
8192	91.712	38.355	16.929	16.764

For large systems, the processing of many small microcells requires launching many threads. In the case of sparse systems, most of those threads will do little work. In the conventional cell list, larger systems will require launching fewer threads compared to the microcell list, where those threads will have work to do, as larger cells will have particles to process, even in the case of sparse systems.

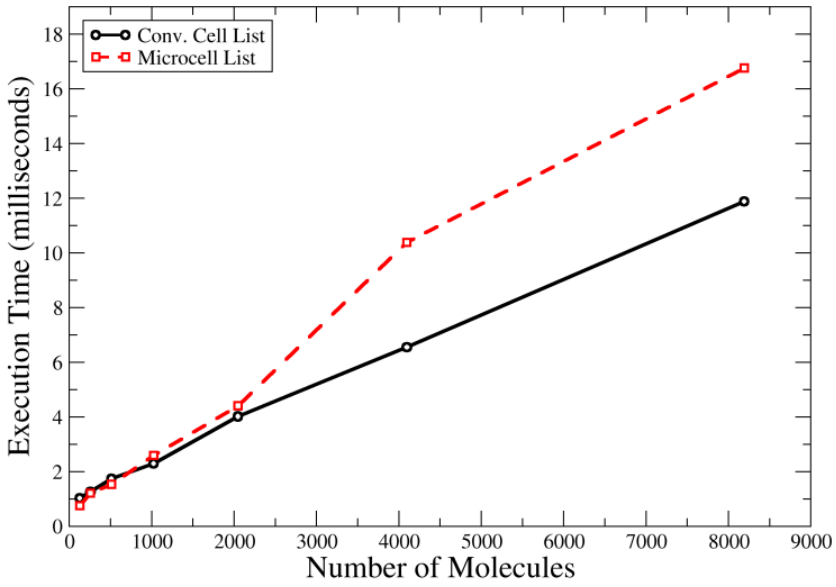


Figure 33: Execution time in milliseconds for the best configurations of the conventional and microcell lists when running on a GPU cluster

3.6.3.3 Parallel Platform comparisons

When all three parallel platforms are compared, multicore CPUs, Phi coprocessors, and GPUs, it can be observed that the best performance is achieved using GPUs. Tables 11 and 12 show speedups of the GPU platform over the Phi and multicore CPU clusters for both the conventional and the microcell lists. It can be noticed that the Phi accelerators that ran the MPI+OpenMP code are the slowest of all three platforms. Major factors that contributed to the GPU's best performance include using of the GPUDirect technology that reduces the communication overhead between GPUs, and having good mapping of the work load to the GPU's threads and thread blocks.

Table 11: Speedup of the GPU runtimes over the multicore CPU runtimes for the best configurations at each problem size

Number of Mols	Conv. Cell list	Microcell List
128	2.610	2.828
256	2.833	2.515
512	2.168	2.210
1024	2.964	2.201
2048	2.169	2.166
4096	1.478	1.163
8192	1.302	1.210

Table 12: Speedup of the GPU runtimes over the Intel Xeon Phi coprocessor runtimes for the best configurations at each problem size

Number of Mols	Conv. Cell list	Microcell List
128	42.319	46.865
256	41.382	41.434
512	35.874	42.325
1024	31.086	37.002
2048	18.879	26.072
4096	19.737	13.455
8192	11.037	9.018

3.7 Summary

This chapter focused on presenting the GOMC simulation's design and challenges that were faced when optimizing the code to run on the GPU. The optimizations included the remapping of energy interactions to produce a more balanced workload for the brute force GPU code. In addition, the chapter went over the cell list code implementations and optimizations for the OpenMP and the GPU code. The optimizations for the GPU microcell list code show that efficiently mapping the problem to the GPU hardware can lead to better performance, such as using 3D thread blocks and only using one thread block to eliminate the need for block synchronization. While the microcell list achieved better performance compared to the conventional cell list for the particle interactions, the need to launch many threads to calculate the system's total energy made the microcell list slower.

CHAPTER 4 PFC GRAIN GROWTH

This chapter will present the overall description of the PFC grain growth implementation. It explains the approach, procedures, software, and hardware used to realize the implementation. In this chapter, the serial code may also be referred as the host or CPU code, while the GPU code will be referred to sometimes as the device or parallel code.

4.1 PFC GPU Implementation

This section describes the PFC simulation and properties that are ported to the GPU and how they were implemented. The grain growth simulation for the PFC model simulates 2D hexagonal crystals. By using the GPU, large systems can run for longer simulated time periods in a reasonable amount of time. The CPU code is responsible for initialization, I/O, and launching kernels. Because the PFC mainly processes the ψ array and transforms it throughout the simulations steps, the ψ array will not be copied back to the CPU except when output is performed. The flowchart of the simulation is shown in Figure 34.

4.1.1 System Functionality

The most computationally intensive parts of the PFC simulation are the processing of the ψ array and the other auxiliary arrays used in the simulation, and the fast Fourier transforms (FFTs) [59, 72]. FFTs are used to compute discrete Fourier transforms (DFTs). The ψ array and some arrays are of size $lx \times ly$, which is demanding to compute since they are processed several times in each step. Different kernels are written to handle different processing operations on the ψ and auxiliary arrays. A thread can process one or more positions. In each time step, there are several FFT calls, some are forward, in which they transform double to complex arrays, and others are backward.

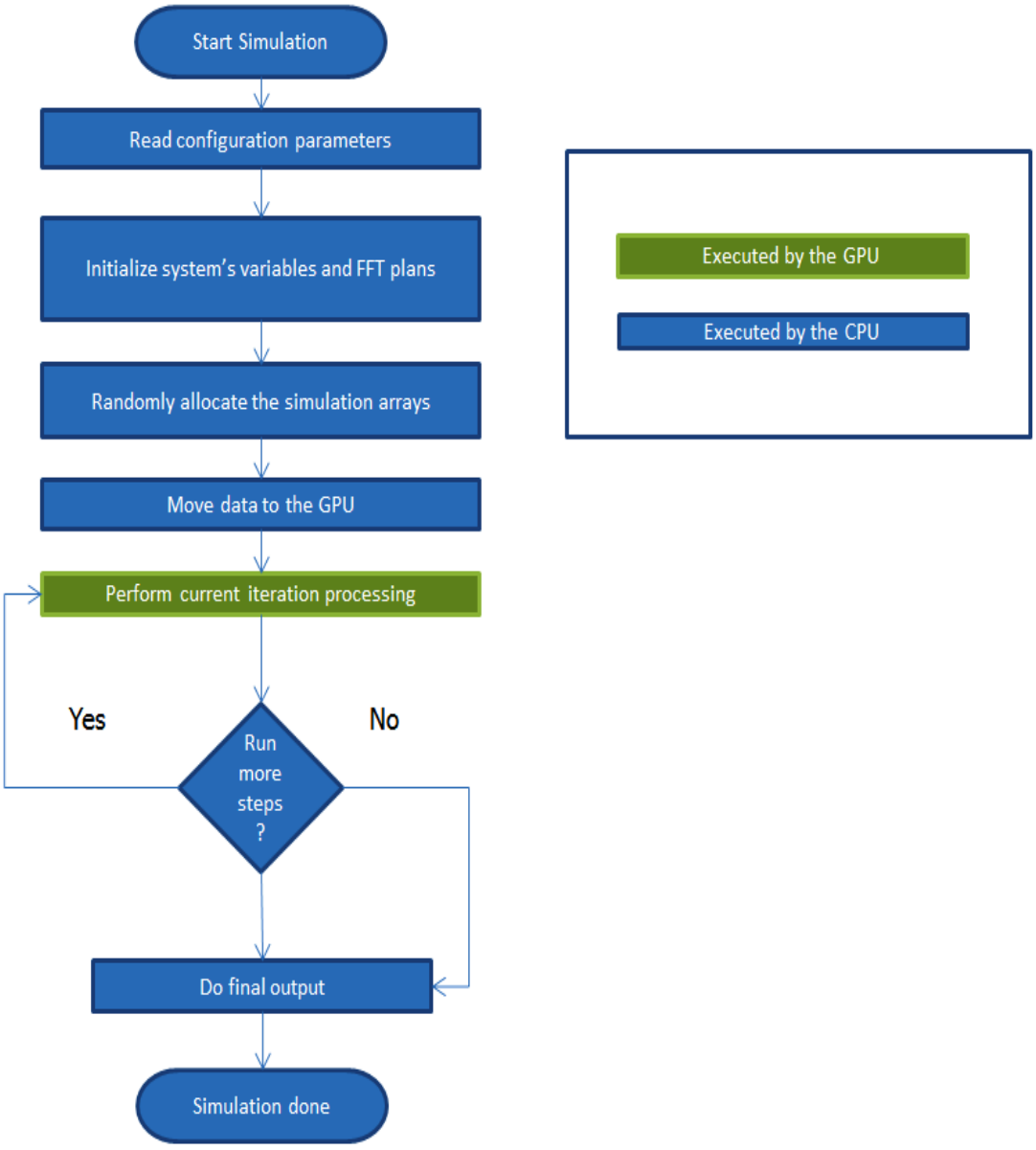


Figure 34: PFC system flowchart

For the serial code, the FFTW library [72] is used to implement the FFT functions. FFTW is the fastest publically available implementation of the FFT functions [72]. The cuFFT library, available from NVIDIA, is used for the GPU code, which is the fastest GPU FFT library [59].

4.1.2 Orientational Correlation Function (g_6)

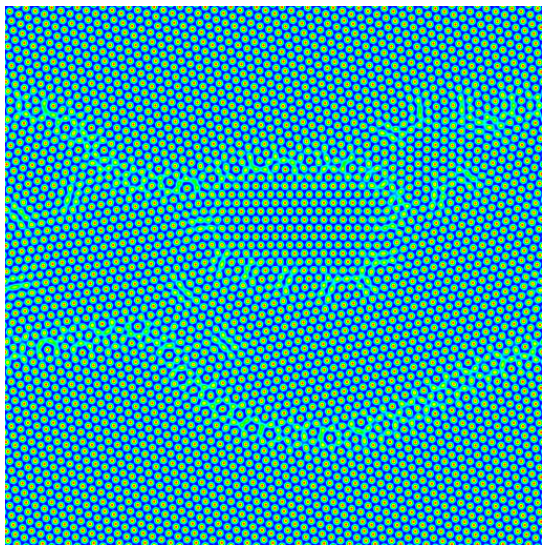
The g_6 function can be calculated in different ways. One way is described in equation 4.1

$$\boldsymbol{g_6(r) = \langle cos\ \left(6\left(\theta_i - \theta_j\right)\right)\rangle} \qquad (4.1)$$

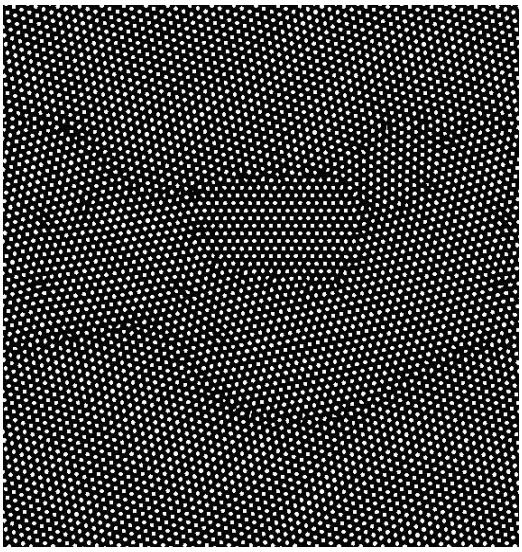
where r is the distance between two atoms, and θ is the local lattice orientation [14, 26, 35].

To calculate the g_6 function on the GPU, the following steps are executed:

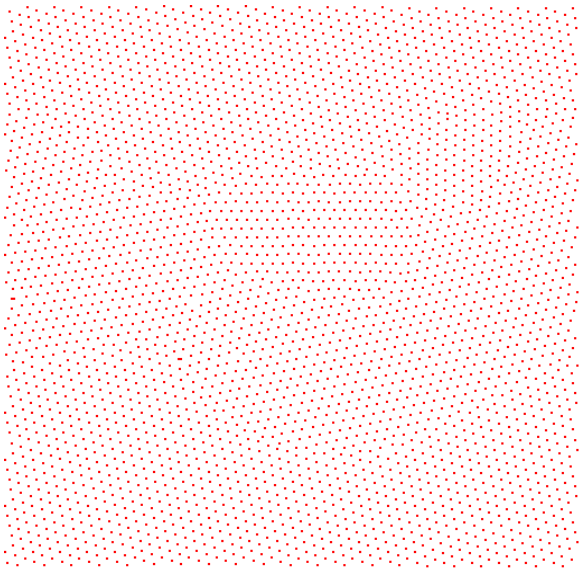
1. The ψ array values are examined to detect atoms. When ψ reaches a local maxima value in any spatial point that means this location represents an atom. One way to do the detection is by using the connected component labeling algorithm [81]. First, a threshold is set for the ψ values, and thereby excludes all ψ values that represent vacancies. Within a connected component, the local maximum value of ψ is detected and marked as an atom. Atoms are then stored in a list sorted by the x-coordinate of its position, because they are detected in a row-by-row fashion. Figure 35 gives an illustration of the atom detection method.
2. Using Delaunay triangulations, the nearest neighbors are located for each atom. Delaunay triangulations are the dual of Voronoi diagrams [29, 30]. A Voronoi diagram for a group of points is constructed of what is called Voronoi cells. For each point, there will be a corresponding cell that contains all the points that are the closest to that point. From the Voronoi diagram, the Delaunay triangulations can be extracted and used to find the closest neighbors for each atom.



