1-2-2013

Advanced Optimization Techniques For Monte Carlo Simulation On Graphics Processing Units

Eyad Hailat
Wayne State University,

ADVANCED OPTIMIZATION TECHNIQUES FOR MONTE CARLO SIMULATION ON GRAPHICS PROCESSING UNITS

by

EYAD HAILAT

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

2013

MAJOR: COMPUTER SCIENCE

Approved by:

Advisor Date

DEDICATION

Dedicated to my parents, Majed Hailat and Amal Alshouha. To my wife, Maram, and to my little angel, Lilian. Also, I dedicate this work to my brothers and sisters and their families. Special thanks to all my friends.

ACKNOWLEDGMENTS

First, I would like to thank God almighty for giving me the chance to start and finish my Ph.D. and for all other good things happened to me in my life. Also, I am deeply grateful to my advisor, Dr. Loren Schwiebert for all the help and guidance during my Ph.D. study from day one. It is with his guidance, mentoring, and help the journey through graduate school was possible.

In addition, I would like to thank Dr. Jeffery Potoff, Dr. Hongwei Zhang and Dr. Weisong Shi for serving on my dissertation defense committee.

TABLE OF CONTENTS

Dedication . . . . **ii**

Acknowledgments . . . . **iii**

List of Tables . . . . **vii**

List of Figures . . . . **viii**

Chapter 1: Introduction **1**

1.1 Motivation 1

1.2 GPU Architecture 2

1.3 Structure of Fermi cards 3

1.4 CUDA Review 6

1.4.1 Synchronization and Concurrency 7

1.4.2 Kernel Launch Specifications 8

1.4.3 CUDA Memory 9

1.4.4 CPU-GPU Communication 9

1.4.5 Atomic Instructions on Global Memory 10

1.4.6 Memory Coalescing 10

1.5 CUDA Streams 10

1.6 Monte Carlo Simulations 12

1.7 Random Number Generator 14

1.8 Other Applications of this Work 15

Chapter 2: Related Work **16**

2.1 General Purpose GPU Programming 16

2.2 Monte Carlo Simulations 16

2.3 Domain Decomposition Techniques 21

Chapter 3: Porting Canonical Ensemble to the GPU 23

3.1 Markov Chain Monte Carlo Simulations 23

3.1.1 Metropolis Method and Thermodynamic Ensembles 23

3.1.2 Lennard-Jones Potential 25

3.2 Structure of the Code 25

3.3 Optimizing the canonical ensemble method for the GPU 29

3.3.1 The block size effect 29

3.3.2 The use of pinned memory 30

3.3.3 The use of different GPU memory types 30

3.3.4 Memory coalescing for fetching particle positions 31

3.3.5 Loop unrolling technique in finding total energy 31

3.3.6 Load balancing among threads and contributing particles 34

3.3.7 Atomic operations on global memory transactions 35

3.3.8 Block synchronization through global memory and atomic operations 35

3.3.9 Numerical optimizations: tricks and tweaks 35

3.4 Using cell list structure 37

3.5 Results and Discussion 39

Chapter 4: Grand Canonical Ensemble: One Simulation Box and a Reservoir 50

4.1 MC Simulation for the Grand Canonical 51

4.2 Parallel Algorithm and Implementation Details 52

4.2.1 Implementation without Cell List 53

4.2.2 Cell List Implementation 55

4.2.3 Assigning Cells to Blocks 60

4.2.4 Assigning Threads to Particles 62

4.2.5 Adding Cell List Implementation to the Parallel Grand Canonical Algorithm 62

4.3 Performance Results 62

Chapter 5: Gibbs Ensemble: Two Simulation Boxes **69**

5.1 MC Simulation of the Gibbs Ensemble 69

5.2 Method 71

5.3 Results and Discussion 76

Chapter 6: Configurational Bias Gibbs Ensemble **82**

6.1 Introduction 82

6.2 CBGEMC Method and Implementation 83

6.3 Results and Discussion 87

Chapter 7: Conclusion and Future work **100**

References **103**

Abstract **115**

Autobiographical Statement **117**

LIST OF TABLES

Table 3.1: Thread hierarchy and properties. 30
Table 3.2: Average program execution times (in seconds) and speedup over *Towhee*. 40
Table 3.3: Specifications for the three graphic cards used to run reported experiments. 41
Table 3.4: Desktop computers used for the experiments. 42
Table 3.5: Large vs. small block size and system performance. Numbers shown are speedup. . 44
Table 3.6: CUDA vs Serial results on GTX 480 47
Table 3.7: Results of different cell list models 48
Table 3.8: Speedup of cell list for a million steps 48
Table 4.1: Execution times in seconds for different algorithm implementations. 64
Table 4.2: Legend of blocks, cells, and threads per kernel call. 64
Table 4.3: Different block sizes have different effect on resource utilization. 66
Table 4.4: Speedup of different cell list implementations over no cell list CUDA code. 67
Table 5.1: A comparison of a high and a low end GPU 76
Table 5.2: A comparison of a high and a low end CPU 77
Table 5.3: Execution times in seconds and speedup using the icc compiler. 78
Table 5.4: Execution times in seconds and speedup using the gcc compiler. 78
Table 6.1: Execution times in seconds for CB for different number of CB trials (K). 90
Table 6.2: Execution times in seconds for CB for different number of CB trials (K). 91
Table 6.3: Execution time in seconds of running simulations with 131072 particles in parallel. 92
Table 6.4: Speedup of running simulations with 131072 particles. 93
Table 6.5: Results for CB for different number of CB trials (K) in parallel. 96

LIST OF FIGURES

Figure 1.1: Fermi Memory Structure 4

Figure 1.2: A grid structure 5

Figure 3.1: A particle displacement attempt in Metropolis Monte Carlo method. 24

Figure 3.2: MC Simulation for the canonical method flowchart. 28

Figure 3.3: Calculating the partial sum in shared memory that use adjacent memory locations. . 32

Figure 3.4: Mapping algorithm for work load balancing across threads. 34

Figure 3.5: The volume is decomposed into cells. 38

Figure 3.6: Cell with all 26 adjacent cells. 39

Figure 3.7: Plots of speedup for different block sizes on different platforms. 45

Figure 3.8: Developed serial code vs. *Towhee* elapsed times (logarithmic normalization). . . . 46

Figure 3.9: Serial vs. CUDA execution times for MC simulation on i5 and GeForce 480. 46

Figure 3.10: Execution times on two different platforms with GTX 480. 47

Figure 3.11: Different cell list implementations speedup 49

Figure 4.1: Different methods in assigning cells to thread blocks 59

Figure 4.2: Speedup of different algorithms 65

Figure 4.3: Speedup of CUDA code with different cell list codes over CUDA with no cell list. . 66

Figure 4.4: Execution times for different algorithms with CUDA cell list. 67

Figure 4.5: Cell size and speedup. of particles per cell per thread for N = 262144. 68

Figure 5.1: An illustration of the three move types for the Gibbs ensemble method. 70

Figure 5.2: Execution time for the serial and the CUDA code 79

Figure 5.3: The actual speedup and theoretical speedup of the CUDA code 80

Figure 6.1: Particle transfer move with CB and K equals five. 85

Figure 6.2: Timeline of particle transfer in CB execution using K-1 independent streams. . . . 85

Figure 6.3: Vapor-liquid coexistence curves for Lennard-Jones fluid. 88

Figure 6.4: Histogram sampling the distribution of the gas and liquid phases 89

Figure 6.5: Execution times for K equals 3 and 20 with different algorithms 94

Figure 6.6: Execution times of CUDA code on GTX 480 and K20c for large systems 95

Figure 6.7: Execution time for parallel code with only particle transfer move 97

Figure 6.8: Speedup for parallel code with streams. 98

CHAPTER 1 Introduction

1.1 Motivation

Graphics devices were introduced originally for entertainment. The first graphics processing unit was invented by NVIDIA in 1999 for creating game objects in real-time [106]. Since then, graphics units kept evolving to provide a huge amount of processing power that later was extended to do non-graphics applications such as floating point operations and developed a high level set of shading languages such as DirectX, OpenGL, and Cg.

When GPUs first started using APIs for non-graphics purposes, they were called General Purpose Graphics Processing Units (GPGPU), showing good speedups over other methods. In that era, efforts were spent on utilizing vertex coordinates, texture, and shader programs, which was one drawback of using GPUs for general purpose computing. Another drawback to using the early GPUs for scientific applications is the need for an extensive knowledge of graphics APIs and the GPU architecture itself. Moreover, the lack for even the most basic programming features and operations such as random access to memory for reads and writes, double precision floating point operations, and operations on integer numbers were major reasons for not using GPUs in many applications and considered limiting factors. Later, programming with graphics APIs was replaced with parallel programming languages. Examples of high level parallel programming languages are BrookGPU [15] used for AMD GPUs, Open Computing Language (OpenCL) [120, 82], which is portable between GPU architectures, CUDA [55, 87], which is limited to be used on NVIDIA cards, and BSGP [52]. In addition, Chapel, developed by Cray Inc. [21] is an open source, portable parallel programming language that can be used on commodity clusters or desktop multicore systems.

Although OpenCL and CUDA provide different programming interfaces, they offer similar features. However, OpenCL is designed for a very general architecture; it can be used to program CPUs, GPUs, Digital Signal Processors (DSPs) and other devices from other vendors. This portability came with a price that affects the performance of OpenCL codes. In a performance comparison conducted in [59] between CUDA and NVIDIA's OpenCL implementation, the study shows that CUDA performs better in terms of kernel execution and memory transfer from and to the GPU running almost the same code. In addition, several differences in using both architectures are reported. One main difference between these two languages is that OpenCL can be compiled at runtime, which would slow down the overall execution time of the code. This technique allows the compiler to generate code on the fly targeting a specific GPU architecture. CUDA, on the other hand, does not have this concern since it is designed specifically for a single architecture. Other differences in the two APIs include context creation, mapping kernels to the GPU, and memory copying. See [82] for a complete API reference for programming with OpenCL.

1.2 GPU Architecture

Here we will discuss *Graphics Processing Units* in terms of the architecture and its relationship with the rest of the computer system. Also, we will mention the difficulties that may confront anyone who is interested in utilizing these devices [44, 72, 83, 50, 51].

Even if we can add more transistors to a chip, we cannot scale their voltage like we used to, and we cannot clock these transistors as fast. One solution is to run in parallel. With current GPU technology we can easily get a device loaded with hundreds of processors, and the future trend is even more. So, one should start thinking in parallel and start developing applications without thinking of the serial algorithm first.

There are some weak points for GPUs compared to CPUs in terms of architecture. Examples include that GPUs devote more transistors to arithmetic logic units and less to caches and control prediction in comparison to CPUs. On the other hand, threads in GPUs are considered fine-grained and without expensive context switching like CPU threads. Individual GPU threads are considered to have poor performance compared to CPU threads, but thousands can run simultaneously. Moreover, GPUs have higher memory bandwidth.

Lately, tools have been developed to make the programmer's parallel experience easier. IDEs such as Eclipse, NetBeans, and MS Visual Studio started to allow for integrating parallel programming languages. Also, debugging tools for GPU code started to appear after developing the latest architecture, such as CUDA-GDB and Parallel Nsight for different Linux operating systems, and MS Visual Studio integration for Windows operating systems. In addition, other tools appeared such as Glift, a data structure framework that implements a set of structures that aims to simplify algorithmic development using GPUs. Structures implemented in Glift include a stack, quadtree, and octree [68]; more tools can be found in [53, 115, 116].

In addition, GPU computing became mainstream with the launches of MS Windows 7 and Apple Snow Leopard, so that the GPU will be accessible to any application as a parallel processor, not only as a graphics processor.

The host program performs all memory management,thread synchronization, and other setup tasks and then calls GPU kernels to perform the simulation. Examples of CUDA enabled devices are the GeForce 6 series GPU, their architecture description can be found here [60], and Tesla graphic cards [69].

1.3 Structure of Fermi cards

In June 2008, NVIDIA released a major revision to their architecture. Graphics cards like GeForce GTX 280, Quadro FX 5800, and Tesla T10, were the first cards to have the new unified hardware generation of GT200. This major revision has many updates over the previous one, such as increasing the number of streaming processor cores to almost double. Each register file was also doubled in size. In addition, there is better memory access coalescing that improves memory access efficiency for huge amounts of data. Moreover, double precision floating point support was enhanced to support high performance scientific computing. This NVIDIA generation of CUDA Compute and Graphics architecture is called **Fermi**.

The structure of a graphic card's memory is different from any other device, as seen in figure 1.1. Fermi structure is different from serial processor's memory and older graphic cards. There are two lines of eight Streaming multiprocessors (SMs) around one L2 cache. A host interface connects the GPU to the CPU via a PCI-Express interface. Each SM, see figure 1.2, has 32 cores. Each has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). One CUDA core executes a floating point or integer instruction per clock for a thread.

This memory model was adopted to take the advantage of different threads running the same code. A thread is identified by a `threadIdx` that consists of three coordinates inside the block: `threadIdx.x`, `threadIdx.y`, and `threadIdx.z` that define the `x`, `y`, and `z` coordinates, respectively. On current GPUs, a block can contain up to 1024 threads for x- or y-dimension in compute capability 2.0 and 512 for earlier compute capabilities, and 64 threads for the z-dimension for any compute capability. On the grid level, a three-dimensional grid of blocks is defined referencing the block by `blockIdx.x`, `blockIdx.y`, and `blockIdx.z` variables that are assigned for the block by the CUDA run time system, cannot be changed

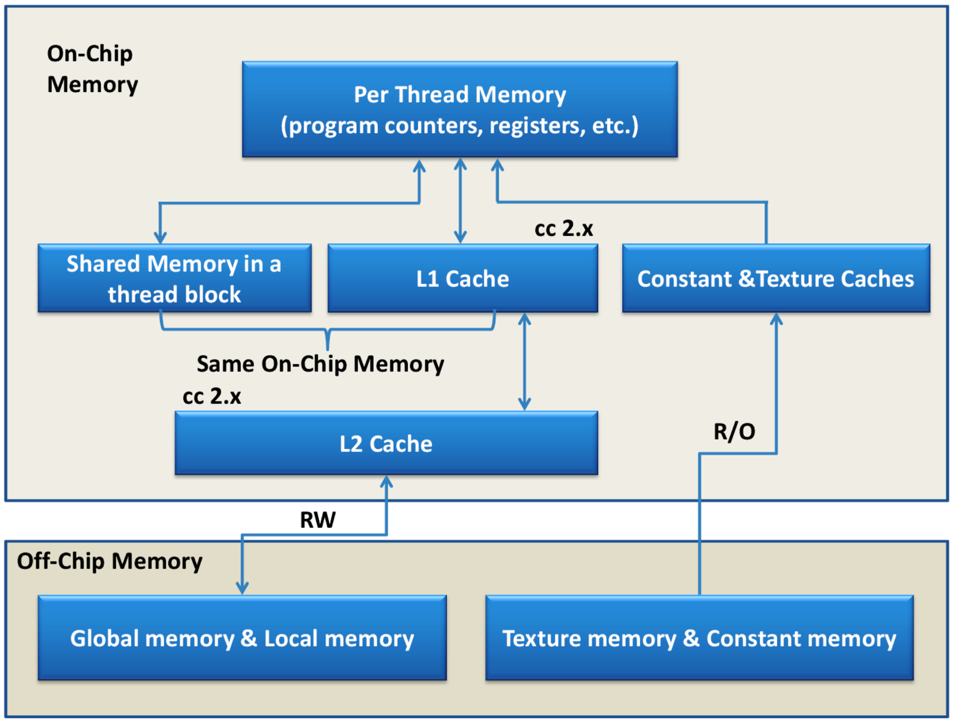

On-Chip Memory
Per Thread Memory (program counters, registers, etc.)
cc 2.x
Shared Memory in a thread block
L1 Cache
Constant &Texture Caches
Same On-Chip Memory
cc 2.x
L2 Cache
R/O
RW
Off-Chip Memory
Global memory & Local memory
Texture memory & Constant memory

Figure 1.1: Fermi Memory Structure

afterward, shared by all threads in that block, and are accessed only inside kernels. The number of blocks in a grid on one of the two dimensions can vary between one and 65,536, and are saved in variables `gridDim.x` and `gridDim.y` for the x- and y-coordinate, respectively.