(a)



(b)



(c)

Figure 35: (a) ψ array plot representation using HDF (b) Regions (white) generated by the connected component algorithm (c) Final atom representation

The serial code implementation used a Delaunay triangulation algorithm called Triangle, which is considered the fastest serial implementation for the Delaunay triangulation [29, 30, 82]. For the GPU code, the GPU-DT library is used for 2D Delaunay triangulations, which is the fastest 2D Delaunay triangulation on the GPU [29, 30]. In all simulations, each atom will represent the center of a hexagonal lattice as shown in Figure 36.

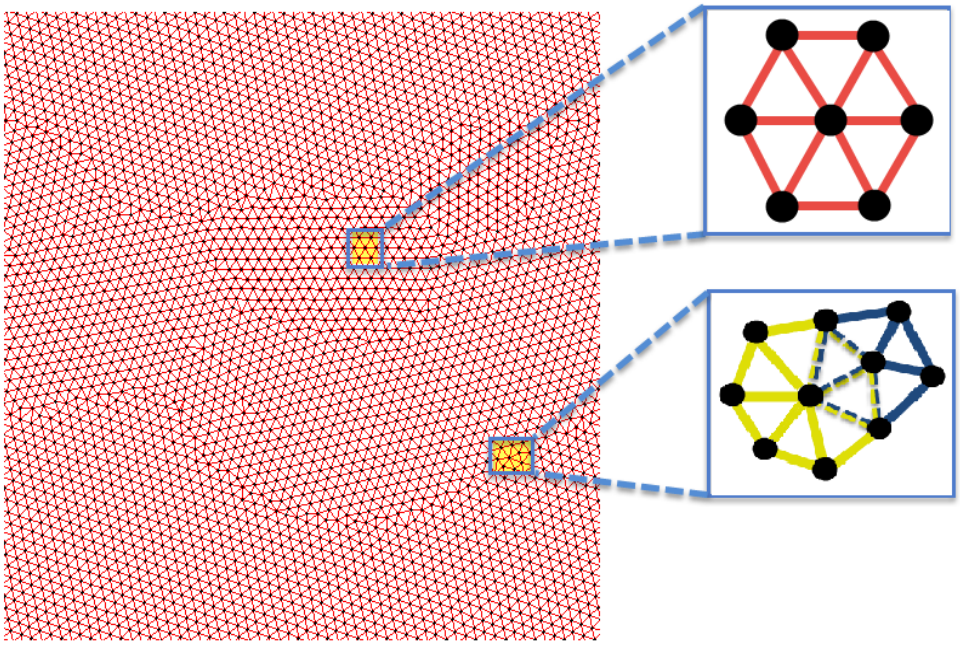


Figure 36: Delaunay triangulations for detected atoms. The figure also shows a hexagonal lattice and two disclinations

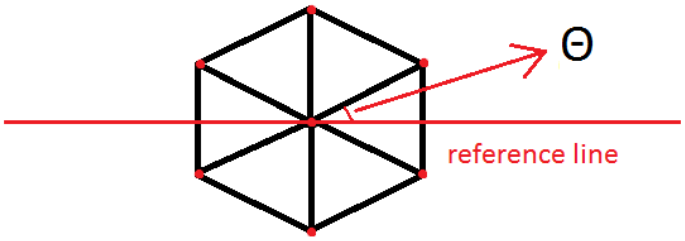


Figure 37: (θ) angle for a hexagonal lattice computed against a reference line

3. The local lattice orientation (θ) [16] is calculated for each atom. The θ angle is calculated by determing how the hexagonal lattice deviates from the reference horizontal line inserted at the start of the simulation. Figure 37 shows an example. The θ angle gives an idea of the orientation of hexagons in grains. Hexagons in the same grain should have almost the same orientation.
4. For each atom, the circular average of θ angles is to be calculated next**.** The circular average can be calculated over different radii by constructing a ring area of radii $r+dr/2$ and $r-dr/2$, where r is equal to $dr \times k$, dr is equal to $4\pi/\sqrt{3}$, and k represents the order number for the circular average. For example, when k is equal to 1, this means that it will be the first search ring surrounding the atom. The sorted list of atoms can be useful for finding atoms that fall within the search ring. To calculate the circular average of θ angles, a tangent square is constructed over the search ring to mark the search area. After that, each thread will perform a binary search to locate the starting and finishing atoms in the atom list by using the upper left and lower right corner x-coordinate, and thereby establishing a search region that is constructed of a strip of rows that has within them the search area. A thread can process one or more atoms. Most g_6 calculations require the calculation to be done for several k values, which means that almost all the points inside the search square of the largest k will be included for some value of k. Figure 38 gives an illustration of the search method.
5. The last step is to calculate the arithmetic mean of the circular averages for every specified radii. This procedure of calculating the g_6 is used for the GPU and serial codes. Disclinations can be counted using the first two steps of the procedure used in the g_6 calculation.

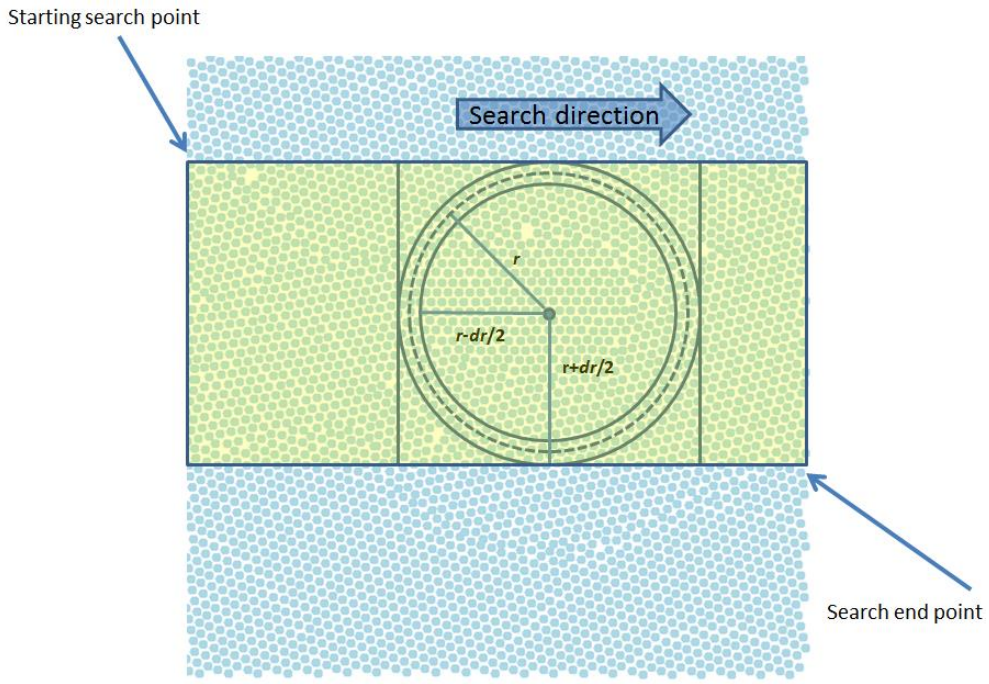


Figure 38: Circular average mechanism

4.1.3 Correlation Length

The correlation length function specifies how g_6 varies over time [109, 110]. To determine the correlation length, we need to pick r where the g_6 is equal to e^{-1}. To do that, we keep calculating g_6 for increasing values of k until we get two values of $\log(g_6)$ that are $\log(g_6(k)) \leq -1 \leq \log(g_6(k+1))$. After that, we do a linear interpolation to get the correlation length. Figure 39 shows the result of the correlation length over time for a 512^2 system size.

4.2 PFC Other Properties

This section will go over some of the properties done in this work that are not yet ported to the GPU. The algorithms specified here will be used as a guide to the planned GPU implementation.

4.2.1 Number and Density of Disclinations and Dislocations

To calculate the disclinations and dislocations, we first determine the atoms that have 5 or 7 neighbors using the same procedure that is used in calculating the g_6. After determining those atoms, we can determine the number of 5 and 7 disclinations, which is just a matter of counting

them. To determine the number of dislocations, we find the pairs of 5 and 7 disclinations and count each pair as one dislocation. Figures 40 and 41 show the count and density of disclinations and dislocations for a system of 512^2 over time.

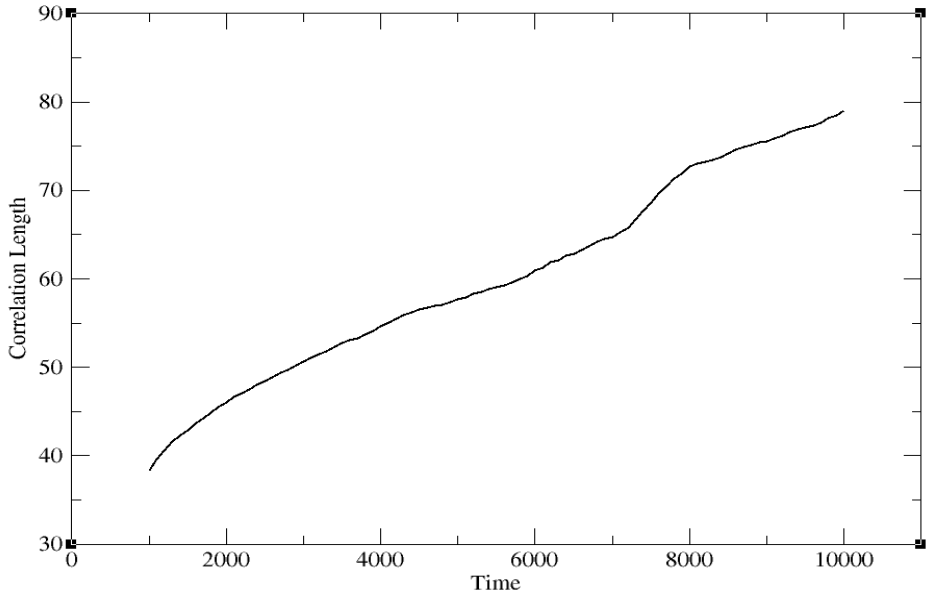


Figure 39: Correlation length

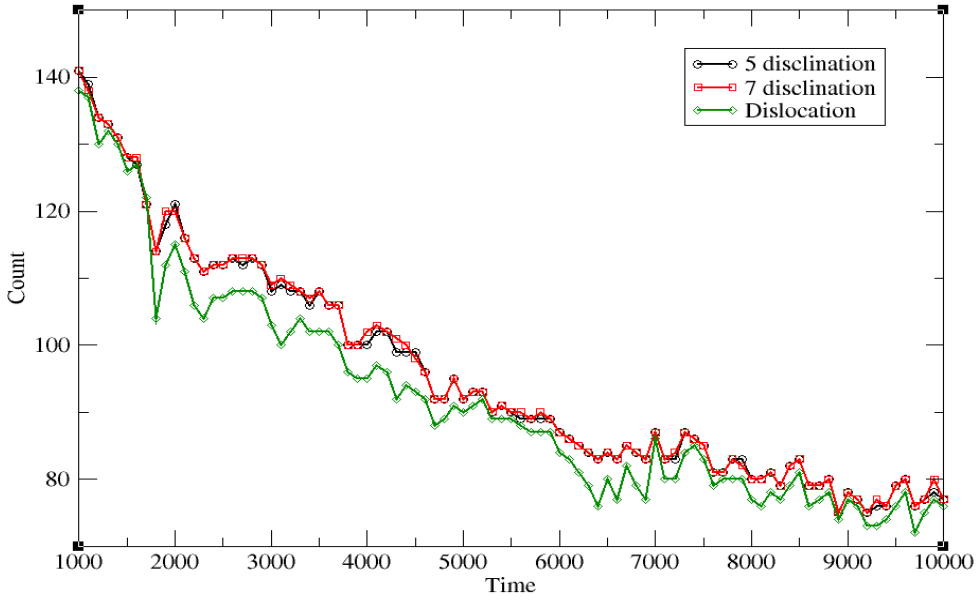


Figure 40: Dislocation and disclination count

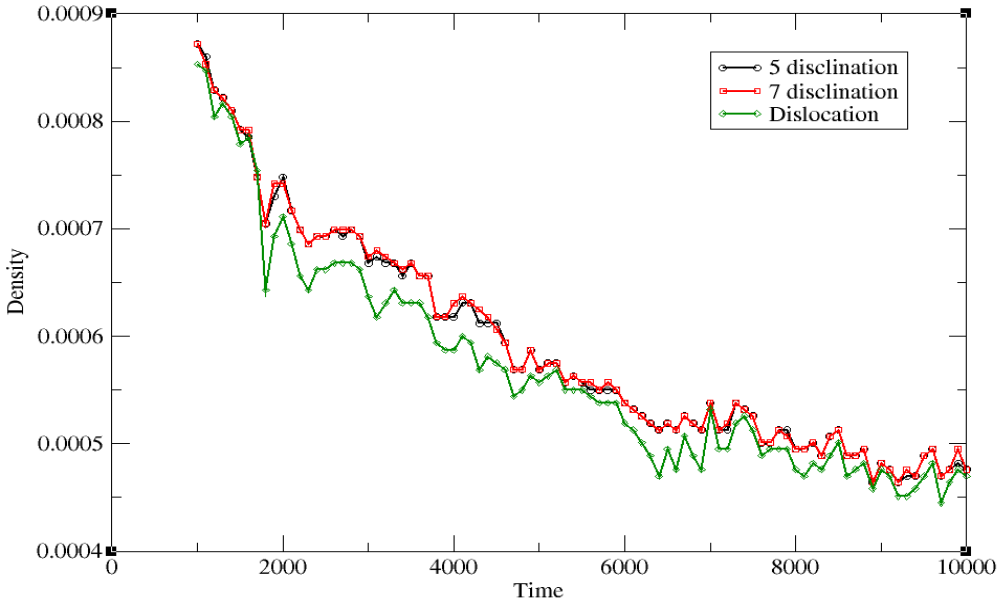


Figure 41: Dislocation and disclination density

4.2.2 Structure Factor

Code listing 1 shows the code that is used to calculate the structure factor. In crystallography, the structure factor is used to describe how a crystal structure scatters radiation and reflects it [108]. Figure 42 shows the structure factor calculated at different time steps for a system of 512^2.