The Fermi memory hierarchy, shown in figure 1.1, illustrates that a thread can use both shared memory and cache. This depends on the nature of the problem. The advantage of this memory model is *scalability* of blocks. A programmer may have a limit on the number of threads in different GPUs but the number of blocks makes it more scalable.

For example, in the code portion below

Listing 1.1: Legal code portion of CUDA kernel launch

```
dim3  BlockSize(3,3,2);
dim3  GridSize(6,4,1);
LaunchKernel<<<GridSize, BlockSize>>>(parameters);
```

the first two statements are declaration statements for a two-dimensional grid with $6 \times 4 = 24$ blocks, each

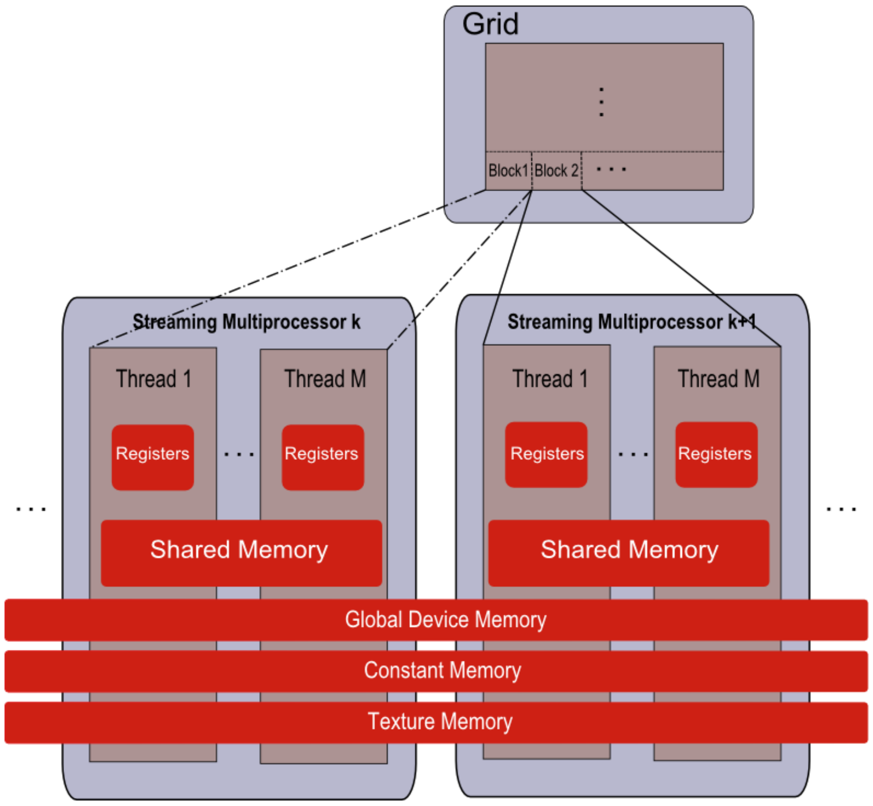


Figure 1.2: A grid structure

block is three dimensional block with a total of $3 \times 3 \times 2 = 18$ threads per block, this makes it a total of $24 \times 18 = 432$ threads that launch this kernel. On the other hand, a code portion like the listing below (1.2) cannot be executed since the number of threads exceeds the allowed number of threads.

Listing 1.2: Illegal code portion of CUDA kernel launch

```
dim3   BlockSize(32,32,2);
dim3   GridSize(6,4,1);
LaunchKernel<<<GridSize, BlockSize>>>(parameters);
```

It is a design decision on how to choose block size. To illustrate, if the number of threads is divided among more blocks, the total number of threads would be the same, but may have a disadvantage of extra overhead when trying to synchronize between blocks through global memory instead on synchronizing in shared memory in the case of working with threads in one block. In addition, there is a limit on the number of blocks that can run concurrently on any SM. In our study we have examined this factor on the system behavior. We study the performance when changing block size for the Canonical Ensemble method.

1.4 CUDA Review

Invented by NVIDIA, the *Compute Unified Device Architecture* (CUDA) parallel computing architecture is now shipped in most NVIDIA graphic cards. This architecture is a hardware and software platform that defines the programming model, memory model, and execution model for issuing and managing computations on GPUs, where the programmer does not need to directly map computations to the graphics pipeline. Compared to previously used graphics languages such as OpenGL and Cg, this architecture is implemented over the C language, which makes the process of developing GPU-based software for scientific computing a much easier job. Basically, CUDA facilitates heterogeneous computing on both the CPU and the GPU by following a general approach:

1. Copy input data from CPU memory to GPU memory.
2. Load the GPU program, called a kernel, and execute. This step may include caching data on chip for enhancing the performance of the GPU.
3. Copy results back from GPU memory to CPU memory.

CUDA has been widely used in many applications around the world, in scientific research, financial market, medical, and many more fields. There are more than 700 GPU clusters installed around the world [55]. In this work we are using the latest CUDA release 5.0. It has support for CUDA-GDB, Visual profiler, unified virtual addressing, N-copy pinning of system memory, etc.

CUDA can be used in two different ways, one way is using the driver API, which provides the programmer with language tools close to the hardware. This method needs more coding and programming effort. Second, the runtime API, an extension to the C programming language, provides the programmer with an easy-to-use set of C functions and extensions, making writing parallel programs relatively easy to learn and apply without the hassle of learning a new language or the underlying pipeline design. However, a programmer needs to be knowledgable of the programming, memory, and thread models of the architecture to best utilize them for the problem of study.

Another issue a programmer must pay attention to is that in the CUDA architecture there is no dedicated initialization function in the runtime API. The device will be initialized, however, the first time a runtime function is called. If timing is being recorded, this property should be taken into consideration and especially when interpreting the error code from the first call into the runtime. So, a statement like `cudaDeviceReset()` can be used at the beginning of the code to initialize the device and record any initialization problems before starting the timer.

1.4.1 Synchronization and Concurrency

Due to the hierarchal thread model of the GPU, threads need to be synchronize on all levels. For example, synchronizing threads in a block is done using instructions such as $\_\_synchthreads()$, $\_\_threadfence()$, $\_\_threadfence\_block()$, and others that can be found in the "NVIDIA CUDA C Programming Guide" [87]. One drawback to the CUDA architecture is that it has no efficient global synchronization. Mainly for two reasons: it is expensive to build this support into hardware for GPUs with this huge number of processing units, and due to the potential deadlock that may occur if we use more blocks. The solution to this problem is to use atomic operations on global memory that is being accessed from all blocks in a grid.

With the CUDA compute capability 2.x devices, concurrent operations can be done through streams. In other words, the GPU can start doing a memory transfer while the CPU is doing other operations. To illustrate, the listing below (1.3) shows a GPU memory copy operation that transfers an array from the device to the host while the CPU is generating random numbers. In this case both the GPU and the CPU are doing different work.

Listing 1.3: Asynchronous memory call leads to overlapping of data transfer and CPU computation

```
cudaStreamCreate(&Stream1);
cudaMemcpyAsync(ParticleXCoord, dev_ParticleXCoord, NParticles *
    sizeof(double), cudaMemcpyDeviceToHost, Stream1);
//Fill array with random numbers
for (int i = 0; i < Size; i++)
        RandomNumbers[i] = RandomNumberGenerator->rand();
```

The programmer should be careful in doing such operations, especially when attempting to execute a piece of code that needs the final output from the GPU. In such a case, a barrier statement, such as `cudaDeviceSynchronize()`, is needed on the host side to make sure that all device operations are done. In addition, this statement is needed when it is necessary to synchronize the CPU thread with the GPU to accurately measure the elapsed time for a particular call or sequence of CUDA calls.

1.4.2 Kernel Launch Specifications

CUDA allows the programmer to launch a function that will be executed in parallel on the device by several threads; this special function is called a *kernel*. To create a kernel, one must use the `__global__` qualifier before the function declaration, and specify the *execution configuration* for the call, which means how many threads, blocks, dynamic shared memory, and the stream number to execute the kernel in the form of `Kernel<<<GridSize, BlockSize, NSize, StreamP>>>(args)`, where grid size and block size are of type `dim3`, which is a structure that has three variables x, y, and z, which all define dimensions of a grid in the first case and the block size in the second case. The third argument in the list is optional and indicates the number of bytes in shared memory that is dynamically allocated for each block for this kernel call. The last argument in the execution configuration of a kernel is the stream handle. The stream handle is of type `cudaStream_t` and refers to the stream that is associated with this kernel call. It is an optional argument and the default stream is 0. An example kernel function declaration is `__global__ void Myfunc(dataType *variable1,...)`. In this case, the statement to call this kernel may look like `Myfunc<<<GridSize,BlockSize,NSize,StreamP>>>(arguments)`. Keep in mind that the only way for the GPU and CPU to communicate is through memory calls, so a kernel always returns nothing.

1.4.3 CUDA Memory

Memory usage has been a factor of success in the invention of GPUs. There are several levels of memory access on the GPU. Figure 1.1 shows that there is:

Per-Thread local memory The access to this memory is local to a specific thread in the kernel only. And it is used for local variables.

Per-Block shared memory This memory can be accessed by any thread in a block; other blocks cannot access this memory. Shared memory is allocated using the qualifier `__shared__`. This is an on-chip memory that is much faster than local and global memory. In terms of performance, the latency for uncached shared memory is about $100\times$ faster than global memory in the best case when there are no memory access conflicts.

Shared memory is partitioned into 16 banks that are organized such that successive 32-bit words are assigned to successive banks, i.e. interleaved. Each bank has a bandwidth of 32 bits per two clock cycles. If threads in the same warp are trying to access different memory locations in the same bank, then there is a conflict. In this case the accesses are serialized. This will decrease the effective bandwidth by a factor equal to the number of distinct memory requests to the same bank.

Global device memory This includes global, constant, local, and texture memory space that are persistent across kernel launches in the same file scope. Constant memory is a cached read-only memory. Reads from this memory cache could be as fast as reading from registers if all threads in a warp read from the same address; otherwise, the latency increases linearly with the number of read requests.

1.4.4 CPU-GPU Communication

GPU and controlling CPU code communicate through memory copies. This mechanism can really affect performance if it is misused due to memory latency. Actually, this can be a limited factor for the performance that parallel processing can provide. So, it is very important to keep CPU-GPU communication to a minimum. However, this communication between CPU and GPU is most efficient if pinned memory is used on the CPU. This is because pinned memory enables asynchronous memory copies (allowing for overlap with both CPU and GPU execution), as well as improves PCIe throughput on FSB systems.

1.4.5 Atomic Instructions on Global Memory

An atomic instruction is an instruction that is guaranteed to be executed in full without interruption from other threads. Atomic instructions are specific device functions that execute read-modify-write atomic operations on a global or shared memory location on mapped page-locked memory. There are many atomic instructions and functions. For example, `atomicAdd()`, `atomicSub()`, `atomicInc()`, `atomicDec()`, `atomicExch()`, `atomicCAS()` (Compare And Swap), `atomicAnd()`, etc. Atomic instructions on global memory are supported only on devices of compute capability 1.1 and above.

Note that atomic functions operating on mapped page-locked memory are not atomic from the point of view of the host or other devices. So, if another non-atomic instruction executed by a warp reads, modifies, and writes to the same memory location, then the read, modify, and write to that memory location occurs in a random order. This depends on the compute capability of the device, and which thread performs the final write operation is undefined.

1.4.6 Memory Coalescing

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more memory transactions depending on the size of the data accessed by each thread and the distribution of the memory addresses across the threads. Throughput can be really affected by this. When we have more transactions with unused words being transferred, we have wasted bandwidth. On the other hand, if we have fewer transactions with fewer wasted words, this is going to increase the overall performance of the system.

1.5 CUDA Streams

A CUDA stream represents a queue of operations that are executed in order. Examples of such operations are kernel launches, memory copies, and event starts and stops. Operations in a stream are executed in the same order that they have been called and multiple streams can be run in parallel on the device. Running multiple streams at the same time on the device adds an extra level of parallelism to the GPU. However, operations must be independent to run in parallel. Keep in mind that different streams may execute their operations out of order with respect to one another or concurrently. For instance, a kernel call and memcpy from different streams can be overlapped. This behavior is possible because the inter-kernel communication is undefined.

Not all CUDA devices allow concurrent stream execution. For example, not all devices of compute capability 1.1 and higher can perform copies between page-locked host memory and device memory concurrently with kernel execution, and only devices of compute capability 2.x and later support the execution of concurrent kernels [87]. This property is device dependent and can be determined by querying the device at run time. Even if a device does support concurrent kernel execution, that doesn't guarantee the device will execute kernels concurrently with other kernels. There must be sufficient resources to run concurrent kernels, which may not be possible if, for example, that kernel is using many textures or a large amount of local memory. When no stream number is explicitly specified, or set the stream parameter to zero, the default stream is used. This will guarantee that all operations executed and in the serial order they appear in.

CUDA streams can be created using simple statements and then be used to run kernels. For example, Listing 1.4 shows the required steps to run multi-stream kernel calls along with memory copy and other host functions. An array of stream handlers is created, three in this example, and used with each kernel, as in lines 2-3. Then streams can be used to run multiple concurrent operations. For example, lines 4-5 will launch three kernel calls on the three streams created before. In line 6, a host function call is executed after the for loop is done. This function will be called before any of the kernel calls return, with no guarantee that this function will be executed before any of the kernel calls. Following that, the statement in line 7 places a request to perform a memory copy into the stream specified by the argument stream, which means that the execution of the memory transfer will start only when stream one returns from executing all proceeding operations on that stream.

To guarantee that the GPU is done with its computations and memory copies, the stream should be synchronized with the host. This can be done for each individual stream using the statement in line 9, or using *cudaDeviceSynchronize*() to synchronize all device operations. Finally, to release the stream handlers, a statement such as the one used in line 12 should be used. In general, as we can see from this example, a stream acts as an ordered queue of operations for the GPU to perform. Other examples on using streams for different execution patterns can be found in [87].

Listing 1.4: An example of using streams to run concurrent operations.

```
cudaStream_t StreamArr[3];
for (int StreamId = 0; StreamId < 3; StreamId++)
    cudaStreamCreate(&StreamArr[StreamId]);
for(int StreamId = 0; StreamId < 3; StreamId++)
     ConcKernel<<<GSize,B, ShMem, StreamArr[StreamId]>>>(StreamId,...);
CallHostfunc();
cudaMemcpyAsync(..., StreamArr[1]);
for (int StreamId = 0; StreamId < 3; StreamId++)
    cudaStreamSynchronize(StreamArr[StreamId])
for (int StreamId = 0; StreamId < 3; StreamId++)
     cudaStreamDestroy(StreamArr[StreamId]);
```

Although the Fermi architecture supports 16-way concurrent kernel launches, there is only one connection from the host to the GPU. So even if we have sixteen CUDA streams, they'll be scheduled through one hardware queue. This can create false data dependencies and limit the amount of expected concurrency.

1.6 Monte Carlo Simulations

The affordability of Graphics Processing Units (GPUs) has made high-performance computing more accessible and financially practical. Furthermore, the growth rate of the computing power in the GPU is more than that for the CPU, so, the GPU offers significant speedup in execution for certain applications. Although both software and hardware developments for GPUs have enabled more high performance computing applications than ever before, writing optimized algorithms and code to utilize these devices remains time consuming and intensive. In this chapter, we describe the development of an efficient GPU implementation for the Monte Carlo simulation of molecular systems in the canonical ensemble method.

Potential functions have long been used in physical simulations to describe the collective or local behavior of molecules in condensed systems. The chief limitation to simulation of physical systems using potential functions is computational cost, a limitation that can be overcome with high performance parallel computing. To study atomistic systems, computer simulations are considered valuable substitutes to lab experiments to get information on the liquid or gas states of chemical compounds and mixtures [23]. Two approaches have been of particular interest to a number of researchers, Monte Carlo (MC) and Molecular Dynamics (MD) simulations [9, 39]. Markov chain MC simulations allow the study of open systems, which are infeasible for a traditional MD code.