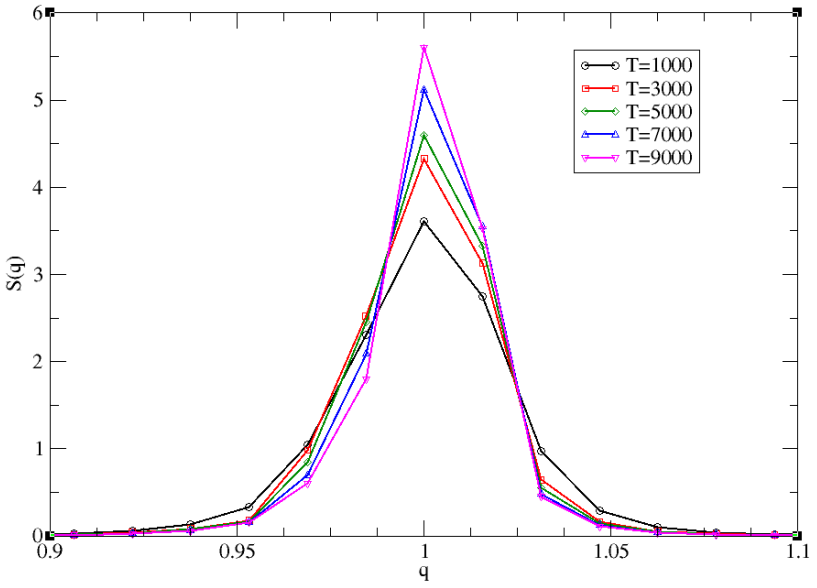


Figure 42: structure factor

```
void structure_factor(double q0, double time, double dx, double dy)
{
int i=0, j=0,n=0 ;
 int w = DATA_DIMENSION/2+1;
for ( i=0;i< lx; i++)
for ( j=0 ; j< ly/2+1 ; j++)
{
sq1[i*w+j] = pow(cabs ( psiq[i*w+j]) ,2);
sq1[i*w+j] = sq1[i*w+j] * scale2d_b;
}
for (i=0;i< lx/2; i++)
{
Sq[i] = 0.0;
total [i] = 0.0;
 }
  for (i=1;i<lx;i++)
   for (j=1;j<ly/2+1;j++)
      for (n=0;n<lx/2;n++)
                {
                if  ((sqrt(q2[i*w+j])>=dq1[n])  &&  (sqrt(q2  [i*w+j])
                <dq2[n]))
                          {
                           Sq[n]=Sq[n]+sq1[i*w+j];
                           total[n]=total[n]+1;
                          }
                 }
for (n=0;n<lx/2;n++)
   if (total[n] >0.0)
      Sq[n] = Sq[n]/total[n];
}
```

Code Listing 1: Structure factor code

As it can be noticed from the code listing, the for loops can be easily ported to the GPU by creating kernels where each thread processes one or more entry of the arrays.

4.2.3 Moments

To calculate the moments for the structure factor, we first calculate the structure factor, and then use the code segment in code listings 2 and 3 to calculate different moments [110]. Figures 43, 44, and 45 show different moment results for a system of size 512^2. Again, the code listings show that the porting to the GPU is straightforward.

```
for (n=0;n<lx/2;n++)
  {   sumSq=sumSq+Sq[n];
      m1=m1+qq[n]*Sq[n] ;
      m2=m2+pow(qq[n],2)*Sq[n];
      m3=m3+pow(qq[n],3)*Sq[n];
      m1_0=m1_0+fabs(qq[n]-q0)*Sq[n];
      m2_0=m2_0+pow((qq[n]-q0),2)*Sq[n];
      m3_0=m3_0+pow(fabs(qq[n]-q0),3)*Sq[n];
  }

if(sumSq > .00001)
  {  m1=m1/sumSq;
     m2=m2/sumSq;
     m3=m3/sumSq;
     m1_0=m1_0/sumSq;
     m2_0=m2_0/sumSq;
     m3_0=m3_0/sumSq;
  }
```

Code Listing 2: Moments and Moments_0 code

```
for (n=0;n< lx/2;n++)
   {
     sumSq=sumSq+Sq[n];
     m1_x=m1_x+fabs(qq[n]-qq[qIndex])*Sq[n];
     m2_x=m2_x+pow((qq[n]-qq[qIndex]),2)*Sq[n];
     m3_x=m3_x+pow(fabs(qq[n]-qq[qIndex]),3)
*Sq[n];
   }

    If (sumSq > .00001)
   {
     m1_x=m1_x/sumSq;
     m2_x=m2_x/sumSq;
     m3_x=m3_x/sumSq;
      }
```

Code Listing 3: Moments_x code

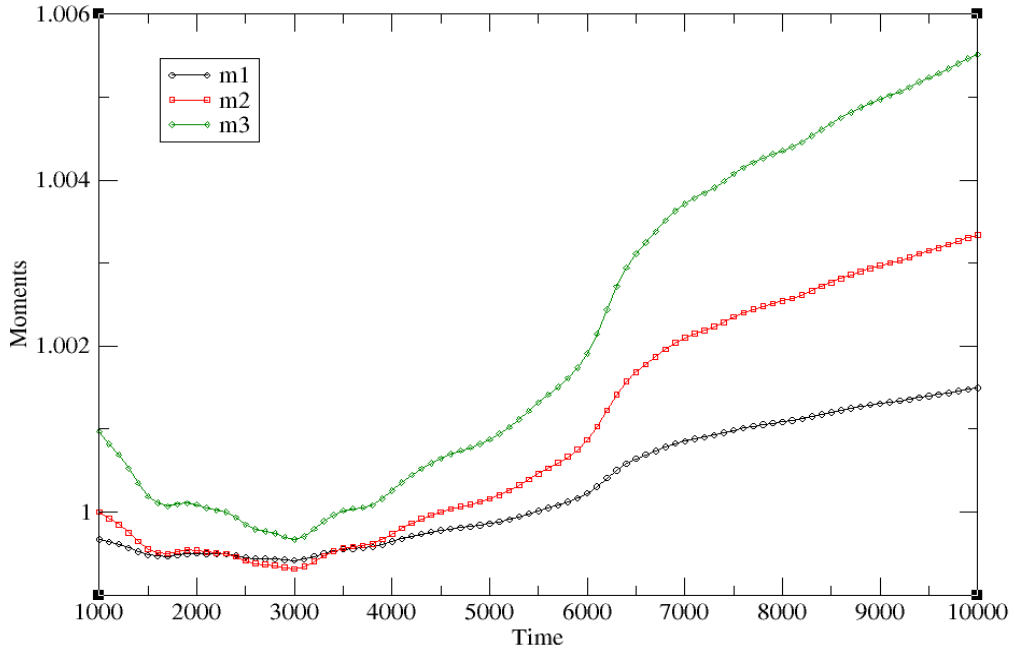


Figure 43: Moments vs. Time

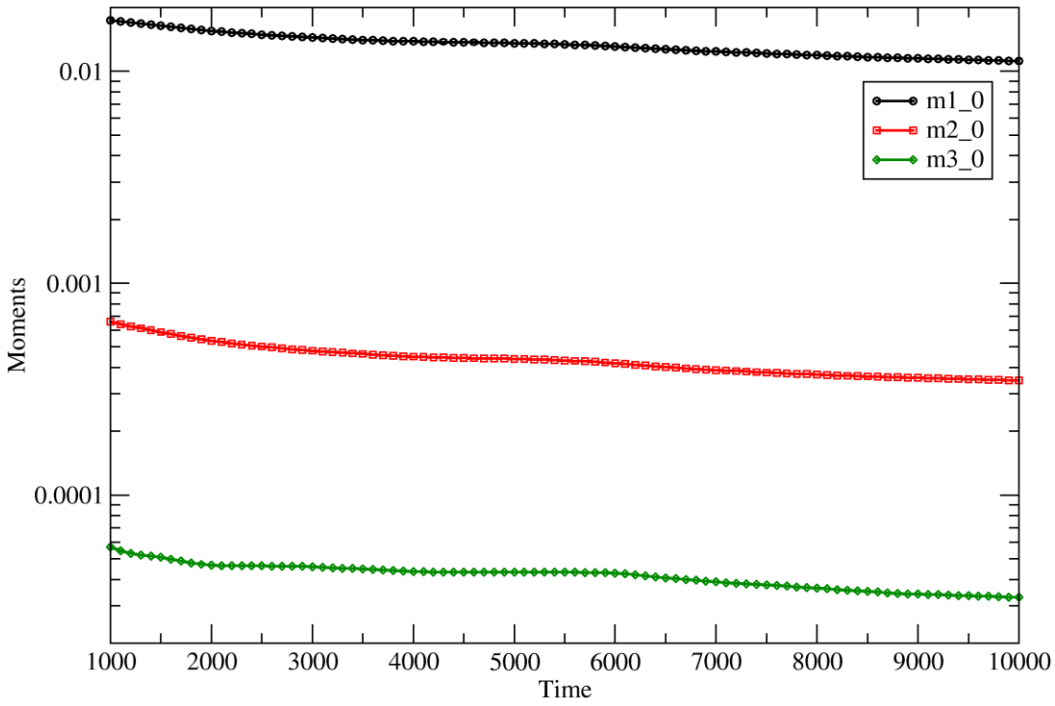


Figure 44: Log-scale plot of moments_0 vs. Time

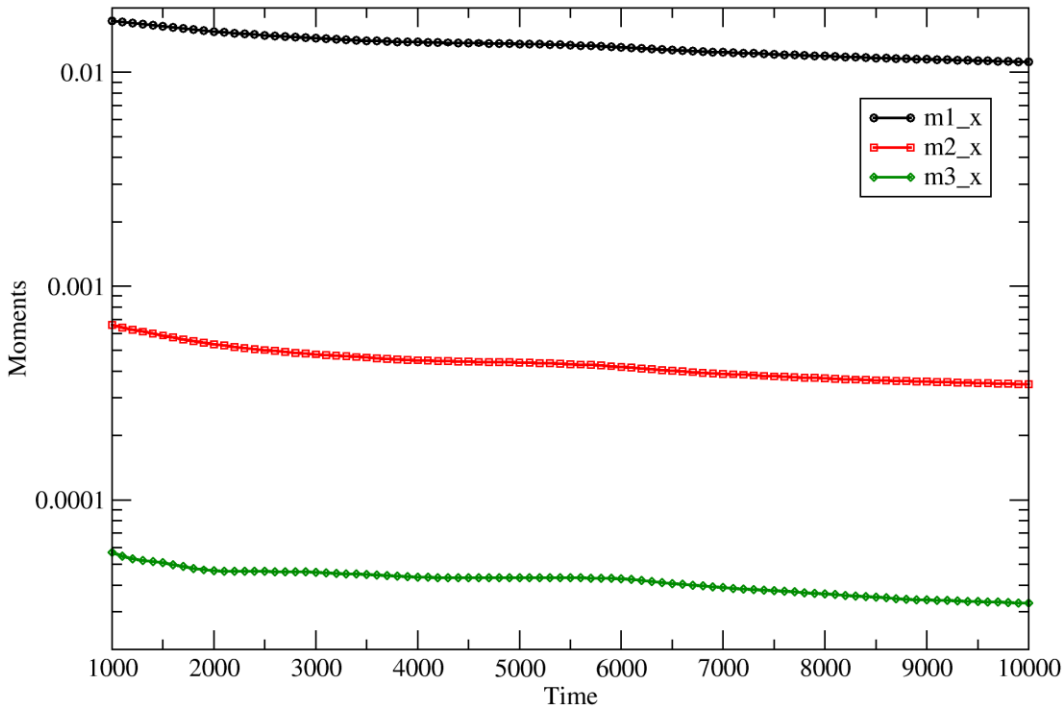


Figure 45: Log-scale plot of moments_x vs. Time

4.2.4 Grain Boundary Detection

Grain boundary detection is a complicated procedure because misorientations can occur within the grain itself, which increases the difficulty of detecting the separation lines between grains. In this work, grain boundaries are detected by using a method that combines different strategies used in a number of grain detection methods [16, 68]. Using more information on misorientations can give a more accurate estimation of grain boundaries. To get an accurate grain detection of grains at the boundaries of the system, the system is extended, and then the detection process is done on the extended area. After the detection process is done, only the grains within the original system boundaries are considered. This process is illustrated in Figures 46 and 47.

By using the same first two steps from the g_6 calculations, three types of information can be calculated:

1. The location of dislocations.
2. The form of each hexagon.

3. The orientation of atoms.

As defined, grain boundaries separate grains that have different orientations, however; it is very difficult to rely on this alone as the orientation of each atom within the same grain can also vary. Figure 48 shows the orientations within the grains for a sample system. To reduce the variations of orientation within a grain, the local mean is calculated for orientation angles for each atom and then use this mean for the orientation angle.

The grain detection method works as follows:

1. To identify atoms that may lie on a grain boundary, all disclinations are marked as a potential grain boundary atom. After that, atoms that have neighbors with an orientation difference of more than 10 degrees are marked.
2. After the initial boundary points are marked, we mark atoms that have one or more boundary marked neighboring atoms as potential boundary atoms. By doing this, a buffer of potential boundary points will surround potential grains. Next, other atoms in the system will be considered as grain atoms, and atoms within an enclosed boundary buffer point strip will be considered as one grain. Figure 49 illustrates this step.
3. After identifying grains, the grains are expanded one layer at a time until we have no more boundary points. Here, each atom of the boundary points will be joined with the grain that most of its neighbors are marked with. Figure 50b shows an illustration of this step for the system shown in Figure 50a.
4. Finally, for each grain, the grain boundaries will be identified by selecting atoms that have more than one neighbor of a different grain. In this step, triple junction points can also be identified, in which each one has neighbors belonging to three different grains. Figure 53 shows the number of triple junctions over time for a system of 512^2. Figure 50c shows the final grains and their boundaries. Figures 51 and 52 give examples of grain boundary detection at different time steps.

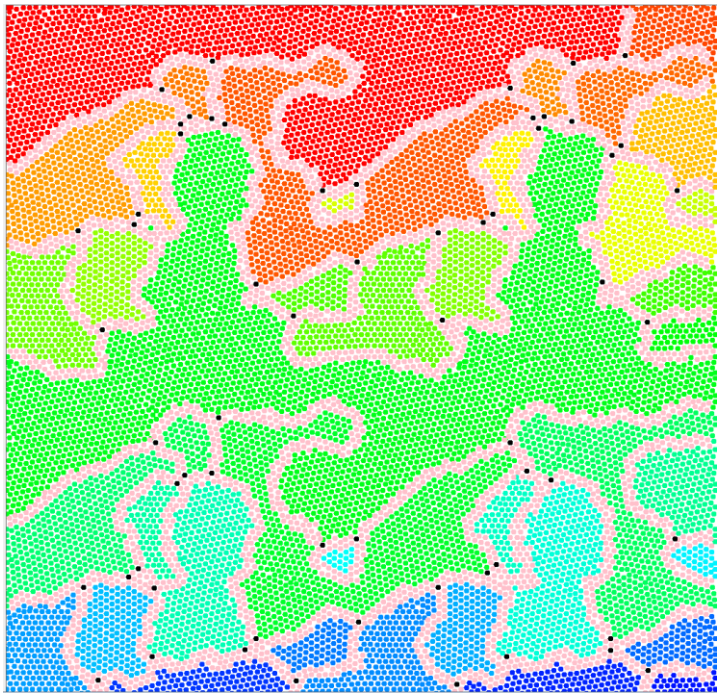


Figure 46: Extended area of the grains

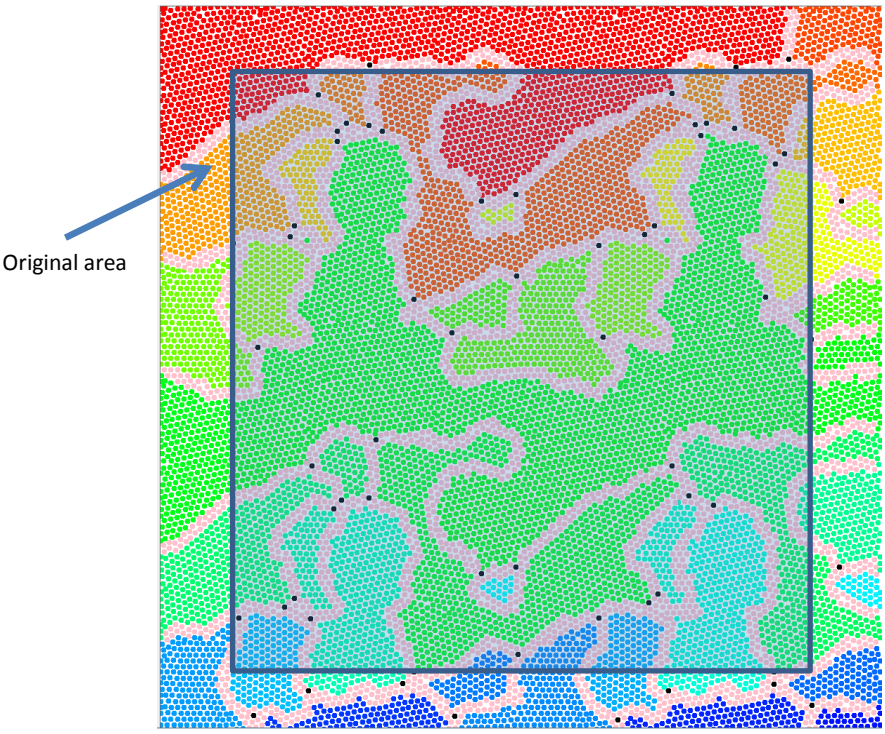


Figure 47: Original area of the grains

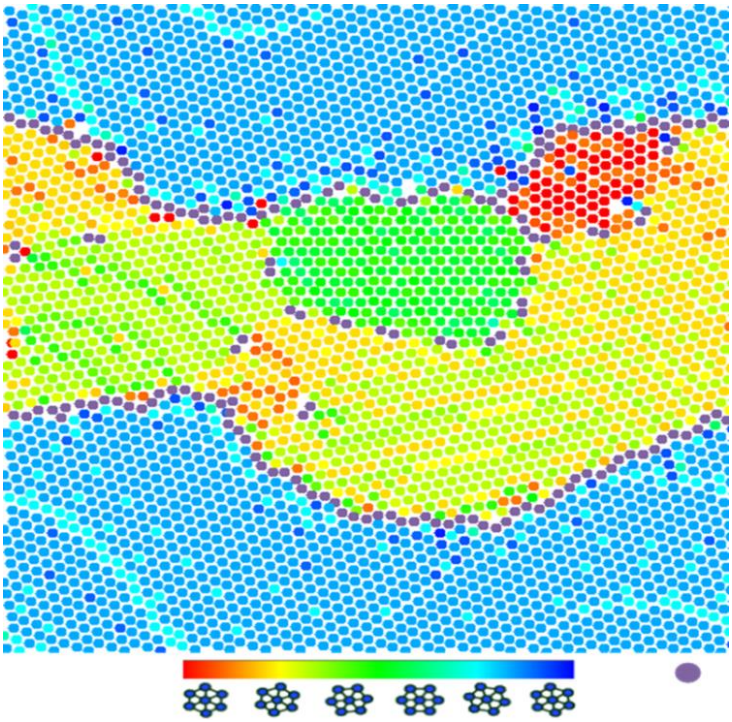


Figure 48: Atom orientation

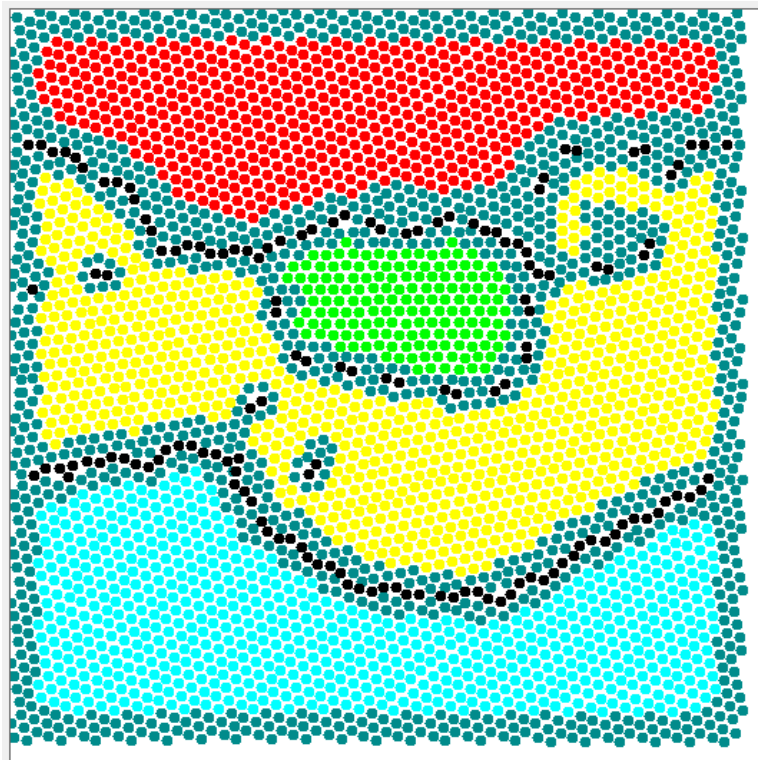
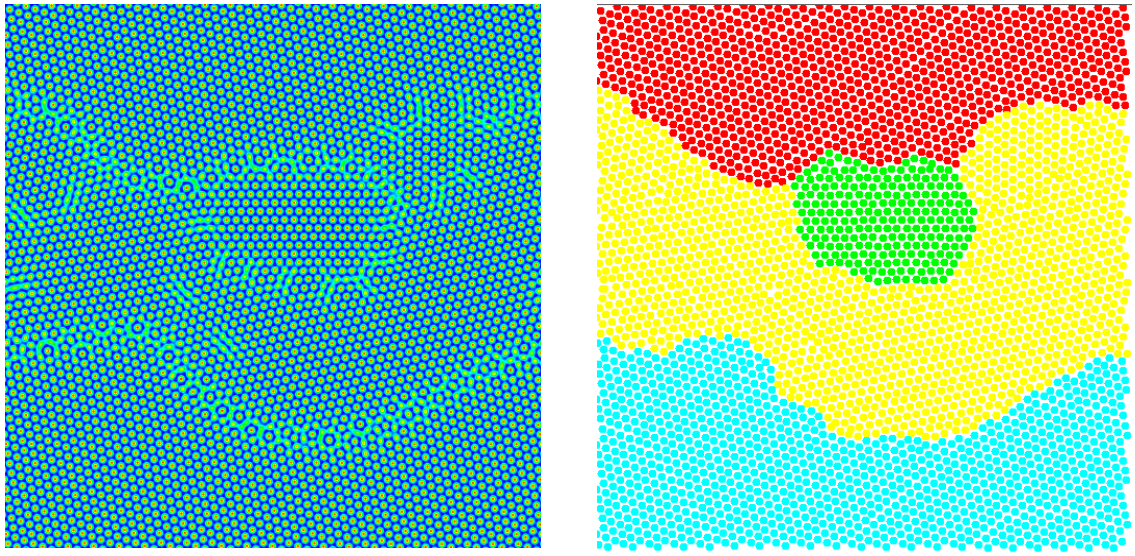
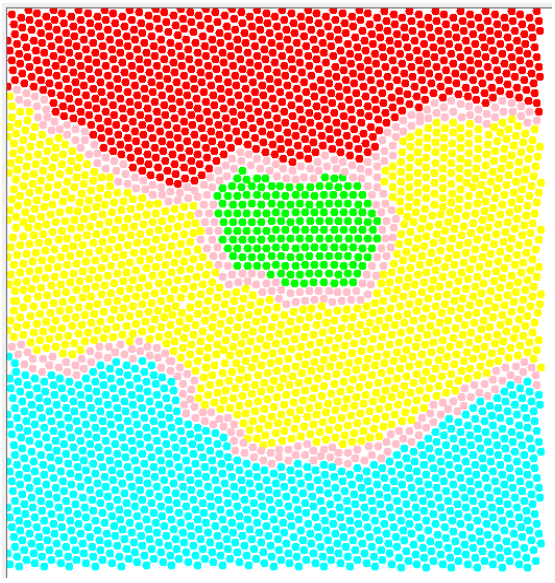


Figure 49: Grain identification and region buffers

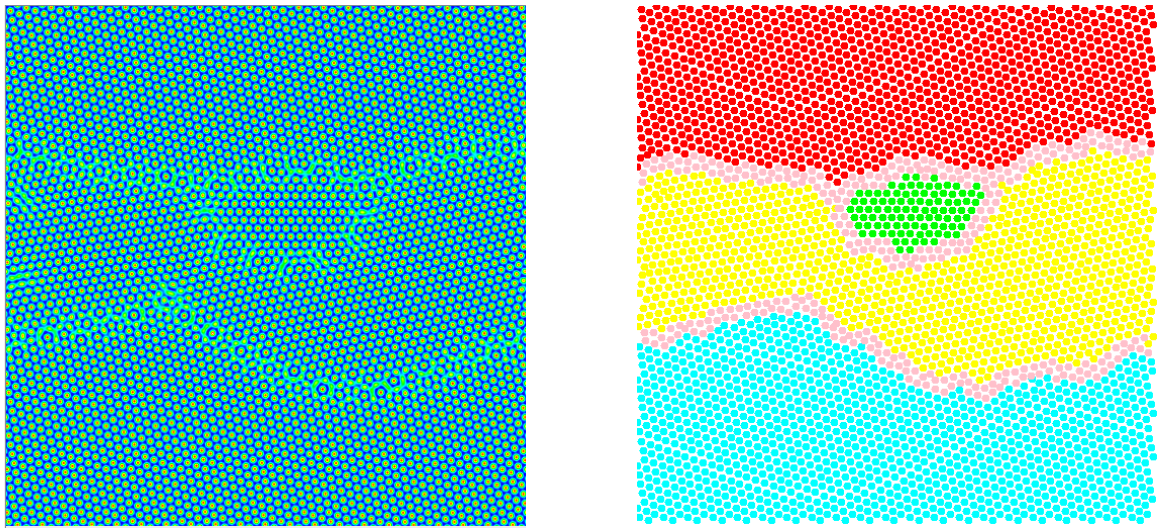


(a) (b)