An example of a system well suited for MC simulation is the adsorption of gases in porous materials, such as activated carbons. MC simulations can accomplish the simulation of the open porous system via trial moves that allow the number of particles to fluctuate. Additional examples of Monte Carlo simulation include:

1. **Prediction of physical properties and phase behavior.** This application is primarily of interest to chemical process industries. For example, given a mixture of compounds, the goal is to predict accurately the coexistence properties of the gas and liquid phases.

2. **Prediction of adsorption isotherms for gases in porous materials.** Typical applications for this are CO_2 sequestration from flue gas, and hydrogen or methane storage. With a fast enough code, one could potentially carry out high throughput screening of candidate materials [30].

3. **Simulation of biological systems at constant chemical potential.** Simulations of the fundamental biomechanical process of membrane fusion have shown divalent cations and water molecules to play a critical thermodynamic role [56]. In order to use simulations to understand this fundamental process that occurs in all living organisms, it is critical to maintain constant ion and water molecule chemical potential to achieve realistic local densities.

4. **The use of nanoparticles to stabilize drug dispersions.** Simulations of nanoparticle dispersions also typically require a constant chemical potential, so that as the microparticles approach each other, the number of nanoparticles varies to maintain chemical equilibria with the bulk. This is a very important application for this work because large system sizes are required to simulate interacting microparticles.

On the other hand, calculations of the Lennard-Jones potential are significantly more complex than the Ising or hard sphere model, since you have to calculate the interactions between all particles within a certain cutoff radius and it requires more work to optimize the calculation on the GPU. One have to account for atoms being in a molecule (and not calculating those interactions); this will also require the calculation of bending and stretching potentials, the generation of multiple trial locations.

Monte Carlo simulations are driven by statistical physics based on energetics, thus it is necessary to pick a potential model to accurately model the studied compound. Perhaps the most common potential model used to describe interactions between particles is the *Lennard-Jones* potentials. While this model is mathematically straightforward, simulating even relatively modest systems requires a substantial amount of computing power. This is due to the tens of millions of iterations required for the Monte Carlo simulation to converge to a solution. With the advent of CUDA enabled GPUs, this previously held shortcoming is now being exploited to more scientists and researchers with smaller monetary and computational resources.

MC implementation could use the grand canonical ensemble method that can be useful in adsorption studies where the amount of material adsorbed is given as a function of the pressure and temperature of the reservoir with which the material is in contact. Moreover, in an interfacial region, gas and adsorbent for instance, the properties of the system are different from the bulk properties, which is a problem if simulating a relatively small system. Hence, we have to simulate a very large system to minimize the influence of this interfacial region [103, 39].

Due to the limited number of parallel operations in a multicore implementation of this algorithm, it is not expected to produce more speedup than a manycore system would produce. Hence, the effort is directed toward manycore technology that provides more parallelism for this algorithm.

1.7 Random Number Generator

Monte Carlo simulation relies on random numbers to compute their results and find probability statistics to investigate problems. We have used the *Mersenne Twister* (MT) random number generator algorithm [77] for generating uniform pseudorandom numbers, which provides a period of $2^{19937}-1$ and a 623-dimensional equidistribution property. Uniform random numbers is a very important part of the correctness of Monte Carlo simulation. If, for example, the sampling is not performed well, then it may result in execution in a limited region of the conformational space. This will result in a small statistical error, but a large systematic error.

The same random seed is used each time to verify the algorithm behavior. However, the user can change a flag that is responsible for a different seed. For now, we have been using the serial version of Mersenne Twister which has the same speed as `rand()` in C [77].

1.8 Other Applications of this Work

This thesis reports the algorithmic changes, optimization techniques, and tricks and tweaks one can use to implement thermodynamic Monte Carlo simulation on GPUs. These optimization techniques can be used for other Monte Carlo simulations such as the problem of Bias Monte Carlo Methods in Environmental Engineering [81], where a number of factors need to be studied at each simulation step. The use of the GPU may reduce the amount of computation time that this algorithm requires since an intensive calculation for different factors is being computed each simulation step. Environmental factors can be mapped to blocks and then results can be aggregated from all blocks in a similar way to the technique we describe in Figure 3.3. Another example is when running multiple environmental setups simultaneously on the GPU. See lessons learned for high level parallelism in § 6.2.

A second application that can benefit from this work is when a Monte Carlo procedure is applied to emulate a biochemical experimental measurement setting along with given enzyme kinetic reactions [81]. Such a system can simulate continuous enzyme assay, which is used for adjustment of the ”experimental” conditions, and end-point enzyme assay ”measurements”. This last case is suitable for parameter identification. While trying to enhance performance of this simulation and to better manage the GPU resources for this domain, a technique such as the one explained in Figure 3.4 to calculate the interaction in enzymes, and the results of the size of thread block can be used with this application. Moreover, memory management and the techniques used to use more shared memory over global memory could be applied here.

CHAPTER 2 Related Work

GPUs have been an affordable alternative to supercomputers and expensive clusters of networked computers. These devices can be installed in commodity computer desktops to run computationally intensive applications with minimal installation effort. This attracted the attention of researchers as well as average users with computationally intensive applications such as movie rendering and image processing. This chapter presents an overview of the most recent work using GPUs, focusing on the work that uses GPUs for implementing Monte Carlo simulations.

2.1 General Purpose GPU Programming

Many algorithms of a parallel nature or needing a great deal of mathematical computation have been ported to the GPU. Algorithms for applications in almost all fields of real life have started thinking of harnessing the power of this cheap technology. For example, protein folding [122, 95], stock pricing [62, 98], sorting and searching [6], SQL queries [7, 22, 54], MRI reconstruction [114, 113], image processing [13] and real-time image processing [29, 123], game physics [119, 20, 45, 64], video processing [46, 124], ray tracing [26], sequence matching (Hidden Markov Models) [123], system-level design tasks (high-level timing analysis) for embedded systems [12], Monte Carlo simulation for different applications [34, 35, 49, 3, 4, 5, 11, 58, 74], Molecular Dynamics simulations [111, 118, 43, 14, 41], mathematical and biological simulations [65, 66, 117, 16, 25, 27, 29], graphs [42, 17], MATLAB [63], and many more.

2.2 Monte Carlo Simulations

The literature illustrates numerous uses of MC methods for a very broad area of applications. For example, applications of MC in Science and Engineering, Quantum Physics, Statistical Physics, Reliability, Medical Physics, Polycrystalline Materials, Ising Model, Chemistry, Agriculture, Food Processing, X-ray Imaging, Electron Dynamics in Doped Semiconductors, Metallurgy, Remote Sensing and many more [81].

In chemistry, computer simulations are considered a valuable substitute to lab experiments to get information on the liquid state of material [23]. Two approaches have been of interest for researchers all around the globe: Monte Carlo (MC) and Molecular Dynamics (MD) simulations [9]. The computational cost of such simulations limits the complexity of potential functions to describe the collective or local behavior of molecules in the condensed systems. However, this limitation has been reduced lately by an impressive increase in computer performance.

Molecular dynamics codes exist, some of which have been modified to utilize the GPU, including LAMMPS [14], NAMD [97], AMBER [107], and HOOMD-blue [5], which was developed from scratch to support the GPU. However, existing GPU-enabled MD codes are inadequate for many biomolecular systems of interest, which require the simulation of an open system. The Monte Carlo method is the ideal technique for this class of biomolecular systems. While systems containing more than 100,000 atoms are routinely simulated with molecular dynamics, Monte Carlo simulations are typically limited to systems containing less than 2,000 atoms.

GPU-driven Monte Carlo simulations of chemical systems have been performed, using lattice gauge theory [19], Ising models [102], and simulations of hard spheres [40]. An Ising model is essentially a spin-flip model. Spins are arranged on a cubic lattice and can have the value +1 or -1. The total energy of the system is the sum of nearest neighbor interactions. In two dimensions, each spin has four interactions, in three dimensions it's eight interactions. All of the possible interactions can be precalculated, so this problem is essentially reduced to running a fast lookup table, although it might actually be faster to do the calculations [102].

Hard spheres is a very simple model where one simply tests for overlap. If the particles overlap, the move is rejected, otherwise, the move is accepted. Hard sphere simulations are typically used to understand colloidal phenomena [31].

The Lennard-Jones system is the most basic model of a "real" fluid [94]. In fact, CH_4 (methane), Xenon, Neon, Argon, and Krypton can be modeled to high accuracy using a single Lennard-Jones bead. The Lennard-Jones model is used as the basis for models of realistic fluids, such as alkanes, alcohols, sugars, proteins, etc. In these cases, multiple beads are combined to form molecules, where each bead represents a single atom. The various Lennard-Jones parameters (epsilon and sigma) are optimized to reproduce experimental data. Examples of the fitting process can be found in [101].

Calculations of the Lennard-Jones potential are significantly more computationally expensive than the

Ising or hard sphere models, since the interactions between all pairs of particles within a certain cutoff radius, r_{cut}, must be calculated. Most recently, a work was published using lookup tables for the canonical ensemble simulation, which focuses on a small size system (N = 128) [61] using the embarrassingly/pleasingly parallel algorithm [2] of multiple identical lightweight single thread simulations. A mapping of one thread per methane-MFI in used. Moreover, in the Lennard-Jones algorithm, a block per each of the methane-MFI, waste recycling with multiple uniform proposals, and waste recycling with multiple displacement proposals is being chosen. However, the authors suggest this approach may be limited for larger atomistic systems. In this work, we present an alternative off-lattice GPU-enabled algorithm for the chemical simulation of Lennard-Jones particles, based on the heavily multithreaded principle of energetic decomposition, also known as the "farm algorithm" which early CPU-based parallel computing studies [125] suggested, but produced insufficient performance. Note that the GPU architecture requires a reexamination of the older algorithms that have been deemed inefficient on CPUs.

Although MD simulations have been studied by more researchers [1, 57], other systems are impossible to simulate using these MD codes, such as the simulation of multi-component adsorption in porous solids [78], which will open the door for applications 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 [18, 96] because of the fluctuation property of MC that MD does not utilize.

A recent work on MC simulation on the GPU for systems of hard disks can be found in [31]. In this method a spatial decomposition technique is used, where multiple particles of short range interaction are moved at the same time in a "sweep" with the space divided so that detailed balance is not violated. To reduce the overhead of unnecessary calculations, a cell list implementation is used where the problem domain has been divided into nine cells. Maintaining a detailed balance in this algorithm adds extra overhead to the original algorithm and to the process of verifying results. For example, shuffling the checkboard set at each sweep step and another shuffling at the particle level in each cell are required to maintain the detailed balance. Another restriction of this method is that the center of a particle shouldn't leave the original cell. In the first example, if the particle shuffling is not being executed, a temporal memory of previous states accumulates through different sweeps, which will result in a violation to the aforementioned properties. While this algorithm has been conducted only for 2D systems, scaling from 2D checkerboarding to 3D checkerboarding is a very difficult task. Moreover, it is not clear if this method can implement other MC methods where the system size changes.

A similar effort of a large scale system is the *Highly Optimized Object-Oriented Many Particle Dynamics* (HOOMD) engine. A MD simulator was created by Ames Lab [5] in collaboration with Iowa State University, and later adopted by the University of Michigan (HOOMD-blue) to perform molecular dynamics simulations utilizing GPUs. HOOMD-blue utilizes CUDA at its core, and additionally showcases many of the innovations expected of a modern reworking for a simulation engine. Simpatico [1, 91] is an extension to HOOMD-blue that has been added with limited support to MC simulations. An implementation for MC and MD simulations on single processors has been developed, and only the MD simulation code runs in parallel. Later, a hybrid MC-MD method has been used to implement the MC simulation, where an outer wrapper has been added to the MD simulation to simulate MC on the GPU. However, Simpatico requires installation and configuration of several modules to integrate the MC simulation with the MD, and even more modules and configurations are required for the GPU implementation [91]. Moreover, HOOMD-blue integration is limited to bond and non-bonded pair potentials, and doesn't work with angle, dihedral link potentials [91], and only works for short range interactions.

LAMMPS, which stands for Large-scale Atomic/Molecular Massively Parallel Simulator [14, 67], has been developed at Sandia National Labs since 1995. Although its goal is to develop a classical molecular dynamics simulation code to run on parallel computers, a limited MC implementation has been added to support a hybrid MD-MC method to enable canonical and grand canonical MC simulations. In addition to the problem of hybrid MD-MC methods that they may fall in a local minimum, this implementation of LAMMPS uses neighbor lists that are re-built every time step, which adds significant execution time to the simulation. A time step executes N move attempts and N should not be set to a small value by the user. This has a tradeoff if not set properly. If the neighbor rebuild is not done often enough, this will invalidate the results, since atoms can move beyond the neighbor list skin distance. In fact, this may affect the overall precision of the system and end in incorrect results.

A parallel implementation of a hybrid MC simulation has been conducted in [71] to parallelize the configurational bias in MC Gibbs ensemble simulations. In their work, a simple way of parallelizing the simulation has been used by distributing the Q simulations over the Q processors along with a sequence of random numbers for each simulation. For each particle transfer move in their algorithm, each processor calculates a total of $Q \times N_{trials}$ [1] random trials in the new box, and a total of $Q \times (N_{trials} - 1)$ in the old one. Then the probability of acceptance is calculated globally for all values from all processors in a serial step. The trial with the lowest energy is chosen. This method assumes that the simulation starts with an equilibrated state, which usually requires less computation. However, such an algorithm maximizes the inter-process communication and doesn't take into account all systems states. Moreover, the total number of trials is dominated by the number of processors Q and there is no fault tolerance if one of the Q processors fails.

To parallelize the displacement move, the work in [71] executes multiple displacement attempts (similar to the ones in an MD simulation) and the use of a technique to maintain a constant temperature is required for this technique. In this hybrid method, the MD technique is used to obtain trial configurations after initial momenta are drawn from a Gaussian distribution. The main drawback for such a hybrid implementation is when there is a local potential minimum due to high energy barriers. In this case, a global minimum may not be realized, since the simulation may get stuck in a local minimum. Only ten processors are used and a factor of four times speedup has been achieved.

The grand canonical method is being widely used to observe the amount of material adsorbed as a function of the pressure and temperature of the reservoir with which the material is in contact. Other simulation techniques such as MD simulations typically have an order of magnitude increase in computation time compared to MC simulations and are possible for only very simple systems. This is due to the fact that MD simulations requires a reevaluation of all pair forces at each time step, and for a large system this requires significant computing resources.

One technique for parallelizing the grand canonical ensemble is to run numerous small independent simulations at the same time. This has been referred to as an embarrassingly parallel algorithm since it is inherently parallel because a set of independent simulation instances can be carried out simultaneously without affecting each other. This algorithm works better for short range particles and becomes problematic for long equilibrated systems. In [61], two algorithms have been implemented to carry out MC simulations on GPUs using the embarrassingly parallel algorithm. First, a method originally used in [38] to enhance the MC simulation by sampling configurations that are normally rejected. To parallelize this method, multiple possible MC steps are done by threads and waste recycling is then used to collect history information from

[1] N_{trials} is the number of CBMC trials referred to as K in Chapter 6.

both the chosen state and the rejected states. With MC simulations, a waste recycling implementation is more straightforward than the MD implementation, where multiple time slice estimators are used to implement this technique. Such techniques focus on the energy and the force evaluations of the proposals that are generated successively by the molecular dynamics. Moreover, another method used is trial states based on displacement random-walk steps. For each step, a set of position proposals from an old to a new location are randomly picked with the same probability of any two sets being selected. This is a requirement to obey the detailed balance. Then the same algorithm used above is used to accept particle moves.

A drawback for this class of algorithms is when sampling configurations that are normally rejected in the case of a dense systems. In this case, the extracted information from nonlocal moves is very low. Moreover, while their implementation depends on the embarrassingly parallel algorithm, our work uses the energy decomposition method (farm algorithm). Although the former method uses several simulations independently with small systems of 128 particles, our code runs for systems of up to 262144 particles.

To the best of our knowledge, there is only one open-source Monte Carlo code (Towhee) [76], and there are no open-source Monte Carlo codes that utilize GPUs to this scale. As a result, only small problem sizes can be run in a reasonable amount of time and this constrains the size of Monte Carlo simulations. It should be noted that attempting to modify code bases such as *Towhee* to include GPU-enhanced functionality would require a large dedicated effort with significant time investment. In addition, rewriting the algorithm usually requires substantial modifications to the core design of the serial algorithm.