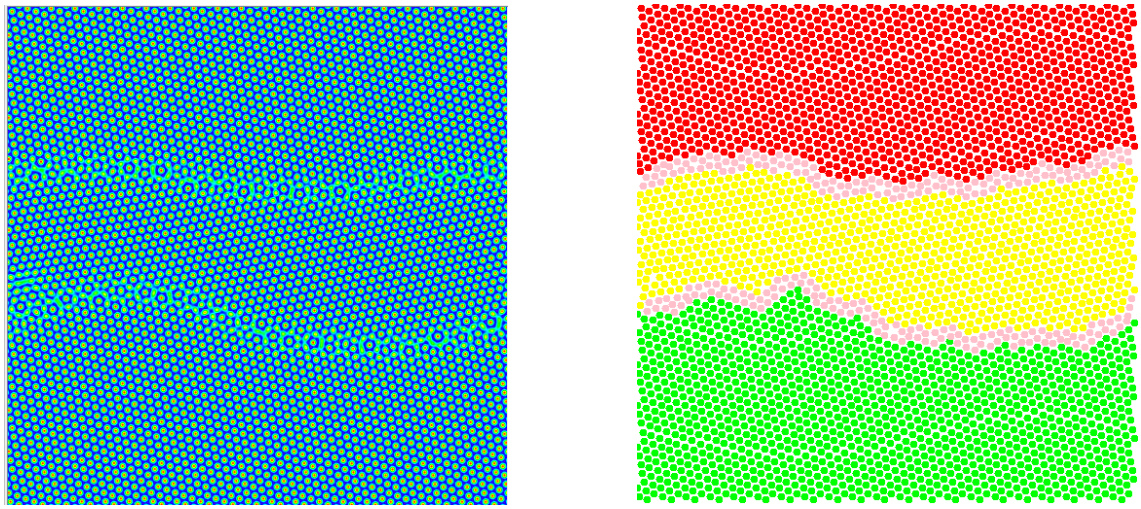
(c)

Figure 50: Example on grain detection for a system of size 512^2 at time 10000 (a) ψ plot (b) detected grains (c) grain boundaries detection



(a) (b)

Figure 51: Grain identification for a system size of 512^2 at time 15000



(a) (b)

Figure 52: Grain identification for a system size of 512^2 at time 20000

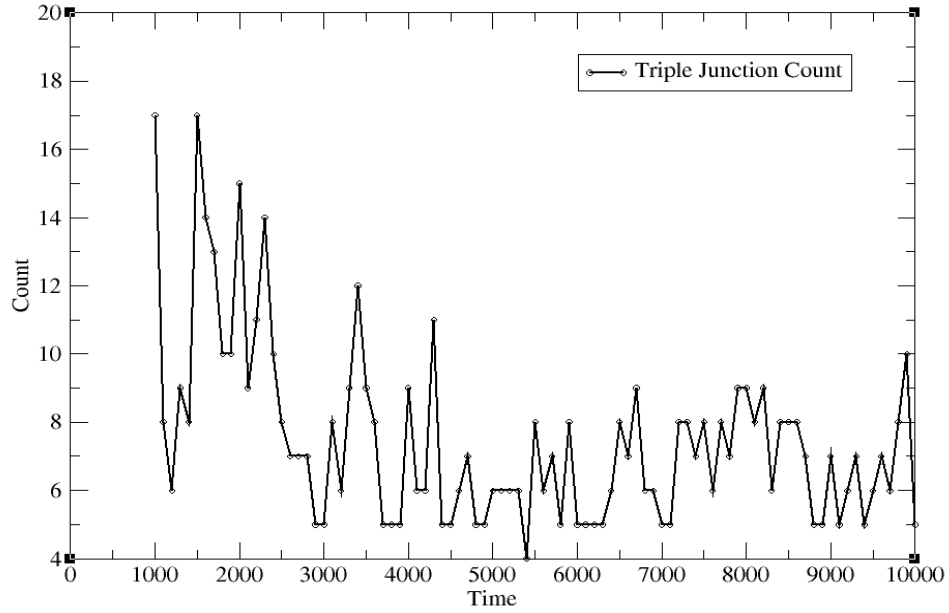


Figure 53 : Triple junction count

4.2.5 Average Curvature and Maximum Curvature of Grain Boundaries

Curvature is used to study the migration of grain boundaries and grain growth, as they migrate toward the center of that curvature [110, 111]. After determining the grain boundaries, we can use equation 4.2 to calculate the curvature of the grain boundaries:

$$\kappa(x,t) = -\frac{\partial^2 h/\partial x^2}{[1+(\partial h/\partial x)^2]^{3/2}} \tag{4.2}$$

where the (*x*,*y*) coordinate at a time *t* can be rewritten as $y = \kappa(x,t)$

The derivatives can be calculated using the following equations:

$$\partial h/\partial x\,|_i = \frac{h(i+1)-h(i-1)}{2\Delta x}\ for\ 2 \le i \le N-1 \tag{4.3}$$

$$\partial^2 h/\partial x^2\,|_i = \frac{h(i+1)-2h(i)+h(i-1)}{(\Delta x)^2} for\ 2 \le i \le N-1 \tag{4.4}$$

where *N* is the number of boundary atoms, and *h* is the grain boundary interface height. Figure 54 shows the maximum and average curvature of grain boundary atoms for a system of size 512^2.

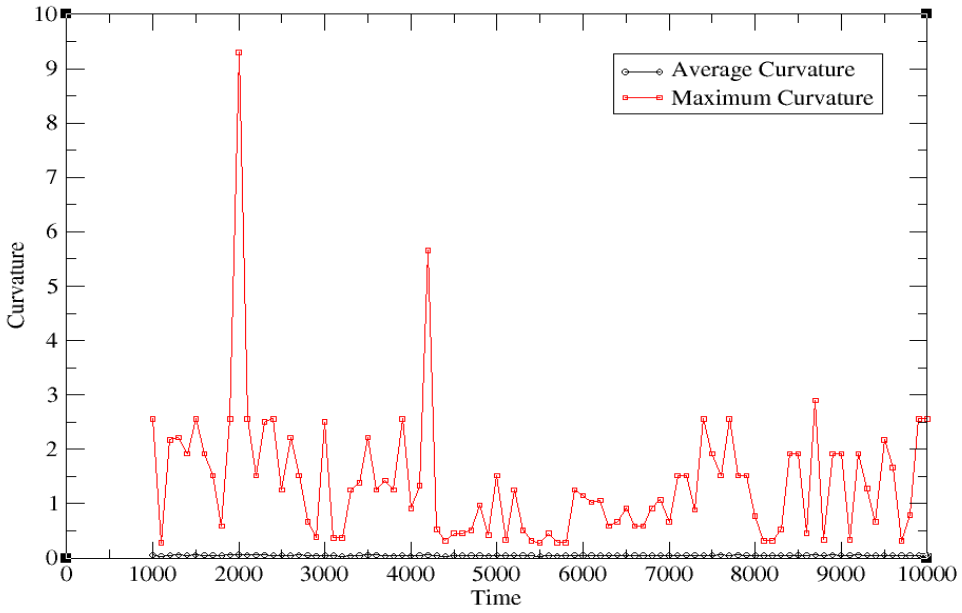


Figure 54: Average and maximum curvature of grain boundaries

4.2.6 Average and Maximum Velocity of Grain Boundary

To calculate the velocity of grain boundaries and triple junctions of grain boundaries, we need to know the current and the previous coordinates of atoms. This is very challenging as there is no way to track specific atoms between time steps. In addition, only the approximate location of each atom is known. To do this, we try to match the atoms in intervals of 10 time steps to increase the accuracy. After determining the pairs of atoms (x_1,y_1), (x_2,y_2), we calculate the velocity for each pair using the following calculations:

$$v_x = (x_2-x_1)/dt \tag{4.5}$$

$$v_y = (y_2-y_1)/dt \tag{4.6}$$

$$v = \sqrt{(v_x^2 + v_y^2)} \tag{4.7}$$

Figures 55 and 56 show the average and maximum velocities of grain boundary atoms and triple junction atoms, respectively.

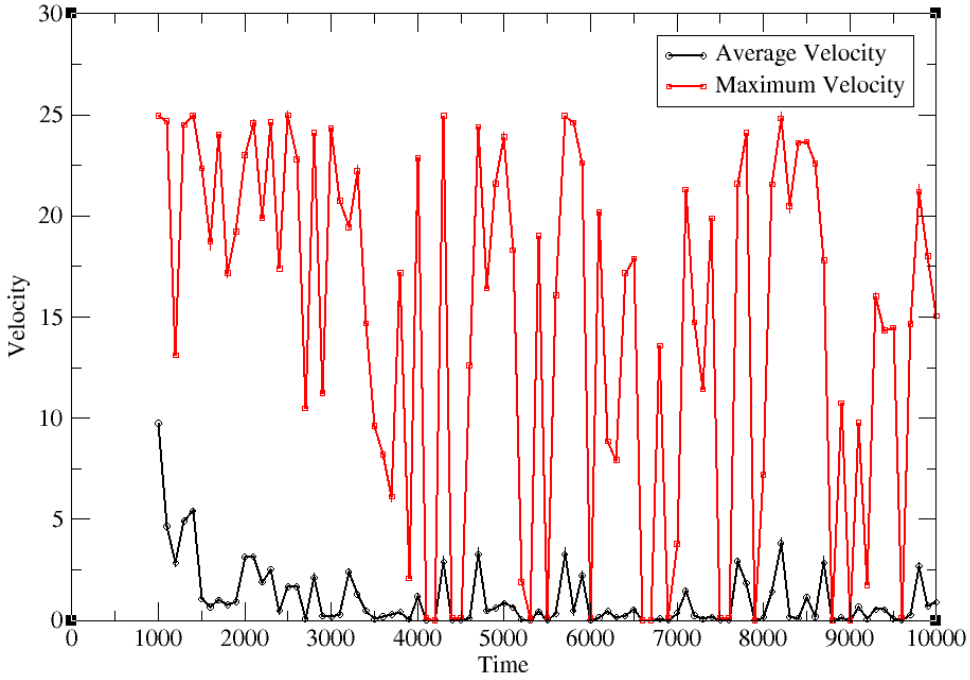


Figure 55: Average and maximum velocity for grain boundaries

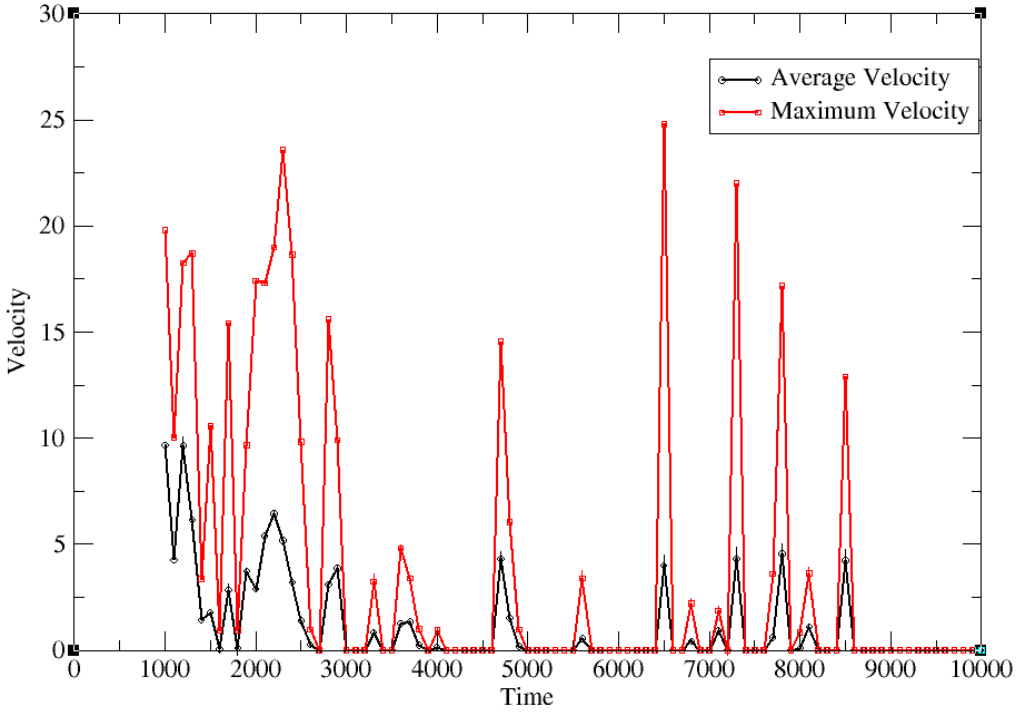


Figure 56: Average and maximum velocity for triple junctions

4.2.7 Grain Angle Misorientation

Here, we calculate the absolute difference of the grain angles between neighboring grains. The angle for each grain is the average of the theta angle for all of its atoms. Figure 57 shows the angle misorientation for a system of size 512^2.