2.3 Domain Decomposition Techniques

Domain decomposition algorithms, such as cell lists, neighbor lists and simply dividing regions into subdomains, are used to minimize the amount of unnecessary calculations with particles outside the particle's cutoff. Neighbor lists are used to hold information about each particle's neighboring particles. Only recently, an efficient implementation of neighbor list on the GPU was viable to implement mainly because of the lack of atomic operations implemented for the GPU in [61]. Implementations such as the work in [5, 112, 121] use the CPU to implement the neighbor list and move it to the GPU. This implementation requires an update for almost each simulation step with the new particle position, cell information, and neighboring particles. One such implementation can be found in [1]. The main drawback that prevented this approach from being used with MC simulation in the past has been the small size of the systems being simulated, which adds more computing overhead more than the clock cycles that neighbor lists could save.

In [5], a MD implementation has been developed to use massive parallel devices and a neighbor list algorithm has been used to reduce the overhead of beyond the cutoff interactions. In their implementation of neighbor list, the domain is divided into cells with each dimension equal to the cutoff value. Then, binning the cells which is placing each particle in it's corresponding cell. After that, using a recursive algorithm, all particles in all cells are examined and placed in a list of visited list of particles. A list containing those particles separated by a minimum image distance less than the cutoff for all particles in the system has to be built from that data structure. However, to minimize the overhead of maintaining this list, the list is not updated until a particle is displaced outside of a skin that is given by $\frac{1}{2}(r_{max} - r_{cut})$, where r_{max} is a value chosen to be more than r_{cut}.

In [92], a parallelization method for canonical MC simulations via the domain decomposition technique has been presented where each domain is further divided into three subdomains. The size of the middle subdomain is chosen as large as possible to minimize interprocess communications due to frequent crossings of particles between adjacent domains, or when updating the two outer subdomains. However, such large domains are not suitable for the GPU because for short range cut off systems, the larger the domain is, the more wasted calculations and the more wasted reserved space.

A cell list implementation for the MD simulations has been implemented in [121, 112, 24]. In such simulations, all particles in the simulation are randomly displaced at the same time, which makes the cell list implementation more beneficial and shows more speedup due to the intensive computation overhead of simulation and the reuse of the cell structure.

With the neighbor list structure, each particle stores IDs of that particle's neighbor particles. An update to the neighbor list should follow a change in a particle's location since particle positions have been changed and are no longer accurate. A work around the complexity of generating a neighbor list is accomplished by instead storing particle positions in a cell data structure and using that directly in the pair force computation. Until now, no available GPU-based Monte Carlo engine has been developed for standard thermodynamic ensemble simulations of Lennard-Jones particles to this scale or uses the cell list to accomplish the simulation.

CHAPTER 3 Porting Canonical Ensemble to the GPU

We present a novel optimized GPU-based Monte Carlo simulation for the canonical ensemble using the CUDA framework. Our system opens the door for simulations of systems with hundreds of thousands of particles and hundreds of millions of simulation steps on a commodity desktop computer loaded with a commodity GPU. In addition, each thread in our model is mapped to one or more unique particle pairs for calculating virial (used to calculate pressure) and energy. Finally, our study shows that a faster CPU does not have a significant impact on the performance of the parallel algorithm while a faster GPU makes a noticeable performance difference for the same platform. To illustrate, running the simulation on a relatively slow CPU gave a speedup of 20.3 times on a Core 2 Duo CPU, compared to 12.33 times speedup on an average Core i5 CPU using the same GeForce GTX 480 card. The parallel execution time was almost the same on both platforms; the difference in speedup is due almost entirely to the relative running time of the sequential algorithm on each platform. Moreover, we research the use of cell list structures for a very large systems.

3.1 Markov Chain Monte Carlo Simulations

A Markov chain method has the property that step $N + 1$ depends on the results collected in step N. Monte Carlo simulations use random sampling to solve computational problems. We are interested in the Monte Carlo simulation of chemical systems that use the Monte Carlo method to evolve system configurations via probabilistic acceptance rules derived from statistical mechanics. The methods that allow Monte Carlo simulation for atomistic systems are described as follows.

3.1.1 Metropolis Method and Thermodynamic Ensembles

While there are many approaches to applying Monte Carlo methods to molecular systems, the most popular one is called the **Metropolis method** [79].

In general, the Metropolis Monte Carlo method [105] is a computational approach to generate a set of C configurations of the system. The iterations are independent of each other, so the probability that the system

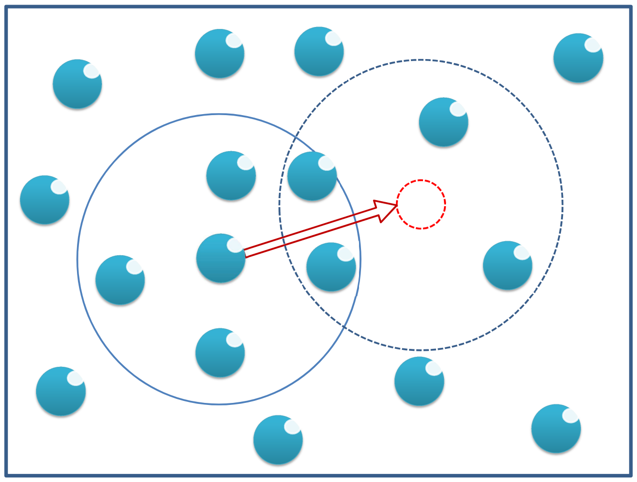


Figure 3.1: A particle displacement attempt in Metropolis Monte Carlo method.

reverts to its previous state is as likely as selecting any other state.

An *ensemble* (also statistical ensemble or thermodynamic ensemble) is an idealization consisting of a large number of mental copies of a system, considered all at once, each of which represents a possible state that the real system might be in [39]. One of the most common ensembles used in the literature is the *canonical ensemble* where the number of particles (*N*), volume (*V*), and temperature (*T*) are fixed. However, the system energy (*E*) and pressure (*P*) are variables. This ensemble is also referred to as the *NVT* ensemble. Using Monte Carlo trials of different configurations, as per the *Boltzmann's Ergodic Hypothesis* [10], this method can give accurate physical information for many systems over a sufficient number of trials.

The acceptance criteria in this case is typically given by first calculating the Boltzmann factor:

$$e^{(-\beta \Delta E)}, \tag{3.1}$$

where ΔE is the change in energy from the previous state to the tested state, β is given by $(1/k_B T)$, k_B is the *Boltzmann constant*, and T is the temperature of the system. The result of this equation is typically compared to a random number in the range $[0, 1)$. If the random number is higher than the Boltzmann factor, the move is accepted. This approach is known as the Boltzmann probability distribution [39].

3.1.2 Lennard-Jones Potential

The Lennard-Jones potential is a frequently used short-range interaction model to simulate interactions between a pair of particles [28]. The potential is given by:

$$U_{LJ} = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right], \tag{3.2}$$

where ϵ is the depth of the potential well, σ is the collision diameter for interacting particles, and r is the distance between interacting particles. As can be observed, the mathematical succinctness of this formula encourages its predominant use in the literature. From an implementation perspective, however, the simulation tends to be computationally intensive even for small systems. Specifically, the computation of interaction forces among molecules in a Lennard-Jones simulation which is given by the equation:

$$F_{LJ} = 24\epsilon \left[2 \left(\frac{\sigma^{12}}{r^{13}} \right) - \left(\frac{\sigma^{6}}{r^{7}} \right) \right]. \tag{3.3}$$

This portion of the simulation is responsible for nearly all of the execution time [70]. The complexity of computing particle interactions is typically reduced by maintaining the total system energy and computing only the change in energy of the system when a particle is displaced. Therefore, each displacement attempt takes $O(N)$ time, where N is the number of particles in the system.

The reader is referred to [39] for the proof of the validity of this method and further chemical details.

3.2 Structure of the Code

In order to gain a better understanding of my implementation, a high-level view of the serial algorithm is presented, see Algorithm 1. In this algorithm, an initial system energy is calculated, then a randomly chosen particle is moved to a random location. Finally, the acceptance rule is calculated as a function of the change in energy for that specific particle.

The CUDA architecture has some limitations that affect the system performance. For example, as the kernel cannot write directly to an output device, all system status and move results have to be copied back to the CPU for further processing and for output to files. Since the GPU and CPU do not share a common memory space, memory transfers are required to update the system status on the GPU if the CPU has changed some shared variables and vice versa.

Developing a parallel GPU algorithm is largely domain driven. Our parallel algorithm has the same structure as the serial one due to the serial nature of the Monte Carlo algorithm. However, specific functions have been ported to the GPU. The flowchart in Figure 3.2 illustrates the main operations of the parallel implementation:

1. Generate a sequence of random numbers and move them asynchronously to the GPU along with system configuration parameters such as particle positions, current energy, current virial, number of particles in the system, etc.
2. Repeatedly perform trial move attempts within the main loop. For each trial pick a random particle to move a random distance in a random direction and calculate the difference in energy (ΔE) for the selected particle in the old and new locations. This includes:
    (a) Assign threads to particles
    (b) Calculate partial energy sums from all threads
    (c) Calculate partial energy sums from all blocks
    (d) Assign the result to ΔE
    (e) Calculate the Boltzmann factor
3. Compare a random number to the resulting probability of acceptance calculated from the previous step
4. If the move is accepted, apply the changes to the system and adjust status
5. Periodically, output system status and particle positions to a data file
6. If there are more steps to execute, go to step 2

Figure 3.2 shows the hybrid CPU-GPU system, and illustrates data movement between the host and the device using double line arrows. Moreover, the data flow has been labeled to illustrate the specific data being transferred for that particular step. Eventually, the host has the main loop that the simulation executes, in

Algorithm 1 Serial Canonical Ensemble Monte Carlo Algorithm

```
input: Number of particles and Volume
input: Temperature
input: $\epsilon$, $\sigma$, r$_{cut}$

// Calculate initial energy of the system
for i = 1 to N-2 do
    for j = i+1 to N do
        total_energy += calculate_pairwise_energy(i, j)
    end for
end for
// Main Loop
for  i = 1 to step_number do
    // Randomly select a particle to move
    s = selected_particle $\leftarrow$ rand()
    Old_particle_loc $\leftarrow$ particle_location(s)
    // Randomly move to a new location
    New_particle_loc $\leftarrow$ rand()
    // Calculate the selected particle's energy for the old and new locations
    for k = 1 to particles, k ! = s do
        old_energy_contrib += calculate_pairwise_energy(Old_particle_loc, k)
        new_energy_contrib += calculate_pairwise_energy(New_particle_loc, k)
    end for
    deltaE = new_energy_contrib - old_energy_contrib
    calculate_acceptance_rule()
    if  accepted  then
        total_energy += deltaE
        current_config $\leftarrow$ new_config
        update_system_status()
    else
        //Leave current system state
    end if
    //Update the rate of accepted moves
    // Solve if the system in equilibrium
    // Periodically write system status to disk
end for
```

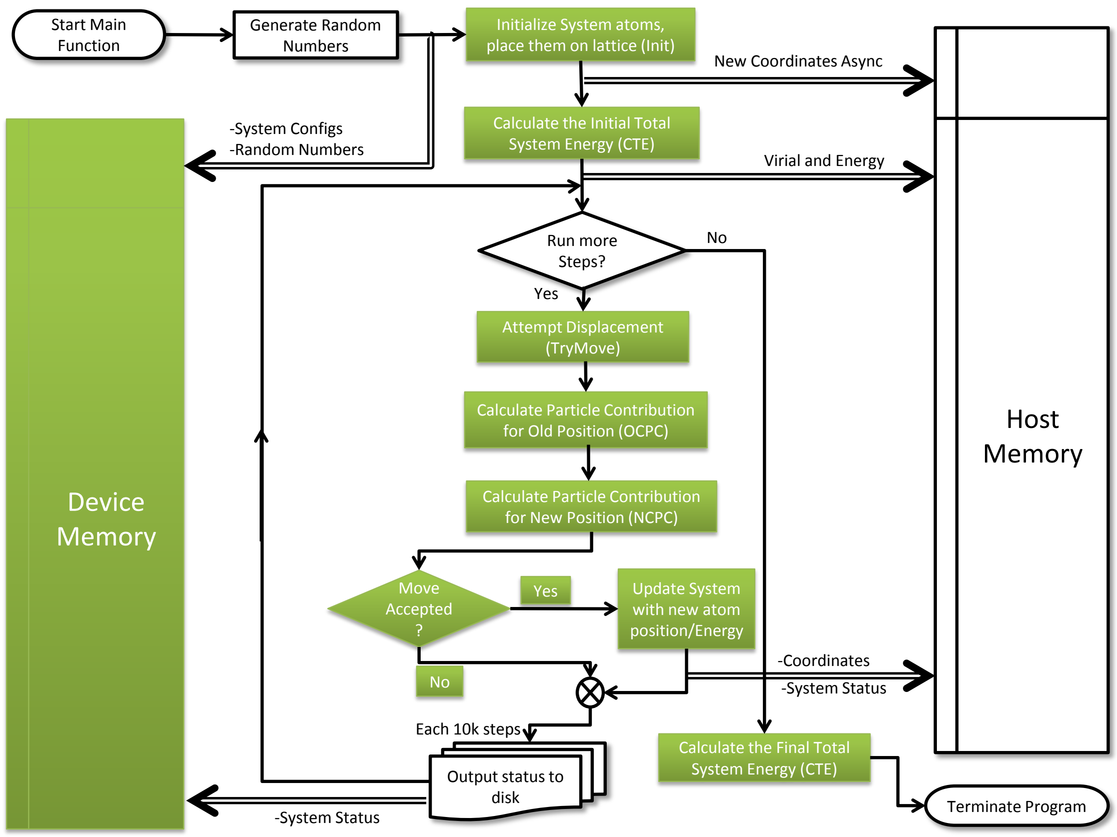

Start Main Function
Generate Random Numbers
Initialize System atoms, place them on lattice (Init)
New Coordinates Async
Calculate the Initial Total System Energy (CTE)
-System Configs
-Random Numbers
Virial and Energy
Run more Steps?
No
Yes
Attempt Displacement (TryMove)
Calculate Particle Contribution for Old Position (OCPC)
Calculate Particle Contribution for New Position (NCPC)
Move Accepted ?
Yes
Update System with new atom position/Energy
No
-Coordinates
-System Status
Each 10k steps
Output status to disk
Calculate the Final Total System Energy (CTE)
Terminate Program
-System Status
Device Memory
Host Memory

Figure 3.2: Monte Carlo Simulation for the canonical Ensemble method flowchart. Kernel functions are in filled shapes.

addition to the I/O necessary to output system status to disk. The kernel function *TryMove*() is responsible for handling the particle displacement attempt. The system energy and pressure are stored from the previous state and will be used to calculate the acceptance criteria for each displacement attempt. Moving this function to the device led to significantly better overall system performance, because the pairwise interactions can be calculated in parallel.

The *Calculate_Total_Energy()* function is another important function in this simulation. This function calculates the current system energy resulting from each interacting pair of particles, which requires $O(N^2)$ computations. Many optimizations have been applied to this function. The main focus was to balance the workload across the threads and hide the global memory latency. This function is executed only twice, at the beginning of the simulation to find the initial system energy and at the end of simulation for verification purposes.

3.3 Optimizing the canonical ensemble method for the GPU

In this section, we shall consider specific strategies implemented to optimize the canonical ensemble code. This list mentions a number of significant optimizations that have boosted the performance of this MC simulation. Since a parallel algorithm cannot be generalized to all problem domains, we have focused on the optimizations that enhanced the overall performance of this particular class of problems.

3.3.1 The block size effect

The number of threads per block is limited by resources that the device can allocate to each block. For devices of compute capability 1.x the maximum number of threads per block is 512 threads, and 1024 threads per block for devices of compute capability 2.x. One may think to load the GPU with the minimum number of threads per block so that less threads share resources per block to increase the performance. However, this is not the case. The main drawbacks to using smaller block sizes are the reduced sharing of data among threads and the limit on the maximum number of blocks that can run on an SM.

Threads in one block can share data through fast shared memory, blocks on the other hand can share data only through device global memory, which is much slower than shared memory. Another drawback for small block sizes is the need for synchronization mechanisms. While threads in the same block can synchronize

Table 3.1: Thread hierarchy and properties.