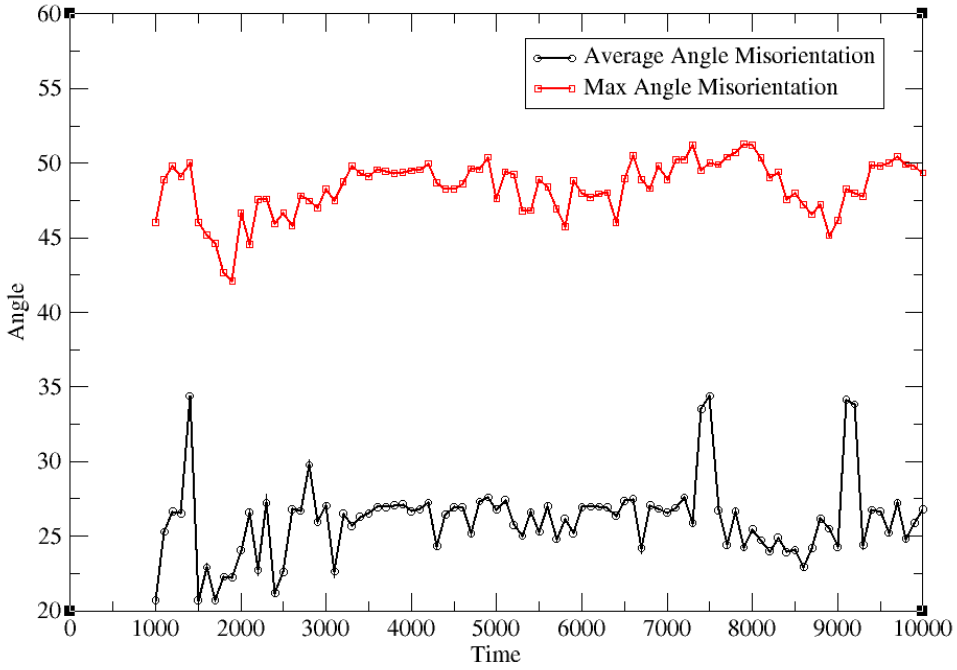


Figure 57: Average and maximum angle misorientation of neighboring grain angles

4.3 Experiments and Discussion of Results

As in section 3.6, I used the standard speedup metric for performance comparisons of the GPU codes and the serial codes. The total elapsed execution time measurement starts with the input of the parameters and ends with the final output.

4.3.1 Software and Hardware Setup

The PFC serial simulation is written in C++. The GPU version is written using CUDA 7.0. For the output, I used the HDF [73] software to generate the system's state figures as in Figure 50a. Results have been generated using the NVIDIA K40c. Table 13 gives the hardware specifications for the K40c GPU used in the preliminary experiments, and the CPU used to run the serial code.

Table 13: List of major specifications of the parallel processors for the experiments

	CPU	GPU
Model	Intel® Xeon ® E5-2680 v2	Nvidia Tesla K40
Micro Architecture	Ivy Bridge (EP)	Kepler
Number of cores	10 (**Hyper threaded**)	2880 (15 ×192)
Clock Frequency	2.85 GHz	0.745 GHz

4.3.2 Performance Analysis

To analyze the performance of the GPU and the serial codes, I conducted three experiments; one for the g_6 function, the second experiment is for the PFC grain growth simulation, and the third for the correlation length function. The three experiments ran for 10000 time steps, and used 128 threads per block. To evaluate the systems' performance, I ran the codes according to the specified parameters, and then generated the speedup plots shown in Figures 58, 59 and 60, and Tables 14, 15 and 16.

The first experiment is used to compute the execution time of the g_6 function. As can be observed from Figure 58 and Table 14, for a system size of 8192^2, the speedup is almost 21 times. The main bottleneck is the circular average step, as the search is conducted by each and every atom in the system.

Table 14: g_6 runtime results (seconds), average of 20 runs

System size	Serial execution time	GPU execution time	Speedup
512^2	1.11	0.225	4.933
1024^2	5.3	0.43	12.326
2048^2	28.69	1.97	14.563
4096^2	173.7	8.325	20.865
8192^2	1174.04	56.001	20.965

Table 15: PFC simulation run time (seconds), total simulation time

System size	Serial execution time	GPU execution time	Speedup
512^2	760.03	47.1	16.137
1024^2	4185.6	150.3	27.848
2048^2	18101.56	540.62	33.483
4096^2	85900.1	2230.4	38.513
8192^2	395260.87	8536.604	46.302

Table 16: Correlation length function run time (seconds), average of 20 runs

System size	Serial execution time	GPU execution time	Speedup
512^2	0.77	0.225	3.42
1024^2	1.99	0.43	4.63
2048^2	11.5	1.97	5.84
4096^2	68.83	8.325	8.27
8192^2	540.3	56.001	9.65

For the PFC grain growth simulation experiment, Figure 59 and Table 15 show that the speedup is more than 46 times for a system size of 8192^2. This system size requires a huge memory size (more than 10 GB of memory) for the cuFFT plans, so I only conducted the results on the K40c, as it has 12 GB of RAM.

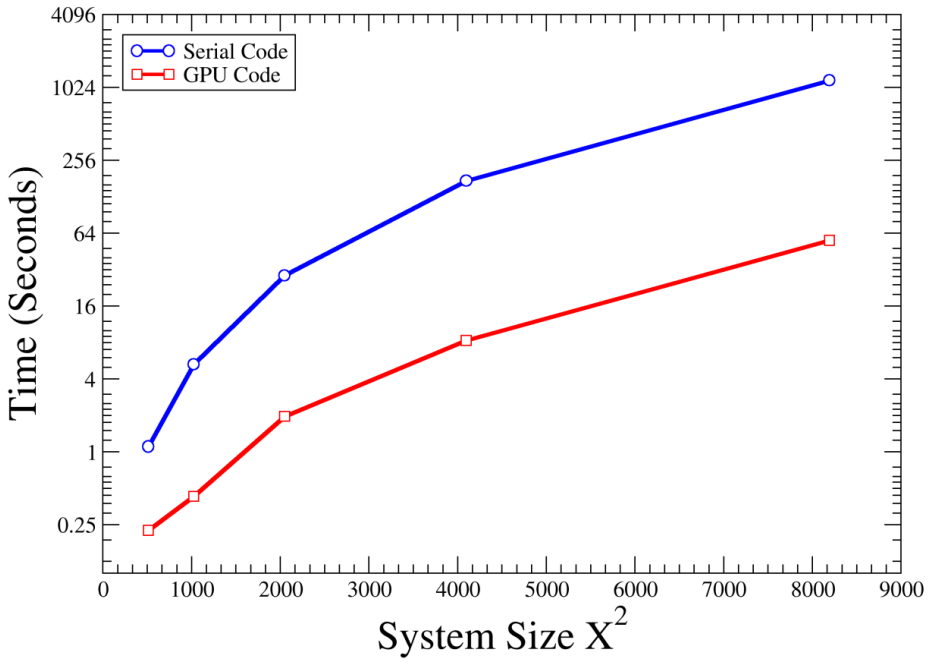


Figure 58: Log-scale run time comparison for the g_6 function

The correlation length function is implemented by using the g_6, but it does not need to calculate it for all *r* values for the serial code, but the GPU code calculates all the values of g_6 for all the r values, and then finds the correlation length. Table 16 and Figure 60 show the runtime and speedup for the correlation length function codes.

The ψ array and other auxiliary arrays are always on the GPU and they are only copied back to the CPU for output purposes. In this way, the overhead of memory communication between the CPU and the GPU is reduced significantly.

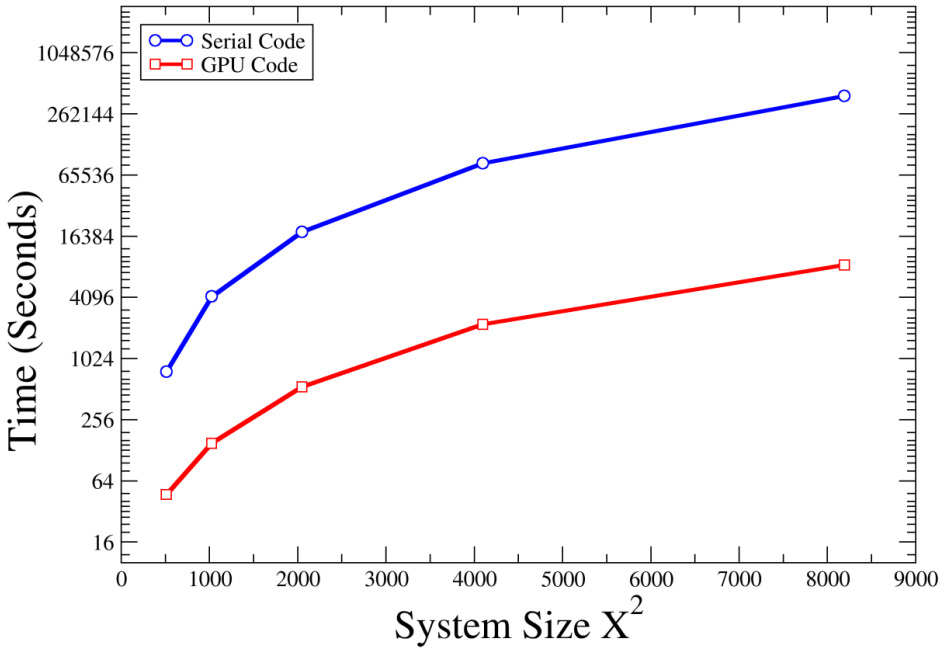


Figure 59: Log-scale runtime comparison for the PFC iterator simulation

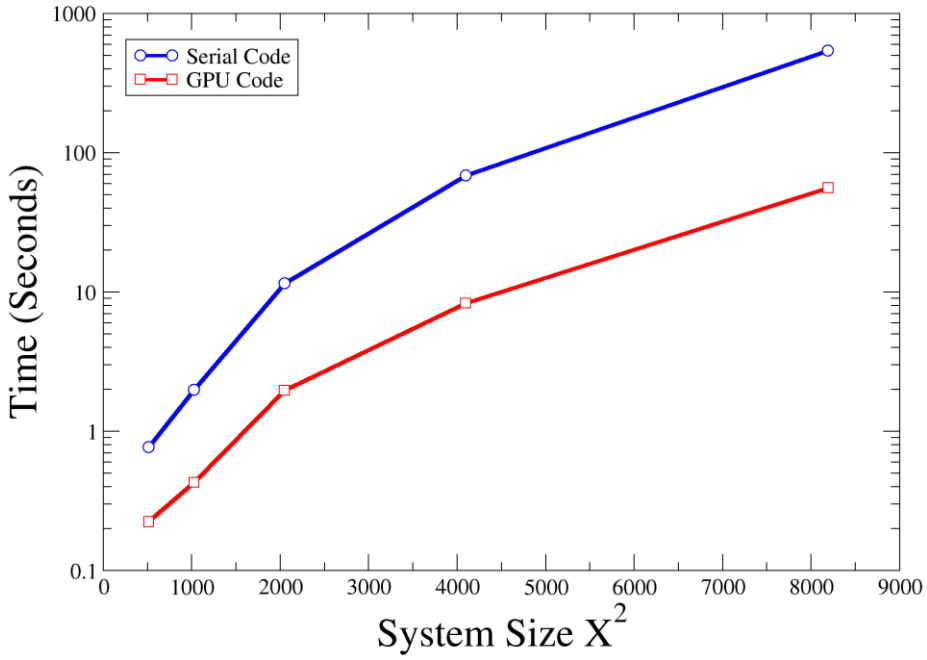


Figure 60: Log-scale runtime comparison for the correlation length function

4.4 Summary

In this chapter, I presented how I implemented and optimized the crystal growth simulation using the PFC model. One of the most challenging aspects of this simulation is that the atoms are represented by a periodic function, where local maxima represent atom centers. As the simulation runs, the atom centers change, and the code needs to detect them at any time we need to calculate any property related to atom positions. I faced many other challenges during this research, such as designing different GPU kernels to solve the differential equations, calculating the g_6 property, grain detection, dislocation and disclination detection, and running the system with large problem sizes. For the PFC simulation, most of the data structures and arrays were placed on the GPU to reduce the memory transfer overhead, and only copy them back to do I/O. Currently only three properties have been ported to the GPU, although I plan to do port more in the near future.

CHAPTER 5 CONCLUSION AND FUTURE WORK

Designing efficient algorithms for a parallel architecture can offer significant performance advantages over just porting an existing sequential algorithm. In addition, exploiting the design features of modern coprocessors can help in achieving even more speedups.

This dissertation has shown how the microcell list helped in achieving speedups over the conventional cell list for the MC CPU and Intel's Xeon Phi coprocessor for both the molecule and the system intermolecular interactions. For the GPU, the microcell list was better than the conventional cell list only for the molecule intermolecular energy interactions, and it did not show any speedup for calculating the system's intermolecular energy. The main reason is that many threads are created to calculate the interactions of this large number of small cells, which many of them can be empty in a sparse system. This issue was not visible in the OpenMP implementation because those empty cells were not processed, as a neighbor list was constructed from the nonempty cells. As for storage, the microcell list requires more space to store and maintain all of the small cells compared to the large cells of the conventional cell list.

This dissertation also introduced two scalable cell list implementations for calculating the system's intermolecular force interactions for the Gibbs ensemble using two hybrid parallel platforms, MPI+OpenMP and MPI+CUDA. Experiments were run on different parallel architectures, including multicore CPUs, Phi coprocessors, and GPUs. I also studied the effect of increasing the number of threads and devices for the MPI+OpenMP code, and the number of devices for the MPI+CUDA code. The presented results show that both algorithms scale well on all platforms. The GPU results were the best compared to the multicore CPU and Phi results. The microcell list achieves better results on small to medium sized systems, while the optimized conventional cell list performs better on large systems. The main reason for that is the overhead of maintaining and processing a large number of small cells. The microcell list achieves better results on small systems, while the optimized conventional cell list performs better on large systems.

For the grain growth simulation, the code has reached the stage where I implemented a serial and a GPU version that calculate many properties. This will enable us to produce results and analyze them to be ready for publication. The code now can run large simulations of 8192^2 that achieve a speedup of more than 46 times over the serial code for the PFC simulation.

At this stage, the two projects have reached a mature state. For the last 18 months, the GOMC group and I have released 9 beta releases GOMC code, and in October/ 2015 we plan to release version 1.0 of GOMC. The next stage is going to be the implementation of the *Ewald* code.