Coarse	Size	Associated Resources	
		Memory Scope	Processing
Thread	–	registers, local memory	1 core
Warp	32 threads	registers, local memory	1 SM
Block	512/1024 threads for 1.x & 2.x compute cap.	shared memory, L1, L2 cache	1 SM
Grid	65,536 per dim 64 on z-dim	global, constant, texture	device scope

execution through lightweight CUDA statements such as `__syncthreads()`, threads in different blocks need other techniques to accomplish synchronization such as the technique mentioned in section 3.3.7. In this study, several block sizes are examined and we have reported the performance measurements for each.

3.3.2 The use of pinned memory

Pinned memory enables *asynchronous* memory copies (allowing for overlap with both CPU and GPU execution) as well as improving PCIe throughput. An example of using pinned memory in the NVT Monte Carlo simulation is the storage of pre-generated random numbers. Random numbers are needed for each step of the algorithm; we used the Mersenne Twister [77] random number generator on the CPU to produce a sequence of random numbers that are copied periodically and asynchronously from the CPU to the GPU. In addition, the system takes advantage of high throughput pinned memory when periodically transferring particle coordinates modified by the GPU to the CPU for checkpointing.

3.3.3 The use of different GPU memory types

Shared memory can be accessed by any thread in that particular block. Other blocks, on the other hand, have no access to this memory. Table 3.1 shows the GPU structures that can access shared memory. One of the strengths of the GPU is the existence of shared memory and cache. However, the amount of this high throughput memory is limited to a maximum of 48K per Streaming Multiprocessor (SM). Table 3.3 lists the different memory specifications for the cards used in this study.

Allocating shared memory in CUDA can be performed by using the `__shared__` qualifier. Access to this on-chip memory is more than an order of magnitude faster than local or global memory. The size of the block is a key factor for the allocated shared memory and global memory. If a block has many threads using shared memory, better performance can be expected if there is enough shared memory. On the other hand, if a block has fewer threads, there will be more blocks in the grid that need to be synchronized using global memory transactions. This last scenario will add extra memory latency overhead to the simulation.

Our implementation uses shared memory to aggregate partial sums among blocks and keeps track of common variables that will be used by all threads in a block. However, §3.3.8 shows an unavoidable use of global memory to synchronize blocks in a grid. Another type of memory that can enhance the overall system performance is constant memory. Our application uses constant memory to store fixed system parameters that are used throughout the simulation to avoid expensive global memory CPU-GPU communication.

3.3.4 Memory coalescing for fetching particle positions

Combined memory accesses can have a dramatic effect on the throughput of the program. For instance, if the threads are not accessing adjacent memory locations within a transaction, bandwidth is needlessly wasted. On the other hand, fewer memory transactions are required when accessing contiguous memory locations, which increases the overall performance of the system. Whenever possible, our implementation uses a sequence of threads to access neighboring locations in global memory in order to achieve memory coalescing. Moreover, when calculating the total energy and the total virial contribution to the pressure of the system, each block of threads will typically be responsible for finding more than one pairwise particle sum, where particles are stored in global memory. The access pattern to global memory in this case maps consecutive thread IDs to consecutive global memory locations.

3.3.5 Loop unrolling technique in finding total energy

Loop unrolling is a potential optimization technique where the modified loop performs more than a single iteration in each pass of the loop. This process is typically performed automatically by an optimizing

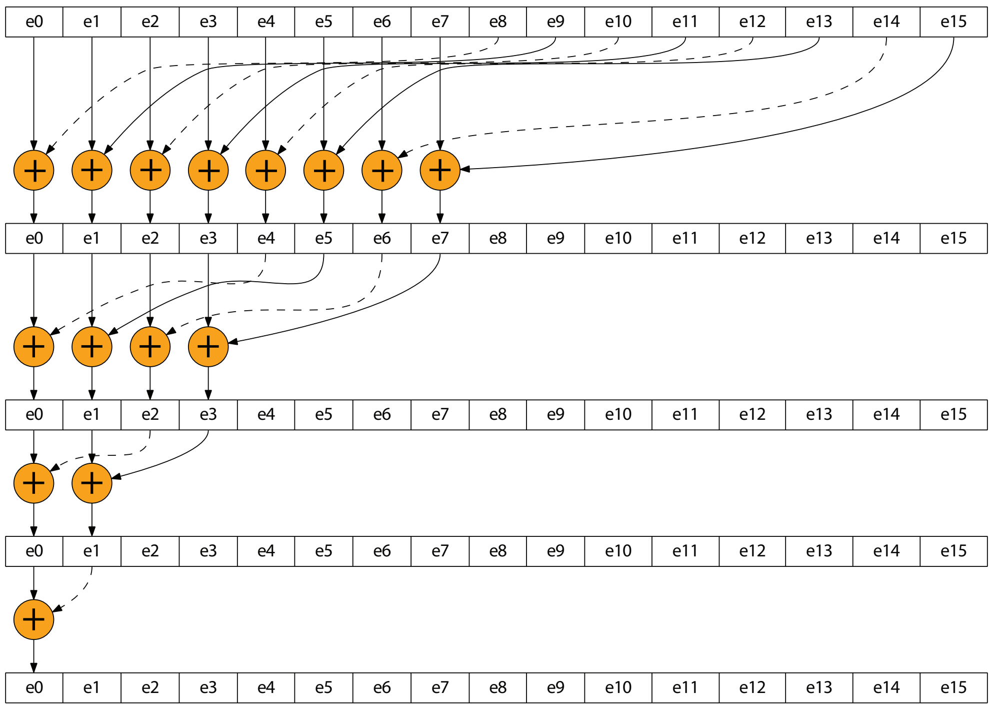


Figure 3.3: Calculating the partial sum for values in shared memory that use adjacent memory locations. Each circle represents work done by a thread.

compiler, but in certain circumstances can be undertaken by the programmer. These specific circumstances arise if the opportunity to break a dependency chain in the loop presents itself.

In the context of our program, loop unrolling has been applied to perform all of the summation steps within a single warp in parallel[1]. This technique is similar to, but more general than, the one mentioned in [48], where slightly more optimizations have been made. For example, our implementation handles cases where there is not an exact multiple of two elements to calculate, there are more threads than particles in the system, or there are more blocks than threads. Even so, we use fewer global memory transactions than [48], and rely more on caching, which was not available for their implementation. Algorithm 2 shows the summation process for one block. Each thread is responsible for finding a partial sum. To accomplish this, only $(N/2)$ threads are needed. The constant *tid* represents the thread ID in the block on the *x*-axis that is assigned at execution time.

Note that in the switch-case statements, all subsequent cases are executed until a break statement is encountered. For example, when the *offset* value is eight, then the case for eight, four, and two will be

[1]Threads in the same warp do not need to be synchronized.

Algorithm 2 Partial sum showing loop unrolling

```
// offset equals largest power of two less than the block size
// Check if we have more blocks than threads
// Start summing the values in cache memory
i ← offset
while i > 32 do
    if tid < i then
        cachedEnergy[tid] ← cachedEnergy[tid + i] + cachedEnergy[tid]
    end if
    __syncthreads()
    i ← i / 2
end while
// Find the sum of the first 64 values
if tid < 32 then
    offset ← min(offset, 64)
    switch (offset) do
      case 64:  cachedEnergy[tid] ← cachedEnergy[tid + 32] +cachedEnergy[tid]
      case 32:  cachedEnergy[tid] ← cachedEnergy[tid + 16] +cachedEnergy[tid]
      case 16:  cachedEnergy[tid] ← cachedEnergy[tid + 8] +cachedEnergy[tid]
      case 8:   cachedEnergy[tid] ← cachedEnergy[tid + 4] +cachedEnergy[tid]
      case 4:   cachedEnergy[tid] ← cachedEnergy[tid + 2] +cachedEnergy[tid]
      case 2:   cachedEnergy[tid] ← cachedEnergy[tid + 1] +cachedEnergy[tid]
    end switch
end if
```

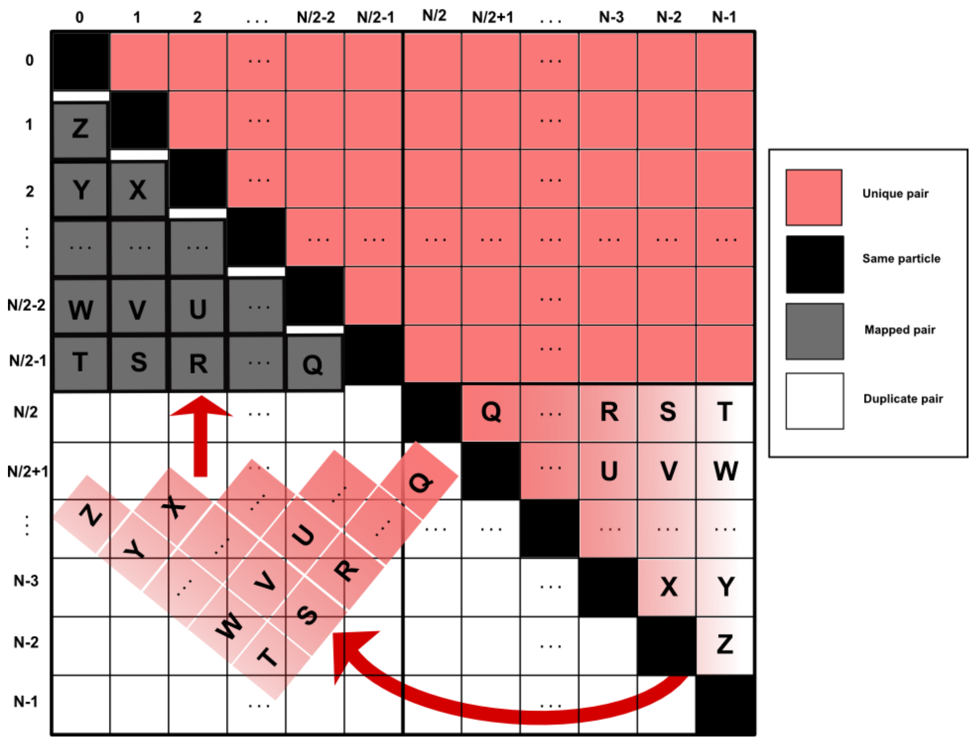


Figure 3.4: Mapping algorithm for work load balancing across threads.

executed. This technique is illustrated in Figure 3.3. For blocks with enough threads, applying this method to the kernel code enhances performance significantly.

3.3.6 Load balancing among threads and contributing particles

Only unique particle pairs should be considered in the total energy calculation, which means that for N particles $N(N-1)/2$ unique pairs must be evaluated for potential interaction. An $N \times N$ square matrix can be used to illustrate the pair interactions as seen in Figure 3.4. The unique pair interactions are in the upper triangular or lower triangular matrix; here the upper triangular matrix is chosen. To create a contiguous block of unique interactions, the lower right unique interactions of the square can be mapped into the upper left quadrant so that the unique interactions will be contiguous rows. This mapping produces $N/2$ rows of contiguous unique pair interactions. At this point it is easy to balance the load among threads. For instance, one thread could be assigned to only one row, or one row could be divided among more than one thread.

As N grows, one row could be divided among multiple threads. Finer grained threads can access locations in one row using a block cyclic distribution to take advantage of contiguous data locations. For example, if four threads are assigned to each row, each thread will process locations using an offset of four. With this technique, threads will access adjacent locations on each pass, which reduces the memory latency. Figure 3.4 provides further illustration of the mapping algorithm.

3.3.7 Atomic operations on global memory transactions

This operation is useful to avoid a race condition for cases where multiple threads are competing to modify a particular memory location. For instance, atomic operations make it possible to synchronize blocks in a grid, since blocks can share only global memory. This is discussed further in §3.3.8. On the other hand, atomic operations can lower performance since they serialize accesses to global memory, add extra instruction processing, and require busy waiting. However, Fermi cards offer a more efficient implementation of atomic operations than older GPUs do.

3.3.8 Block synchronization through global memory and atomic operations

The current structure of the CUDA architecture does not support explicit synchronization between blocks of a grid. So, we adopted a technique that uses atomic operations on global memory to achieve this goal, which is presented in [87]. This method defines a boolean variable that is set to *true* when the last block finishes. After this, the threads in this last block will handle the last piece of work to be performed in the kernel call. Specifically, the threads of the last block finish the work by collecting partial sums found in thread zero of each block. These steps are illustrated in Algorithm 3. This tree-based method is the most efficient parallel technique for finding the sum of a large array and uses many features that are inherent to the GPU.

3.3.9 Numerical optimizations: tricks and tweaks

Several mathematical operations have been optimized to improve the overall performance of the program execution. For example, mathematical functions such as `__fdvidef`, `__log2f`, and `__expf` are natively supported by the GPU hardware and execute in fewer clock cycles. This offers a significant performance advantage for a system with extensive mathematical operations. Other examples include the use of shift left and shift right for multiplying or dividing by two, respectively. Since Monte Carlo simulation does a significant number of repetitive mathematical operations, these optimizations gave the most performance

Algorithm 3 Block synchronization technique

```
// Thread 0 of each block has the sum of all values for that block
if tid = 0 then
    GlobalEnergy[blockId] ← cachedEnergy[0]
    LastBlock ← atomicInc (&BlocksDone, gridDim.x) = gridDim.x - 1
end if
__syncthreads()

// The last block sums the results of all blocks via global memory.
if LastBlock then
    // Move all block values from global memory to shared memory.
    if tid < BlocksPerGrid then
        cacheEnergy[tid] ← GlobalEnergy[tid]
    end if
    // If you have more blocks than threads, reduce the extra values.
    i ← ThreadsPerBlock
    while i < gridDim do
        if tid + i < BlocksPerGrid then
            cachedEnergy[tid] ← GlobalEnergy[tid + i] + cachedEnergy[tid]
        end if
        i ← i + ThreadsPerBlock
    end while
    __syncthreads()
    // The threads in the last block have gathered the results of all the blocks.
    // Use Algorithm 2 to combine the values from all threads to get the total.
end if
```

improvement among all the optimizations tested.

Another optimization technique is to use more efficient mathematical operations. For example, a double-precision division operation such as $[A < (B/2.0)]$ is replaced with an addition operation such as $[(A+A) < B]$, substituting the cost of a division operation with an addition. A second example of mathematical optimization is when calculating the Boltzmann factor in Equation 3.1. Since the denominator is a constant, we instead compute the reciprocal once at the start of the simulation and replace a division with a multiplication.

3.4 Using cell list structure

The energetic decomposition algorithm used here does suffer from conducting extra calculations in finding the Boltzmann Factor at each move attempt. This problem has been solved for MD simulations using Verlet lists [5]. In the Verlet list algorithm, a neighboring list for each particle in the system is maintained. The cost of maintaining this structure is easily hidden in MD algorithm since all particle neighbors move at once in all steps. Moreover, the original algorithm has been modified to delay the update of the neighbor list creation a certain number of steps. However, such an algorithm has not been explored with MC simulations due to the fact that only small systems have been simulated with MC, so the cost of maintaining the neighbor list is higher than the performance gain.

Another structure that has been used with MD simulations to speed up the performance of calculating the pairwise system energy is called the Cell List structure [112, 121, 5]. In this structure, the simulation box is divided into cells of equal size and each cell contains the indexes of the particles inside the borders of that cell, as in Figure 3.5. The process of assigning particles to cells is called *binning*. Since all particles in MD simulation are displaced at the same time, the cell list has to be regenerated correspondingly. To enhance the performance of MD simulations, the cell list structure is used to bin particles in cells, then the Verlet list is generated. In [5], the performance of the MD code using the Verlet list outperforms the performance of that using just the cell list structure.