Future work will include further research into tuning the microcell list for the GPU implementation, in which the processing of empty cells needs to be reduced, and introduce better load balancing, and also tune it for different box densities. In addition, further work will be conducted on how to improve the microcell list and tune it for larger problem sizes. I will also study the MPI+OpenMP implementation on the Phi coprocessors to see if we can improve its performance. Furthermore, I will implement other MPI reduction methods to be used for larger clusters. Finally, I plan to evaluate other spatial indexing algorithms that may be able to further reduce the number of unnecessary pairwise energy calculations. As for the PFC work, I plan to port all the calculated properties to the GPU, and work on the 3D PFC simulation.

REFERENCES

[1] J. Nickolls and W. J. Dally, "The GPU Computing Era," *IEEE Micro*, vol. 30, no. 2, pp. 56-69, 2010.

[2] D. B. Kirk and W. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, 2nd ed., Elsevier Inc., 2012.

[3] "Oak Ridge Computing Facility," [Online]. Available: https://www.olcf.ornl.gov/titan/. [Accessed Jan 2015].

[4] "OpenCL," [Online]. Available: https://www.khronos.org/opencl/. [Accessed Jan 2015].

[5] S. Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs, Elsevier, 2013.

[6] NVIDIA, "CUDA C PROGRAMMING GUIDE v7.5," 2015.

[7] "NVIDIA GPU-ACCELERATED LIBRARIES," [Online]. Available: https://developer.nvidia.com/gpu-accelerated-libraries. [Accessed Feb 2015].

[8] J. Mick, E. Hailat, V. Russo, K. Rushaidat, L. Schwiebert and J. Potoff, "GPU-accelerated Gibbs ensemble Monte Carlo simulations of Lennard-Jonesium," *Computer Physics Communications*, vol. 184, no. 12, pp. 2662–2669, 2013.

[9] "LAMMPS," [Online]. Available: http://lammps.sandia.gov/. [Accessed Feb 2015].

[10] "NAMD," [Online]. Available: http://www.ks.uiuc.edu/Research/namd/. [Accessed Oct 2015].

[11] "Amber," [Online]. Available: http://ambermd.org/. [Accessed Feb 2015].

[12] "HOOMD-blue," [Online]. Available: http://codeblue.umich.edu/hoomd-blue/. [Accessed Jan 2015].

[13] M. Bjerre, J. M. Tarp, L. Angheluta and J. Mathiesen, "Rotation-induced grain growth and

stagnation in phase-field crystal models," *Physical Review E,* vol. 88, no. 2, pp. 020401-020404, 2013.

[14] S. Wang, "Molecular Dynamics Simulation Study of Grain Boundary Migration in Nanocrystalline Pd," B.S. Thesis, Department of Mechanical Engeering, University of Science and Technology, Beijing, China, 2006.

[15] R. Backofen, K. Barmak, K. Elder and A. Voigt, "Capturing the complex physics behind universal grain size distributions in thin metallic films," *Acta Materialia,* vol. 64, pp. 72-77, 2014.

[16] J. M. Castillo, J. Gross, H.-J. Wunderlich, C. Braun and S. Holst, "Acceleration of Monte-Carlo molecular simulations on hybrid computing architectures," in *Proceedings of the 2012 IEEE 30th International Conference on Computer Design (ICCD)*, Montreal, QC, Canada, 2012.

[17] I. Beichl and F. Sullivan, "The metropolis algorithm," *Computing in Science Engineering,* vol. 2, no. 1, pp. 65-69, 2000.

[18] D. Frenkel and B. Smit, Understanding Molecular Simulation: From Algorithms to Applications, 2nd ed., Academic Press, 2001.

[19] J. I. Siepmann and D. Frenkel, "Configurational bias Monte Carlo: a new sampling scheme for flexible chains," *Molecular Physics: An International Journal at the Interface Between Chemistry and Physics,* vol. 75, no. 1, pp. 59-70, 1992.

[20] C.-T. Yang, C.-L. Huang and C.-F. Lin, "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters," *Computer Physics Communications,* vol. 182, no. 1, pp. 266-269, 2011.

[21] L. Wolf, "The GPU Revolution," *Chemical & Engineering News,* vol. 88, no. 44, pp. 27-29, 2010.

[22] C. M. Wittenbrink, E. Kilgariff and A. Prabhu, "Fermi Gf100 GPU Architecture," *IEEE Micro,* vol. 31, no. 2, pp. 50-59, 2011.

[23] N. Wilt, The CUDA Handbook: A Comprehensive Guide to GPU Programming, Addison-Wesley Professional, 2013.

[24] D. A. Vega, C. K. Harrison, D. E. Angelescu, M. L. Trawick, D. A. Huse, P. M. Chaikin and R. A. Register, "Ordering mechanisms in two-dimensional sphere-forming block copolymers," *Physical Review E,* vol. 71, no. 6, pp. 061803-061815, 2005.

[25] O. Vaulina and E. Vasilieva, "Orientational order and formation of topological defects in two-dimensional systems," *Journal of Experimental and Theoretical Physics,* vol. 117, no. 1, pp. 169-176, 2013.

[26] P. J. Steinhardt, D. R. Nelson and M. Ronchetti, "Bond-orientational order in liquids and glasses," *Physical Review B,* vol. 28, no. 2, pp. 784-805, 1983.

[27] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.

[28] G. Rong, T.-S. Tan, T.-T. Cao and Stephanus, "Computing two-dimensional Delaunay triangulation using graphics hardware," in *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, Redwood City, CA, USA, 2008.

[29] M. Qi, T.-T. Cao and T.-S. Tan, "Computing 2D Constrained Delaunay Triangulation Using the GPU," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 5, pp. 736-748, 2013.

[30] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE,* vol. 96, no. 5, pp. 879-899, 2008.

[31] J. O'Rourke, Computational Geometry in C, Cambridge University Press, 1998.

[32] M. Matsumoto and T. Nishimur, "Mersenne twister: a 623-dimensionally equidistributed

uniform pseudo-random number generato," *ACM Transactions on Modeling and Computer Simulation (TOMACS),* vol. 8, no. 1, pp. 3-30, 1998.

[33] W. Liu, B. Schmidt, G. Voss and W. Müller-Wittig, "Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA," *Computer Physics Communications,* vol. 9, no. 1, pp. 634-641, 2008.

[34] C. Harrison, D. E. Angelescu, M. Trawick, Z. Cheng, D. A. Huse, P. M. Chaikin, D. A. Vega, J. M. Sebastian, R. A. Register and D. H. Adamson, "Pattern coarsening in a 2D hexagonal system," *Europhysics Letters,* vol. 67, no. 5, pp. 800-806, 2004.

[35] E. Hailat, V. Russo, K. Rushaidat, J. Mick, L. Schwiebert and J. Potoff, "Parallel Monte Carlo simulation in the canonical ensemble on the graphics processing unit," *International Journal of Parallel, Emergent and Distributed Systems,* vol. 29, no. 4, pp. 379-400, 2014.

[36] E. Hailat, K. Rushaidat, L. Schwiebert, J. R. Mick and J. J. Potoff, "GPU-based Monte Carlo simulation for the Gibbs ensemble," in *Proceedings of the High Performance Computing Symposium*, San Diego, CA, USA, 2013.

[37] J. Ghorpade, J. Parande, M. Kulkarni and A. Bawaskar, "GPGPU Processing in CUDA Architecture," *Advanced Computing,* vol. 3, no. 1, pp. 105-120, 2012.

[38] B. Garzon and M. E. Costas, "Shape of hexatic domains of a two-dimensional Lennard-Jones system," *The Journal of Physical Chemistry,* vol. 97, no. 51, pp. 13860–13863, 1993.

[39] A. R. Brodtkorb, T. R. Hagen and M. L. Sætra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," *Journal of Parallel and Distributed Computing,* vol. 73, no. 1, pp. 4-13, 2013.

[40] M. Arora, S. Nath, S. Mazumdar, S. B. Baden and D. M. Tullsen, "Redefining the Role of the CPU in the Era of CPU-GPU Integration," *IEEE Micro,* vol. 32, no. 6, pp. 4-16, 2012.

[41] J. A. Anderson, E. Jankowski, T. L. Grubb, M. Engel and S. C. Glotzer, "Massively parallel

Monte Carlo for many-particle simulations on GPUs," *Journal of Computational Physics,* vol. 254, no. 1, pp. 27-38, 2013.

[42] A. Adland, Y. Xu and A. Karma, "Unified Theoretical Framework for Polycrystalline Pattern Evolution," *Physical Review Letter,* vol. 110, no. 26, pp. 265504, 2013.

[43] K. Moreland and E. Angel, "The FFT on a GPU," in *Proceedings of SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003* , San Diego, CA, USA, 2003.

[44] G. J. Tucker and S. M. Foiles, "Molecular dynamics simulations of rate-dependent grain growth during the surface indentation of nanocrystalline nickel," *Materials Science and Engineering,* vol. 571, no. 1, pp. 207-214, 2013.

[45] NVIDIA, *NVIDIA's Next Generation CUDA Compute Architecture: Fermi,* 2010.

[46] NVIDIA, *NVIDIA Next Generation CUDA Compute Architecture: Kepler GK110,* 2012.

[47] K. B. Daly, J. B. Benziger, P. G. Debenedetti and A. Z. Panagiotopoulos, "Massively parallel chemical potential calculation on graphics processing units," *Computer Physics Communications,* vol. 183, no. 10, pp. 2054-2062, 2012.

[48] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L.Beberg, D. L. Ensign, C. M. Bruns and V. S. Pande, "Accelerating molecular dynamic simulation on graphics processing units," *Journal of Computational Chemistry,* vol. 30, no. 6, pp. 864-872, 2009.

[49] M. Garland, S. L. Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang and V. Volkov, "Parallel computing experiences with CUDA," *IEEE Micro,* vol. 28, no. 4, pp. 13-27, 2008.

[50] D. Geer, "Taking the Graphics Processor beyond Graphics," *Computer,* vol. 38, pp. 14-16, 2005.

[51] J. Kim, J. M. Rodgers, M. Athnes and B. Smit, "Molecular monte carlo simulations using graphics processing units: To waste recycle or not?," *Journal of Chemical Theory and Computation,* vol. 7, no. 10, pp. 3208-3222, 2011.

[52] R. Farber, CUDA Application Design and Development, Elsevier Inc., 2011.

[53] W.-M. W. Hwu, GPU Computing Gems: Emerald Edition, Elsevier Inc., 2011.

[54] F. P. Preparata and M. I. Shamos, Computational Geometry: An Introduction, Springer-Verlag, 1985.

[55] M. Cosnard and D. Trystram, Parallel Algorithms and Architectures, International Thomson Computer Press, 1995.

[56] E. Lindholm, J. Nickolls, S. Oberman and a. J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro,* vol. 28, no. 2, pp. 39-55, 2008.

[57] L. Loyens, B. Smit and K. Esselink, "Parallel gibbs-ensemble simulations," *Molecular Physics,* vol. 86, no. 2, pp. 171-183, 1995.

[58] D. Luebke and G. Humphreys, "How GPUs Work," *Computer,* vol. 40, no. 2, pp. 96-100, 2007.

[59] NVIDIA, "CUFFT LIBRARY USER'S GUIDE v5.5," 2013.

[60] J. Fang, A. Varbanescu and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *Proceedings of 2011 International Conference on Parallel Processing (ICPP)*, Taipei City, Taiwan, 2011.

[61] C. J. O'Keeffe and G. Orkoulas, "Parallel canonical monte carlo simulations through sequential updating of particles," *The Journal of Chemical Physics,* vol. 130, no. 13, pp. 134109, 2009.

[62] A. Z. Panagiotopoulos, "Direct determination of phase coexistence properties of fluids by monte carlo simulation in a new ensemble," *Molecular Physics,* vol. 61, pp. 813-826, 1987.

[63] J. J. Potoff and D. A. Bernard-Brunel, "Jeffrey J. Potoff and Damien A. Bernard-Brunel. Mie potentials for phase equilibria calculations:Application to alkanes and perfluoroalkanes," *The Journal of Physical Chemistry B,* vol. 113, no. 44, pp. 14725-14731, 2009.

[64] T. Preis, P. Virnau, W. Paul and J. Schneider, "Gpu accelerated monte carlo simulation of the 2d and 3d ising model," *Journal of Computational Physics,* vol. 228, no. 12, pp. 4468-4477, 2009.

[65] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of Computational Chemistry,* vol. 28, no. 16, pp. 2618-2640, 2007.

[66] S. J. Zara and D. Nicholson, "Grand canonical ensemble monte carlo simulation on a transputer array," *Molecular Simulation,* vol. 5, no. 3, pp. 245-261, 1990.

[67] H. Kelly, "Parallel Monte Carlo Simulations of Vapour-Liquid Systems," M.S. Thesis, Department of Computer Science, The University of Edinburgh, Edinburgh, UK, 2006.

[68] E. Lazar, "Molecular Dynamic Studies in the Fracturing of Metals," B.S. Thesis, Department of Computer Science, Yeshiva University, New York, NY, USA, 2005.

[69] D. D. Donno, A. Esposito, L. Tarricone and L. Catarinucci, "Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD," *IEEE Antennas and Propagation Magazine,* vol. 52, no. 3, pp. 116-122, 2010.

[70] "GOMC," [Online]. Available: http://gomc.eng.wayne.edu. [Accessed May 2014].

[71] "Charmm," [Online]. Available: http://www.charmm.org/. [Accessed Nov 2013].

[72] "FFTW," [Online]. Available: http://www.fftw.org/. [Accessed Nov 2013].

[73] "HDF," [Online]. Available: http://www.hdfgroup.org/. [Accessed Dec 2013].

[74] "VMD," [Online]. Available: http://www.ks.uiuc.edu/Research/vmd/. [Accessed Jan 2014].