Although the cell structure eliminates the need for executing extra pairwise energy calculations for those particles outside the cutoff, maintaining the structure is not free of charge. Moreover, due to the lack of atomic operations support in the older versions of GPUs, the use of this structure was not efficient enough. However, the overhead of maintaining the cell list is minimized for the NVT method, since the size of the

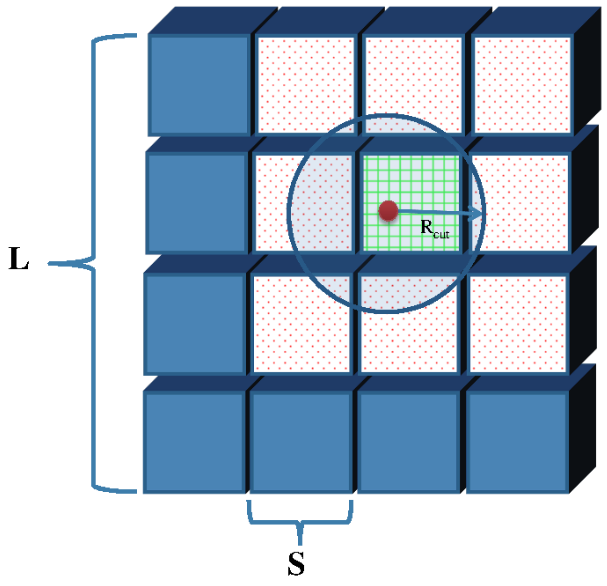


Figure 3.5: The volume $V = L^3$ is decomposed into $T \geq 3$ cells per dimension, with cell dimension of size $S \geq r_{rcut}$.

box is fixed and there is only one particle moving at a time.

In the NVT method, the use of cell list could be summarize in the following operations:

1. Generate the cell list. The cell list is generated at the beginning of the simulation and all particles are binned in their corresponding cells. This operation is done on the device side and not needed on the host side.

2. Generate adjacency list. A list of all adjacent cells on all axes is generated. Each cell calculates its 26 adjacent cell IDs, shown in Figure 3.6. This is also created once at the beginning of the simulation.

3. Parse the cell list. This is done through assigning one thread per particle in a cell. For this purpose, several algorithms have been developed to investigate the optimal number of cells per thread block. The most straightforward algorithm is when assigning one cell per thread block. A total of 27 blocks is created, of size 32 threads each to calculate a particle's contributing energy in the old and the new locations. Moreover, one, three, and nine cells per block have been tested.

4. Maintaining the cell list. There are two operations executed on the cell list to maintain the updated list. First, deleting from a list. If one particle moved far enough to enter another cell, then the particle

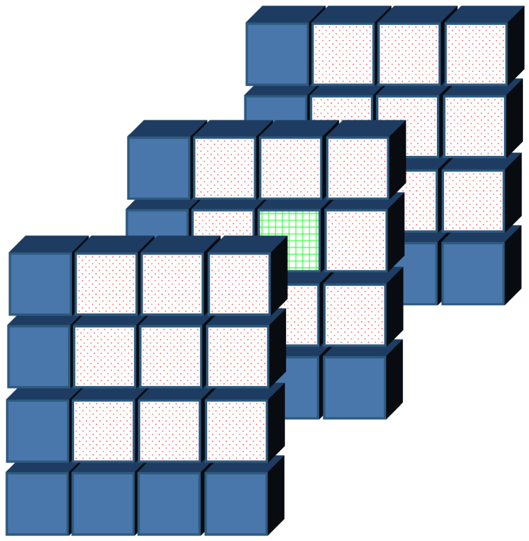


Figure 3.6: Cell with all 26 adjacent cells.

should be removed from its old list. Second, a particle has moved to a new location and should be inserted into a new cell.

To maintain the flow of the text, a more detailed description of cell list structures can be found in § 4.2.2.

3.5 Results and Discussion

While we are using the CUDA architecture as an extension to the C language to implement the parallel algorithm, other Monte Carlo simulation codes are written in Fortran. Therefore, we started by re-implementing the serial algorithm in C/C++. The serial code is statistically equivalent and in close agreement with publicly available canonical ensemble simulation results from the National Institute of Standards and Technology (NIST) [90]. A parallel algorithm was then developed starting from our serial code. Results from the CUDA and single-core CPU implementations match exactly when the same random seed is used.

The comparison between the serial code presented in this work and the *Towhee* serial code using the same configurations, shown in Table 3.2, depicts a huge performance improvement of up to 438.3 times faster than the *Towhee* implementation for a relatively small system size. Note that *Towhee*'s slower runtime prevented running experiments for larger system sizes. However, since the serial and parallel codes developed for this study ran in a reasonable amount of time, results for system sizes larger than the ones found in Table 3.2 are

Table 3.2: Average program execution times (in seconds) and speedup over *Towhee*.

N	**Serial**	**Towhee**	**Speedup**
256	6.31	270.2	42.8
461	10.1	908.0	89.9
512	11.58	1118.2	96.56
1024	21.07	2897.2	137.5
2048	40.29	9642.8	239.3
4096	73.34	32150.3	438.3

reported.

The proposed parallel algorithm would not make a fair comparison against *Towhee* for two reasons. First, *Towhee* has additional functionality, which includes electrostatic interactions via Ewald summation, configurational bias methods, and multiple ensembles (isobaric-isothermal, grand canonical and Gibbs ensemble). These are features that are not yet supported by our code, and require additional computational overhead. Second, there is no easy way to ensure that *Towhee* and our parallel code contain the same set of program optimizations. So, it would be difficult to distinguish the speedup due to parallelism from the speedup due to the use of more efficient algorithms.

The recorded elapsed time includes the time to read from the input file, allocate memory, transfer data to the device, and run the massively threaded algorithm for each particle displacement attempt, but not the time to calculate the final system state, which is a validation step and not part of the simulation. Most test runs are for 2^n particles, and corresponding volumes of 2^{n+1} where $8 \leq n \leq 18$. The average speedup is for the CPU and GPU running a million simulation steps[2], where each step is a move attempt. For all runs, we used (-O3) and (-m64) flags passed to the gcc compiler. Furthermore, performance is measured in terms of the speedup, which is the ratio between the serial and parallel end-to-end application execution times. However, for statistical validation all experiments have been run five times and the average of these runs is used. All of the five tests run times show very close agreement. Precisely, the difference between this average of five runs and any single run was always less than 3%. In fact, out of sixteen hundred runs, only nine deviated

[2]Although hundreds of millions of simulation steps are required to obtain scientifically accurate simulation results, one million steps is sufficient to show the relative speedup of the GPU code.

Table 3.3: Specifications for the three graphic cards used to run reported experiments.

	GeForce GTX 460	**GeForce GTX 560**	**GeForce GTX 480**
Number of cores	336	336	480
Streaming Multiprocessors	7	7	15
Max Shared Mem. per SM	48 KB	48 KB	48 KB
Global Mem. (GDDR 5)	1 GB	1 GB	1536 MB
Processor clock (MHz)	1300	1700	1401
Max block size	1024	1024	1024
Mem. Bandwidth (GB/sec)	108.8	128	177.4
Compute Capability	2.1	2.1	2.0

from the average for that configuration by more than 1%.

Three different graphics cards have been used to run the experiments. The specifications of all three cards can be found in Table 3.3. Although the GeForce GTX 480 is an older model than the GeForce GTX 560, the former has more global memory and higher memory bandwidth, which enhances the performance of this application domain. Moreover, there are twice as many multiprocessors in the GeForce GTX 480, which allows for scheduling double the number of blocks at the same time compared to the other two cards. While we have access to a high end NVIDIA® Tesla® card, we could not report results obtained with this card due to the lack of a high end CPU such as the Intel® Xeon® processor. The results achieved in this work were obtained with the commodity desktop processors described in Table 3.4.

Table 3.5 shows different block sizes and their effect on the overall simulation speedup compared to the single-core serial code. Performance-wise it can be noted that:

1. When the number of threads per block is small, the need for more global memory accesses for synchronization rises. This is most pronounced when there are only 32 threads per block. The worst performance for this case was when the system size is 131,072 particles, and has 4,096 blocks.

Table 3.4: Desktop computers used for the experiments.

GPU	CPU	RAM	OS
GTX 480	Intel Core i5-2500K	8 GB	CentOS 6.2
GTX 560	Intel Core i5-2500K	8 GB	CentOS 6.2
GTX 460	AMD Phenom II	6 GB	Ubuntu 11.04
GTX 480	Intel Core 2 Duo	2 GB	CentOS 6.2

2. 64 threads per block offers the best performance when there are less than 8,192 particles in the system. With this block size, load balancing of shared and global memory usage is achieved for the reduction operation. In addition, exactly two warps are scheduled for each block. This is consistent with prior research on optimal block size given in [84]. However, end-to-end execution time is the worst when the system size is larger than 8,192 particles as seen in Table 3.5. This is evidence that the GPU's performance depends on the problem size and specifications, and not on a general rule.

3. Systems consisting of at least 8,192 particles, but less than 32,768 particles achieve the best performance with 128 or 192 threads per block. Resource sharing is critical for large systems, and less resources are allocated when larger blocks are used.

4. A further performance improvement is observed in systems larger than 65,536 particles when assigning 128 threads per block. The performance improvement is nearly the same as with 192 threads running in a block, which is a multiple of 64, too. This is due to the fair share of resources and the balance in using shared versus global memory.

These results show that selecting the optimal block size is not trivial. For example, in our case 128 particles per block is the recommended block size for very large systems, but does not perform best for smaller systems.

Looking at Table 3.5, also plotted in Figure 3.7, a detailed comparison between different GPUs running on different platforms is observed. We obtain significant performance improvement for some GPUs over others running the same code on the same platform. For instance, the GeForce GTX 480 obtains up to 12.33 times speedup compared with a maximum of 7.35 times speedup for the GTX 560 running on the same desktop. This is because of the extra core count and memory capacity of the GTX 480 over the GTX

560. Also, from Figure 3.7 we notice the same pattern of speedup for all runs on all systems. Speedup is increasing gradually with the system size and shows the best performance for the largest systems.

Figure 3.8 plots the execution times in seconds for the serial code against *Towhee* [76] on an Intel® Core™ i5 and Figure 3.9 compares the execution time of the serial algorithm with the parallel algorithm running on the GeForce® GTX 480 GPU. As the system size increases, the execution time for both algorithms increases. However, the execution time of the serial algorithm grows much faster than the parallel version. Note that the break even point where the GPU code starts to overcome the added overhead and shows better performance than the serial code is when the system has more than 512 particles, as shown in Table 3.6. Memory transfers and parallel function invocation are the main causes of this overhead. The speedup ratio seen in Figure 3.6 shows rapid improvement as the system size grows and we expect more speedup for larger systems.

The parallel algorithm has been tested with the same GPU on different machines. Figure 3.10 shows that the speed of the CPU (host) has a negligible impact on the execution time of the parallel algorithm. It is clear that fast CPUs do not provide significant speedup for the parallel code presented here. This was true for all problem sizes we tested. From this, we can conclude that parallel algorithm is executing almost entirely on the GPU and keeping the overhead of executing on the CPU to a minimum.

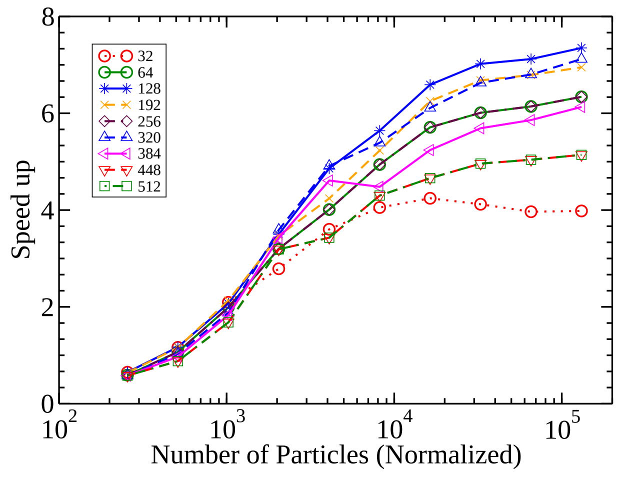
It is often more efficient to perform a small amount of calculations on the CPU while using the GPU to perform the inherently parallel parts of the computation, particularly since these computations can then be overlapped. The reason behind getting worse performance with the CUDA code on small problem sizes is that the overhead of the kernel calls and the memory transfers exceeds the amount of parallelism we can extract from a small problem size. For instance, in small systems with less than 256 particles, the overhead of parallel execution is more than the gain of running the code on the GPU, and for systems of size 512 particles and more, the problem becomes large enough to leverage the GPU parallelism and overcome the overhead of CPU-GPU coordination and data movement and hide memory latency.

Domain decomposition techniques for molecular systems are candidates to enhance the performance by eliminating extra out of range calculations. However, the overhead of maintaining a data structure of neighboring particles for such low computation intensive applications wasn't promising before. Now, with the possibility of simulating very large systems, a neighbor list algorithm [39] could show decent speedup for systems with ten thousand particles or more.

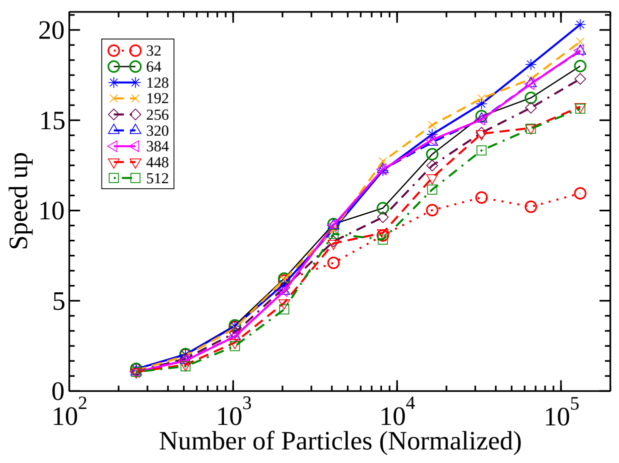
In Table 3.7, the execution time of the CUDA implementation with different variations of cell lists are

Table 3.5: Large vs. small block size and system performance. Numbers shown are speedup.

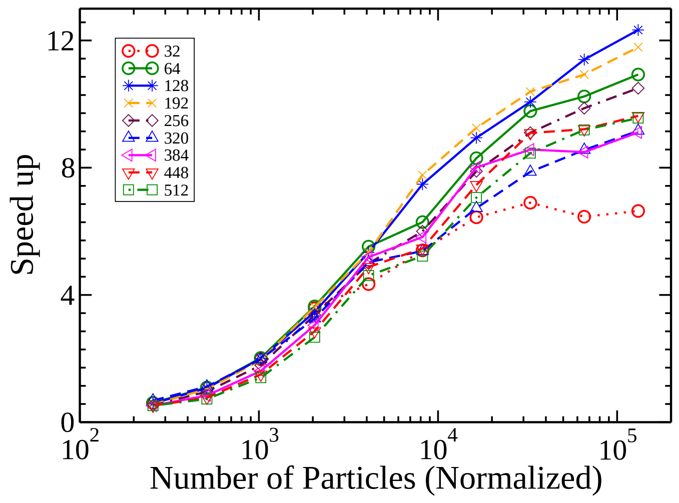
			Number of Particles									
			256	**512**	**1024**	**2048**	**4096**	**8192**	**16384**	**32768**	**65536**	**131072**
GTX 560 + i5	**Block Size**	**32**	0.65	1.16	2.09	2.79	3.60	4.05	4.24	4.12	3.97	3.98
		64	0.59	1.07	1.99	3.20	4.01	4.94	5.71	6.01	6.14	6.34
		128	0.65	1.17	2.07	3.50	4.86	5.64	6.59	7.02	7.12	7.35
		192	0.64	1.16	2.13	3.47	4.24	5.23	6.25	6.68	6.79	6.95
		256	0.59	1.07	1.99	3.20	4.01	4.94	5.71	6.01	6.14	6.34
		320	0.58	1.03	1.88	3.59	4.91	5.39	6.11	6.63	6.80	7.12
		448	0.58	0.88	1.68	3.19	3.43	4.30	4.66	4.96	5.04	5.14
		512	0.58	0.82	1.57	2.99	3.38	4.19	5.00	4.96	5.03	5.22
GTX 480 + i5	**Block Size**	**32**	0.60	1.08	1.98	3.58	4.34	5.40	6.44	6.90	6.46	6.64
		64	0.61	1.09	2.02	3.64	5.52	6.29	8.30	9.78	10.24	10.93
		128	0.59	1.07	2.00	3.47	5.34	7.48	8.94	10.07	11.40	12.33
		192	0.73	1.25	1.96	3.54	4.58	5.53	7.45	8.74	9.47	10.00
		256	0.68	1.16	2.07	3.21	4.67	5.64	7.21	8.26	8.89	9.31
		320	0.68	1.12	1.98	3.26	5.03	5.38	6.73	7.87	8.56	9.16
		448	0.68	0.98	1.79	3.27	3.47	4.29	5.13	5.86	6.30	6.58
		512	0.68	0.92	1.67	3.06	3.40	4.16	5.55	5.86	6.29	6.70
GTX 480+C2D	**Block Size**	**32**	1.21	2.02	3.55	6.12	7.10	8.62	10.02	10.72	10.21	10.95
		64	1.23	2.05	3.64	6.23	9.25	10.13	13.12	15.23	16.24	18.00
		128	1.22	2.02	3.59	5.93	8.95	12.21	14.22	15.91	18.09	20.30
		192	1.15	1.95	3.49	6.18	8.93	12.72	14.73	16.21	17.29	19.35
		256	1.07	1.80	3.22	5.73	8.29	9.63	12.49	14.31	15.69	17.29
		320	1.06	1.69	3.03	5.53	9.17	12.27	13.78	15.07	17.03	18.83
		448	1.06	1.46	2.69	4.88	8.18	8.76	11.81	14.25	14.57	15.75
		512	1.06	1.37	2.49	4.52	7.71	8.39	11.16	13.33	14.53	15.64
GTX 460 + Ph II	**Block Size**	**32**	0.74	1.26	2.17	2.79	3.56	3.94	4.50	4.75	4.91	5.06
		64	0.75	1.26	2.17	3.70	4.37	5.28	6.75	7.46	7.91	8.29
		128	0.59	1.07	2.00	3.47	5.34	7.48	8.94	10.07	11.40	12.33
		192	0.56	1.04	1.95	3.62	5.34	7.77	9.25	10.39	10.93	11.79
		256	0.53	0.96	1.79	3.37	4.99	6.00	7.89	9.10	9.87	10.50
		320	0.52	0.90	1.69	3.25	5.49	7.55	8.72	9.56	10.72	11.36
		448	0.52	0.77	1.50	2.86	4.88	5.46	7.46	9.09	9.21	9.63
		512	0.52	0.73	1.40	2.67	4.61	5.22	7.06	8.46	9.19	9.56



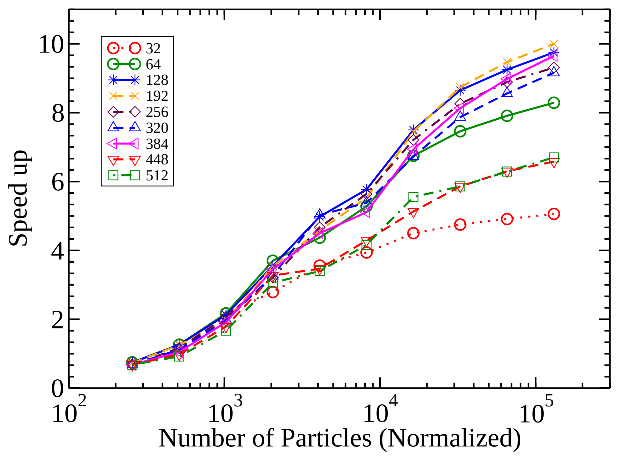
(a) Speedup on i5 CPU and GF 560



(b) Speedup on Core 2 Duo CPU and GF 480



(c) Speedup on i5 CPU and GF 480



(d) Speedup on Phenom II CPU and GF 460

Figure 3.7: Plots of speedup for different block sizes on different platforms.

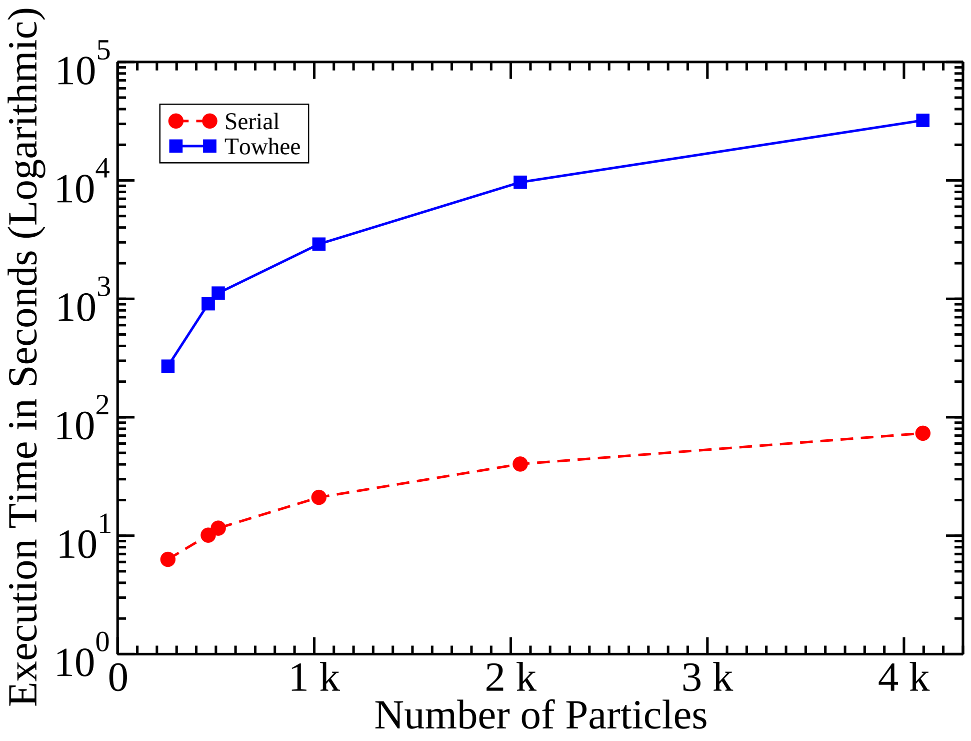


Figure 3.8: Developed serial code vs. *Towhee* elapsed times (logarithmic normalization).

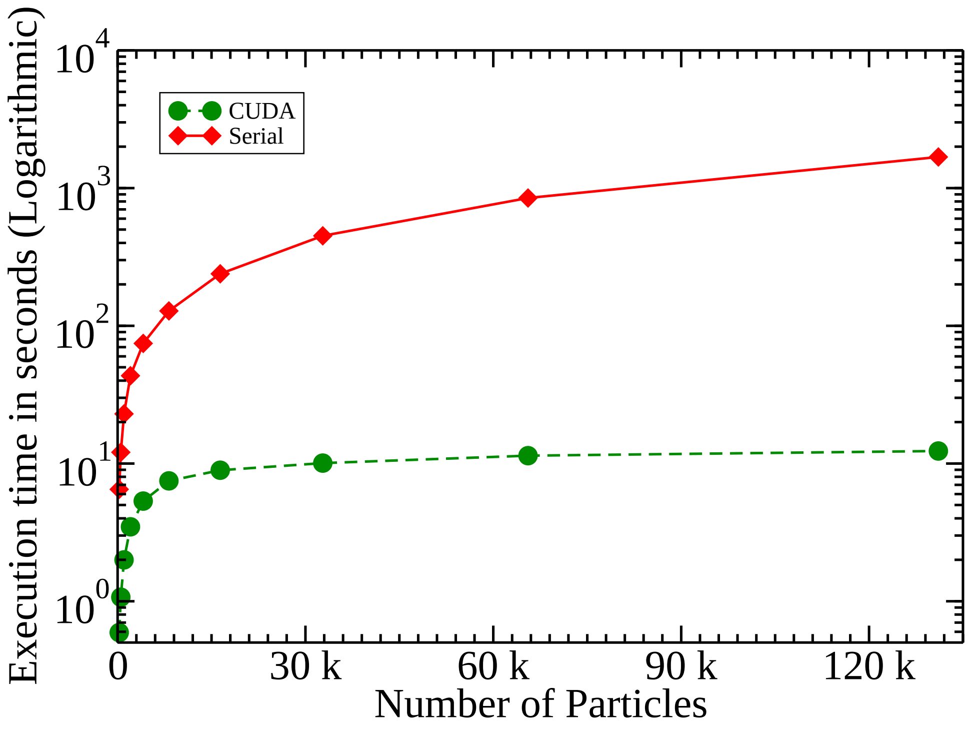


Figure 3.9: Serial vs. CUDA execution times for MC simulation on i5 and GeForce 480.

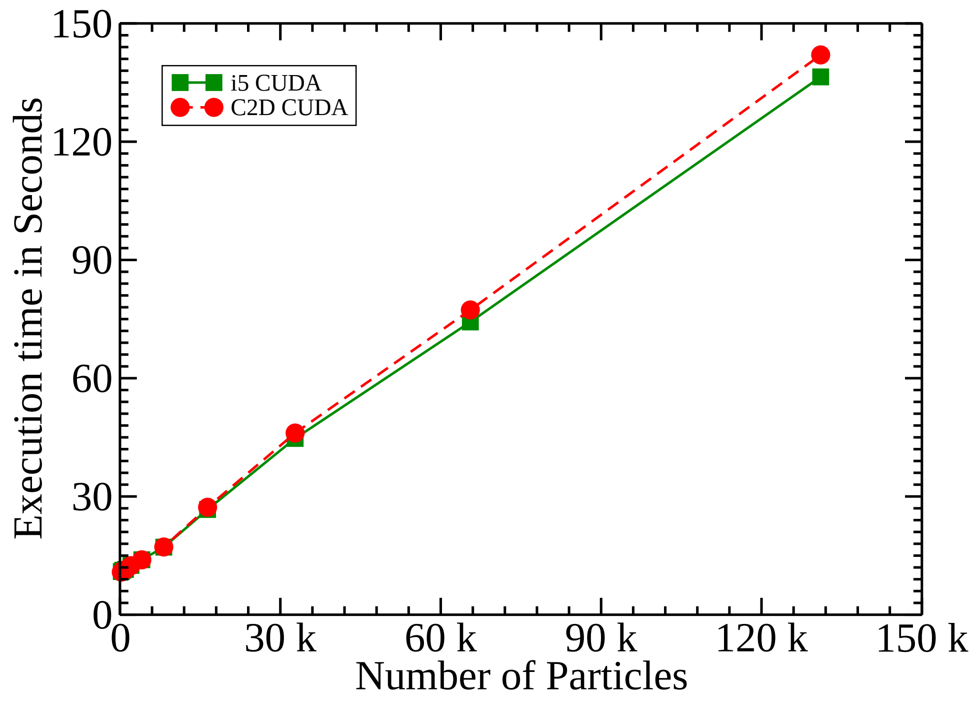


Figure 3.10: Execution times on two different platforms with GTX 480.

Table 3.6: Average program execution times (in seconds) and speedup of CUDA over serial code for a million steps on i5 and GTX 480.

N	**Serial**	**CUDA(128)**	**Speedup**
256	6.5	11.0	0.6
512	12.1	11.3	1.1
1024	23.0	11.5	2.0
2048	43.0	12.5	3.5
4096	74.5	14.0	5.2
8192	128.2	17.1	7.5
16384	238.8	26.7	9.0
32768	450.6	44.7	10.1
65536	847.0	74.3	11.4
131072	1681.9	136.4	12.3
262144	3659.8	243.7	15.0

Table 3.7: Average program execution times (in seconds) for a million steps with different cell list models.

N	**CUDA**	**(27x1)**	**(9x3)**	**(3x9)**	**(1x27)**
4096	14.0	17.1	17.6	18.8	23.6
8192	17.1	17.6	17.1	18.8	23.4
16384	26.7	17.1	17.1	18.6	23.4
32768	44.7	17.7	17.2	18.6	23.2
65536	74.3	17.4	17.6	19.2	23.5
131072	136.4	19.0	19.0	20.4	25.3
262144	243.7	24.4	22.4	23.6	31.0

Table 3.8: Speedup for a million steps with different cell list models over the code without cell list.

N	**(27x1)**	**(9x3)**	**(3x9)**	**(1x27)**
4096	0.8	0.8	0.7	0.6
8192	1.0	1.0	0.9	0.7
16384	1.6	1.6	1.4	1.1
32768	2.5	2.6	2.4	1.9
65536	4.3	4.2	3.9	3.2
131072	7.2	7.2	6.7	5.4
262144	10.0	10.9	10.3	7.9

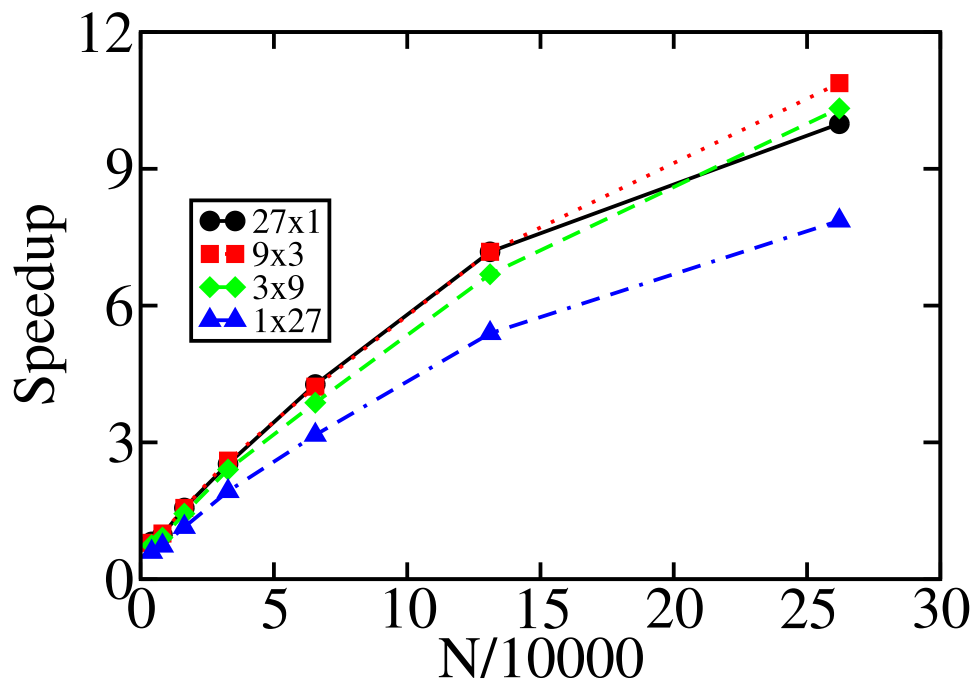


Figure 3.11: The effect of different cell list implementations on speedup against the CUDA implementation without cell list.

compared against that of the CUDA implementation without cell list. In the cell list implementation, the simulation is divided into equal size cells, which means the amount of work executed within the cell is the same for any system size. However, the execution time shows a relatively slight increase in execution time for systems larger than 131,072 particles because of the extra time taken by memory transfers.

Different cell list algorithms manage GPU resources in a slightly different way. Four such algorithm's performance results can be seen in Table 3.8 and Figure 3.11. We note that the best results are achieve when one thread is assigned to three particles in three cells[3] referred to by (9x3) where 9 is the number of blocks needed to process the 27 cells.

[3]Figure 4.1 depicts the cell list algorithms mentioned here.

CHAPTER 4 Grand Canonical Ensemble: One Simulation Box and a Reservoir

The main goal of Monte Carlo and Molecular Dynamics simulations is to compute equilibrium properties of classical many-body systems or to estimate the average properties of systems with a very large number of accessible states. However, MC methods make it more feasible to simulate open systems that MD cannot simulate, because the latter algorithm doesn't support open systems, systems with an addition or deletion of particles [39].

We present a high optimized GPU algorithm for MC simulation using the grand canonical method. For verifying accuracy and as a comparison base, the study started by implementing an optimized serial code that runs on a single CPU core. This code is being used to verify the accuracy of the CUDA code thermodynamic results, for performance measurement, and as a second option for running the code on machines without a GPU. One may argue that developing a code that runs on multicore processors may be an alternative to the code running on manycore devices. Although this claim may be true, investing the time in developing a multicore code that runs on up to four or more cores will not outperform a code that is development on a GPU with tens or hundreds of cores. It is wise to invest the development time in writing code that will definitely run faster and scales for very large problems.