[75] "PSF file format," [Online]. Available: http://www.ks.uiuc.edu/Training/Tutorials/namd/namd-tutorial-win-html/node24.html. [Accessed Nov 2013].

[76] "PDB file format," [Online]. Available: http://www.rcsb.org/pdb/home/home.do. [Accessed Nov 2013].

[77] "Tesla vs. AMD" [Online]. Available: http://www.eweek.com/. [Accessed Sep 2015]

[78] "CUDA in action," [Online]. Available: http://www.nvidia.com/object/cuda-in-action.html. [Accessed Mar 2014].

[79] NVIDIA, "NVIDIA GeForce GTX 750 Ti," 2014.

[80] "Cmake," [Online]. Available: http://www.cmake.org/. [Accessed Feb 2014].

[81] L. Di Stefano and A. Bulgarelli, "A simple and efficient connected components labeling algorithm," in *Proceedings of International Conference on Image Analysis and Processing*, Naples, Italy, 1999.

[82] "Triangle," [Online]. Available: http://www.cs.cmu.edu/~quake/triangle.html. [Accessed Jan 2012].

[83] J. Jeffers and J. Reinders, Intels Xeon Phi Coprocessor High-Performance Programming, Morgan Kaufmann, 2013.

[84] "Tianhe-2" [Online]. Available: http://www.top500.org/featured/systems/tianhe-2/. [Accessed Oct 2015].

[85] "OpenMP" [Online]. Available: http://openmp.org/wp/. [Accessed Mar 2015]

[86] "OpenACC" [Online]. Available: http://www.openacc.org/. [Accessed Oct 2015]

[87] "Maxwell Architecture" [Online]. Available: http://devblogs.nvidia.com/parallelforall/5-things-you-should-know-about-new-maxwell-gpu-architecture/. [Accessed Sep 2015]

[88] L. Schwiebert, E. Hailat, K. Rushaidat, J. Mick, and J. Potoff. "An Efficient Cell List

Implementation for Monte Carlo Simulation on GPUs, " *CoRR* abs/1408.3764. 2014.

[89] L. Verlet, Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*s, vol. 159, no. 1, pp. 98-103, 1967.

[90] A.J. Proctor, C.A. Stevens, and S.S. Cho, "GPU-Optimized Hybrid Neighbor/Cell List Algorithm for Coarse-Grained MD Simulations of Protein and RNA Folding and Assembly," *in Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics (BCB 13)*, Washington DC, 2013.

[91] M.P. Allen and D. J.Tildesley, Computer Simulation of Liquids. Oxford University Press, New York, 1987.

[92] W. Mattson, W. and B. M. Rice. "Near-neighbor calculations using a modified cell-linked list method," *Computer Physics Communications*, vol. 119, no. 2-3, pp. 135-148, 1998.

[93] J. A. Anderson, C. D. Lorenz, and A. Travesset. "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342-5359, 2008.

[94] J. van Meel, A. Arnold, D. Frenkel, S. Portegies Zwart, and R. Belleman. "Harvesting graphics power for MD simulations," *Molecular Simulation*, vol. 34, no. 3, pp. 259-266, 2008.

[95] G. S. Grest, B. Dünweg, and K. Kremer. "Vectorized link cell Fortran code for molecular dynamics simulations for a large number of particles," *Computer Physics Communications*, vol. 55, no. 3, pp.269-285, 1989.

[96] K. Rushaidat, L. Schwiebert, B. Jackman, J. Mick, and J. Potoff, "Efficient Parallel Cell List Algorithms for Monte Carlo Simulations," *in Proceedings of The 2015 Summer Simulation Multi-Conference (SummerSim'15)*, Chicago, IL, July 2015.

[97] S. C. Mcgrother and K. E. Gubbins, "Constant pressure gibbs ensemble monte carlo simulations of adsorption into narrow pores," *Molecular Physics*, vol. 97, no. 8, pp. 955–965, 1999.

[98] T. Cagin and B. M. Pettitt, "Grand molecular dynamics: A method for open systems," *Molecular Simulation*, vol. 6, no. 1-3, pp. 5-26, 1991.

[99] A. Papadopoulou, E. D. Becker, M. Lupkowski, and F. van Swol, "Molecular dynamics and monte carlo simulations in the grand canonical ensemble: Local versus global control," *The Journal of Chemical Physics*, vol. 98, no. 6, pp. 4897-4908,1993.

[100] "Simpatico" [Online]. Available: ttp://research.cems.umn.edu/morse/code/simpatico/html/. [Accessed Mar 2015]

[101] K. Rushaidat, L. Schwiebert, B. Jackman, J. Mick, and J. Potoff, "Evaluation of Hybrid Parallel Cell List Algorithms For Monte Carlo Simulation," *in Proceedings of The 7th IEEE International Workshop on Multicore and Multithreaded Architectures and Algorithms (M2A2 2015)*, New York, NY, Aug. 2015.

[102] "MCCCS Towhee" [Online]. Available: http://towhee.sourceforge.net/. [Accessed Mar 2015]

[103] K. Rushaidat, Z. Huang, and Loren Schwiebert, "PFC property calculation report", Internal report, 2014.

[104] "CUDA Toolkit" [Online]. Available: https://developer.nvidia.com/cuda-toolkit. [Accessed Sep 2015]

[105] "mvapich2" [Online]. Available: http://mvapich.cse.ohio-state.edu/. [Accessed Sep 2015]

[106] J. Mick, M. Barhaghi, B. Jackman, K. Rushaidat, L. Schwiebert and J. Potoff, "Optimized Mie potentials for phase equilibria: Application to noble gases and their mixtures with n-alkanes," *The Journal of Chemical Physics,* vol. 143, pp. 114504, 2015.

[107] "Nvidia’s roadmap" [Online]. Available: http://www.extremetech.com/gaming/201417-nvidias-2016-roadmap-shows-huge-performance-gains-from-upcoming-pascal-architecture. [Accessed Oct 2015]

[108] "Structure factor" [Online]. Available: http://reference.iucr.org/dictionary/Structure_factor. [Accessed Oct 2015]

[109] K. R. Edler, J. Viñals, and M. Grant, "Dynamic scaling and quasiordered states in the two-dimensional Swift-Hohenberg equation," *Physical Review A*, vol. 46, no. 12, pp. 7618-7629, 1992.

[110] K. R. Edler, J. Viñals, and M. Grant, "Ordering Dynamics in the Two-Dimensional Stochastic Swift-Hohenberg Equation," *Physical Review Letters*, vol. 68, no. 20, pp. 3024-3027, 1992.

[111] S. Shahandeh and M. Militzer, "Grain boundary curvature and grain growth kinetics with particle pinning," *Philosophical Magazine*, vol. 93, no. 24, pp. 3231-3247, 2013.

[112] K. R. Edler, M. Katakowski, M. Hahtaja, and M. Grant, "Modeling Elasticity in Crystal Growth," *Physical Review Letters*, vol. 88, no. 24, 2002.

[113] K. R. Edler, N. Provatas, J. Berry, P. Stefanovic, and M. Grant, "Phase-field crystal modeling and classical density functional theory of freezing," *Physical Review B*, vol. 75, no. 6, 2007.

[114] Z.-F. Huang and K. R. Edler, "Mesoscopic and Microscopic Modeling of Island Formation in Strained Film Epitaxy," *Physical Review Letters*, vol. 101, no. 15, 2008.

[115] "Scheduling Loop Iterations in OpenMP" [Online]. Available: https://people.sc.fsu.edu/~jburkardt/c_src/schedule_openmp/schedule_openmp.html. [Accessed Oct 2015]

[116] J. K. Shah and E. J. Maginn, "A general and efficient Monte Carlo method for sampling

intramolecular degrees of freedom of branched and cyclic molecules," The Journal of Chemical Physics, vol. 135, 2013.

ABSTRACT

EFFICIENT ALGORITHMS AND OPTIMIZATIONS FOR SCIENTIFIC COMPUTING ON MANY-CORE PROCESSORS

by

KAMEL RUSHAIDAT

December 2015

Advisor: Dr. Loren Schwiebert

Major: Computer Science

Degree: Doctor of Philosophy

Designing efficient algorithms for many-core and multicore architectures requires using different strategies to allow for the best exploitation of the hardware resources on those architectures. Researchers have ported many scientific applications to modern many-core and multicore parallel architectures, and by doing so they have achieved significant speedups over running on single CPU cores. While many applications have achieved significant speedups, some applications still require more effort to accelerate due to their inherently serial behavior.

One class of applications that has this serial behavior is the Monte Carlo simulations. Monte Carlo simulations have been used to simulate many problems in statistical physics and statistical mechanics that were not possible to simulate using Molecular Dynamics. While there are a fair number of well-known and recognized GPU Molecular Dynamics codes, the existing Monte Carlo ensemble simulations have not been ported to the GPU, so they are relatively slow and could not run large systems in a reasonable amount of time. Due to the previously mentioned shortcomings of existing Monte Carlo ensemble codes and due to the interest of researchers to have a fast Monte Carlo simulation framework that can simulate large systems, a new GPU framework called GOMC is implemented to simulate different particle and molecular-based force fields and ensembles. GOMC simulates different Monte Carlo ensembles such as the canonical, grand canoni-

cal, and Gibbs ensembles. This work describes many challenges in optimizing the GPU Monte Carlo code for such ensembles and how I addressed these challenges. Such challenges include the optimization of the energy calculations for computationally intensive moves, balancing work among threads, and utilizing the hardware of the GPU.

This work also describes efficient many-core and multicore large-scale energy calculations for Monte Carlo Gibbs ensemble using cell lists. Designing Monte Carlo molecular simulations is challenging as they have less computation and parallelism when compared to similar molecular dynamics applications. The modified cell list allows for more speedup gains for energy calculations on both many-core and multicore architectures when compared to other implementations without using the conventional cell lists. The work presents results and analysis of the cell list algorithms for each one of the parallel architectures using top of the line GPUs, CPUs, and Intel's Phi coprocessors. In addition, the work evaluates the performance of the cell list algorithms for different problem sizes and different radial cutoffs.

In addition, this work evaluates two cell list approaches, a hybrid MPI+OpenMP approach and a hybrid MPI+CUDA approach to test for scalability. The cell list methods are evaluated on a small cluster of multicore CPUs, Intel Phi coprocessors, and GPUs. The performance results are evaluated using different combinations of MPI processes, threads, and problem sizes.

Another application presented in this dissertation involves the understanding of the properties of crystalline materials, and their design and control. Recent developments include the introduction of new models to simulate system behavior and properties that are of large experimental and theoretical interest. One of those models is the Phase-Field Crystal (PFC) model. The PFC model has enabled researchers to simulate 2D and 3D crystal structures and study defects such as dislocations and grain boundaries. In this work, I used GPUs to accelerate and optimize the calculation of various dynamic properties of polycrystals in the 2D PFC model. Some properties require very extensive computation that may involve hundreds of thousands of atoms. The GPU implementation has achieved significant speedups of more than 46 times for some large systems simulations.

AUTOBIOGRAPHICAL STATEMENT

Mr. Kamel Rushaidat earned his Bachelor and Master degrees in computer science from the Jordanian University of Science and Technology in Jordan. He worked as a software engineer and a technical trainer in many software companies before he started his PhD in 2009. As a Ph.D. Student, he worked in the field of High Performance Computing.

Mr. Kamel is a recipient of the following awards:

1. Thomas Rumble Fellowship, Wayne State (2009).
2. Graduate Teaching Assistant award, Wayne State (2010-2013).
3. Graduate Research Assistant, Wayne State (2013-2015).
4. Third place award for poster presentation, the 5th annual graduate exhibition, Wayne State University, presented March 18, 2014.

The following are selected publications by Mr. Kamel:

1. **K. Rushaidat**, L. Schwiebert, B. Jackman, J. Mick, and J. Potoff, "Evaluation of Hybrid Parallel Cell List Algorithms For Monte Carlo Simulation", In Proc. of The 7th IEEE International Workshop on Multicore and Multithreaded Architectures and Algorithms (M2A2 2015),New York, NY, Aug. 2015.
2. **K. Rushaidat**, L. Schwiebert, B. Jackman, J. Mick, and J. Potoff, "Efficient Parallel Cell List Algorithms for Monte Carlo Simulations," In Proc. of 2015 The Summer Simulation Multi-Conference (SummerSim'15), Chicago, IL, July 2015.
3. E. Hailat, **K. Rushaidat**, L. Schwiebert, J. R. Mick, and J. J. Potoff, "GPU-based Monte Carlo simulation for Gibbs ensemble," High Performance Computing Symposium, April 2013.
4. J. R. Mick, E. Hailat, V. Russo, **K. Rushaidat**, L. Schwiebert, and J. J. Potoff, "GPU-accelerated Gibbs ensemble Monte Carlo simulations of Lennard-Jonesium," Computer Physics Communications, 184 (12): 2662–2669, December 2013.
5. E. Hailat, V. Russo, **K. Rushaidat**, J. Mick, L. Schwiebert, and J. Potoff, "Parallel Monte Carlo Simulation for Canonical Ensemble on the GPU," International Journal of Parallel, Emergent, and Distributed Systems, 29 (4): 379-400, October 2013.
6. J. R. Mick, **K. Rushaidat**, B. Jackman, Y. Li, L. Schwiebert and J. J. Potoff, "Development of a GPU Optimized Gibbs Ensemble Monte Carlo Simulation Engine". AICHE, 2014.
7. J. R. Mick, **K. Rushaidat**, E. Hailat, Y. Li, L. Schwiebert and J. J. Potoff, "GPU Accelerated Configurational Bias Monte Carlo Simulations of Branched Molecules." AICHE, 2013.