Based on the statistical accuracy of our serial implementation, the serial algorithm has been rewritten to utilize the GPU. The main contribution of this work is a performance enhancement of the CUDA code that makes it possible to run more than quarter of a million particles for tens of millions of simulation steps in the Monte Carlo algorithm in a reasonable amount of time. Instead of waiting days to generate simulation results from huge systems, the user will get her simulation results before finishing lunch. Furthermore, according to feedback from the community, people are interested in conducting more simulations with a relatively small systems, as small as ten thousand particles. This is another strength point to this work, even for small systems, the CUDA code with cell list shows good speedup.

4.1 MC Simulation for the Grand Canonical

The Grand Canonical method (or μ, V, T) is one of the statistical ensembles that are used to represent a possible distribution state in which a simulated thermodynamic system can be in real experiments. The grand canonical ensemble extends the canonical ensemble by defining the values of the temperature (T), volume (V), and the chemical potential (μ) as constants [39]. Particles can interact with each other only when they exist inside the simulated system and when they are within a cutoff radius, r_{cut}, of each other. A reservoir is connected to the simulated box, 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.

This method can be applied to problems such as:

1. Simulate adsorption isotherms. While it is essential to have a detailed knowledge of the behavior of the adsorbed molecules, this type of information is very difficult to obtain experimentally; simulation is the alternative.

2. Could be used in numerical simulations to accurately predict properties of materials and their guest-adsorption characteristics.

3. To determine the equation of state of the Lennard-Jones fluid. One could impose temperature and chemical potential and calculate the density and pressure.

We study systems of particles interacting via the Lennard-Jones potential by calculating the configurational energy of pair interaction, given by:

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \tag{4.1}$$

where *r* is the distance between two interacting particles, ϵ indicates the depth of the potential well, and σ is the collision diameter for the two interacting particles. Note that calculations of the Lennard-Jones potential are significantly more complex than the Ising or hard sphere models, since you have to calculate the interactions between all particles within a certain cutoff radius.

Particles in this ensemble are moving inside the simulation box or between the box and the resevoir.

The criterion for accepting the move is based on the weighting of the Boltzmann factor, Equation 3.1. The acceptance or rejection for the types of particle moves are given by the Metropolis acceptance criterion [80, 8]:

Particle Displacement A random particle is attempting to move randomly within the simulation box. The move is accepted with a probability:

$$acc(s \to s') = min\left[1, B_F\right] \tag{4.2}$$

Insertion A random particle from the reservoir is inserted in a random position in the box. The acceptance probability of this move is given by:

$$acc(C \to C+1) = min\left[1, \frac{V}{\Lambda^3(N+1)} B_F\right] \tag{4.3}$$

where ΔE here equals to $[\mu - U(N+1) + U(N)]$, and Λ is the *thermal de Broglie wavelength*.

Deletion The transfer of a random particle from the box to the reservoir is accepted with a probability:

$$acc(C \leftarrow C-1) = min\left[1, \frac{\Lambda^3 N}{V} B_F\right] \tag{4.4}$$

where ΔE in this case equals to $[\mu + U(N-1) - U(N)]$

Algorithm 4 shows how the serial code works. In general, this algorithm executes *DisplacePercent* of the simulation steps as particle displacement moves, and the rest are divided equally between the insertion and deletion of particles. After each interval of steps, the algorithm ensures that a detailed balance is obeyed.

In each of the three moves, the bottleneck is to calculate the pairwise system's energy from all interacting particles. Since this is a Markov chain algorithm, each step should use the current system status to calculate the probability of acceptance for the next one.

4.2 Parallel Algorithm and Implementation Details

Due to the extremely multithreaded nature of the graphics devices, fine-grained parallelism is needed to keep the processors in the device active. For example, on the device, the smallest unit of execution is the

Algorithm 4 Serial Grand Canonical Ensemble Monte Carlo Algorithm

```
 1: Input: One box of size (N) and volume (V)
 2: Input: non-empty reservoir
 3: //Initialize N particles positions inside the box
 4: //Calculate total system energy
 5: //Main simulation loop
 6: for i= 1 to N$_{steps}$, step=1 do
 7:     //Randomly select a move type
 8:     $R \leftarrow$ rand()
 9:     if ($R <$ DisplacePercent) then
10:         //Attempt particle displacement
11:     else
12:         //Attempt particle transfer
13:         //Insertion/Deletion
14:         //Chose a random source of particle
15:         $Source \leftarrow$ rand()
16:         if ($Source < 0.5$ ) then
17:             //Source box is the box (Del)
18:         else
19:             //Source box is the reservoir (Insertion)
20:         end if
21:     end if
22:     //Solve if the system in equilibrium (Balance)
23:     //Periodically update system status to disk
24: end for
```

warp. All 32 threads in a warp must execute the same instruction and avoid branches in the code that lead to warp divergence. Another requirement to achieve good performance is to hide the memory latency. Even though the device has very high memory bandwidth, the relatively high latency of global memory accesses has to be addressed to get an efficient implementation.

4.2.1 Implementation without Cell List

For this applications, there are mainly two parallel restrictions. First, although the simulation steps have been implemented on the GPU, the CPU should decide which move to execute next and call the corresponding kernel. Moreover, the simulation should periodically be writing system status to disk, an operation not fully supported by current GPUs.

In this section, we describe in detail the simulation moves implemented on the GPU through kernel calls, without the use of cell lists. The implementation of cell lists is then discussed in § 4.2.2.

Calculating Total System Energy

Although this is the most time consuming kernel call for this application domain since all particle pair interactions are being calculated, this functions is being called only once to calculate the initial system energy. The total number of unique pairwise energy calculations is $N(N-1)/2$. If each thread is assigned to find the energy with one particle and all other particles, then the first thread is going to do $N-1$ (one row) calculations and the last thread is going to calculate the energy for only one pair of particles. To solve this thread imbalance issue, the work done by each thread has been mapped so that one thread is calculating pairwise energy for two particles. This cuts the number of required threads per kernel in half and allows for more threads to be scheduled at the same time per SM. Figure 3.4 illustrates this load balancing technique.

Particle Displacement within the Box

In this move, a randomly selected particle attempts to move in a random direction within the simulation box. The amount of energy that this particle is contributing to the system in the new location should be calculated, by first deducting the particle's contribution from its original location, then calculating the new system's energy for the new location. The difference in energy, ΔE, is used for calculating the Boltzmann factor in equation 4.2 to decide whether or not to accept the new location. Upon the acceptance of this particle's move, the system status should be updated, which includes the new energy, new virial pressure, particle's new position, and other configuration variables.

Initially, the *TryMove()* call wasn't on the GPU. In Algorithm 5, the kernel function *TryMove<<<>>>()* calls another function, *CalculateParticlesContributionTM()*, that returns ΔE; this last function was the kernel call. The decision of whether to accept the move or not uses this returned value from the device, and compares it to a randomly generated number. If the move is accepted the new particle position, current energy, etc. should be copied to the host. However, moving the function *TryMove()* to the GPU avoided these memory copies. Now, only thread zero in the last block makes the final calculations and applies changes to the system if the move is accepted, which eliminates the copying of all the parameters to the host. Algorithm 6 shows the parallel algorithm for the function *CalculateParticlesContributionTM()*.

Algorithm 5 Parallel particle displacement

```
1: Input: One box of size (N) particles, (V) volume
2: //Randomly select a particle to displace
3: $P \leftarrow$ rand()
4: $\Delta E \leftarrow$ CalculateParticlesContributionTM()
5: if thread 0 in last block then
6:     Use $\Delta E$ to calculate the acceptance rule
7:     //Select a random number $A$ in [0,1)
8:     $A \leftarrow$ rand()
9:     if A < ProbOfAcceptance then
10:        //Move accepted, apply changes
11:        //Update cell contents
12:    else
13:        //Move rejected
14:    end if
15: end if
```

Insertion and Deletion of Particles

To insert a particle from a non-empty reservoir, a random location should be generated. The device function *CalculateParticlesContributionTPT()* calculates the energy contribution for the new particle in the new location. The difference in system energy from this insertion is calculated by assigning one thread for each particle. The result of each pairwise energy interaction is stored in shared memory. The reduction process described in § 3.3.5 is then executed as shown in Algorithm 7. Algorithm 8 shows the next steps of calculating the probability of acceptance, generating a random number, and comparing the results. If the move is accepted, the current system parameters will be updated to reflect the new particle; otherwise, the system configuration remains unchanged.

The deletion move is similar to the insertion move except that the system is losing a particle and has to change the probability of acceptance as in equation 4.4. When the deletion step is accepted, the reservoir holds that particle, and the system configuration is updated accordingly.

4.2.2 Cell List Implementation

As we have seen in Figure 3.5, if the pair interaction is short-range, such as here, the simulation box can be decomposed into smaller domains, called cells, with the cell length S equal to or greater than the maximum interaction range r_{cut}. For a given particle, all interacting particles are located in the same or

Algorithm 6 CalculateParticlesContributionTM
(no cell list)

```
1: Input: One thread Per Particle
2: //Initialize shared memory
3: //Assign a particle for the current thread
4: for each ParticleID in the system do
5:     // For the particle in the old location
6:     //Determine the true distance between particles
7:     //applying periodic boundary conditions.
8:     //Calculate RadialDistance between particles.
9:     if RadialDistance within cutoff then
10:        //store interaction results in shared memory
11:    end if
12:    // For the particle in the New location
13:    //Determine the true distance between particles
14:    //applying periodic boundary conditions.
15:    //Calculate RadialDistance between particles.
16:    if RadialDistance within cutoff then
17:        //store interaction results in shared memory
18:    end if
19: end for
20: syncthreads()
21: //Apply reduction in shared memory
22: //Move results from each block to global memory
23: //Last block moves data from global to shared memory
24: //Apply reduction in shared memory
25: //final result is $\Delta E$
26: //return $\Delta E$ to caller via global memory
```

Algorithm 7 Calculate Particles Contribution (no cell list)

```
1: Input: One thread Per Particle
2: //Initialize shared memory
3: //Assign a particle for the current thread
4: //For all other particles in the box
5: //Determine the true distance with ParticleID
6: //Applying periodic boundary conditions
7: //Calculate RadialDistance between particles
8: if RadialDistance within cutoff then
9:     //Store interaction results in shared memory
10: end if
11: syncthreads()
12: //Apply reduction algorithm
13: //return $\Delta E$ to caller via global memory
```

Algorithm 8 Parallel Insertion/Deletion

```
1:  Input: One box of size (N) particles, (V) volume
2:  Input: A reservoir of non-zero size
3:  //Randomly select a position to insert into in the box
4:  //Find designated cell
5:  //Calculate the new particle's energy contribution
6:  $\Delta E \leftarrow$ CalculateParticlesContributionTPT()
7:  if thread 0 in last block then
8:      //Use $\Delta E$ to calculate the acceptance rule
9:      //Select a random number $A$ in [0,1)
10:     $A \leftarrow$ rand()
11:     if A < ProbOfAcceptance then
12:         //Move accepted, apply changes
13:     else
14:         //Move rejected
15:     end if
16: end if
```

directly adjacent neighboring cells on all axes. Figure 3.6 shows a 3D model of a simulation box where the dotted cells are the neighboring cells to the cell with crossing lines. Therefore, the cell list algorithm scales sublinearly as most of the operations have a fixed number of particles that need to be considered and this is independent of the size of the system. On the other hand, the cell list algorithm suffers from associated overheads of constructing and maintaining the cell structure. Next, we discuss the algorithm for implementing the cell list and factors of the design.

Building the Cell List and Binning the Particles

There are many techniques proposed [39] for implementing the cell list. We are implementing a technique that maintains a balance of the number of global memory accesses and the minimum number of wasted calculations. While some approaches use a linked list to store the indexes of the particle in each cell, others assign a fixed sized array of placeholders to every cell. The disadvantage with the first scheme is with the random memory accesses to pointers, which prevents threads from memory coalescing and prevents parallel access to particles. The disadvantage of the latter scheme is the extra memory that may be wasted. Although our implementation uses the latter scheme, experiments show that even for large systems, the maximum number of particles in a cell can be relatively small when a suitable cell size is chosen. This is discussed in more detail when we describe the cell size below.

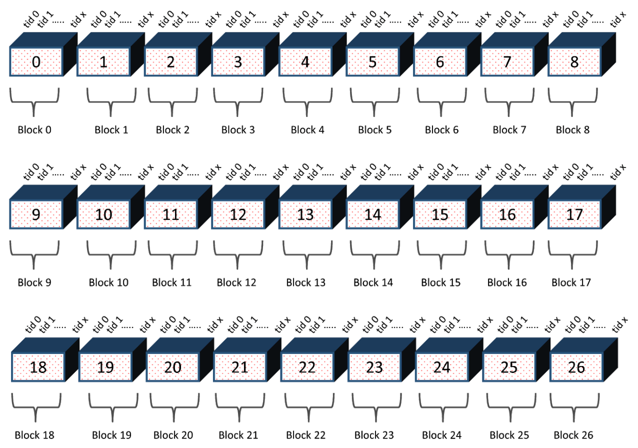
In this implementation, we applied cell lists to run entirely on the GPU, constructed once at the beginning of the simulation and requiring minimum maintenance. First, a data structure (*ParticlesInCells*) to hold the 26 adjacent cells for each cell is constructed, of size $27 \times T$, where T is the maximum number of particles a cell may have. Then, *ParticlesInCells* is bound to texture memory to take the advantage of caching. For the entire simulation run, values will be read through texture fetches. Using texture memory is important for such domain applications especially since one thread block will be reading the same cell indexes for the entire kernel call, allowing for a high rate of cache hits. Algorithms 9 and 10 show how the cell list implementation is used for finding ΔE for both the Particle Move and Particle Insertion/Deletion, respectively.

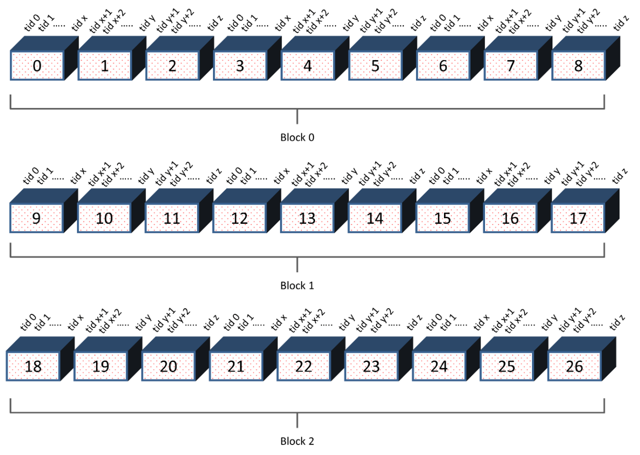
The process of placing particles in cells is called *binning* the particles. This process involves looping through the N particles and placing them into T cells. This process has been optimized so that when we first construct the simulation box and generate the random locations of all particles, we also execute the binning algorithm. This means we skip the overhead of a kernel call, and the need for extra global memory reads for the particle coordinates. The only performance disadvantage is the need to perform a read-modify-write operation to bin the particle to its location in the cell. However, with the Fermi architecture, these atomic operations are efficiently implemented and do not have significant effect on performance.

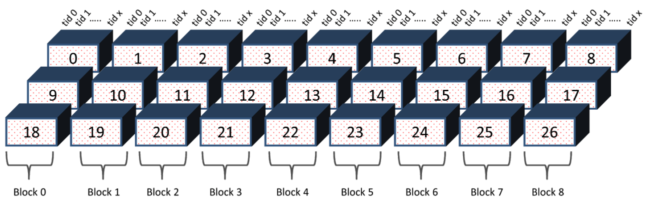
Cell Size

As mentioned before, a fixed size array of placeholders to every cell has been used to implement the cell scheme for our code. This parameter depends highly on the density of the simulation box and how close particles could be to each other. A larger cell size means extra wasted memory locations. On the other hand, a small cell size would affect the true number of particles in range, and a margin of error should be considered. Our implementation considers a cell size large enough to encompass all particles within range. The number of particles in a cell for a simulation with less than 8000 particles does not exceed 16 particles per cell. An upper limit of 32 particles per cell is chosen for larger problem sizes.

Another advantage of the close-to-optimal cell size is to eliminate moving a particle from one cell to another. For example, in the *TryMove()* kernel, if the particle is moving within the same cell, calculating the pairwise energy for the old and the new location will reuse the same particles, allowing for caching the coordinates of these particles.


(a) One block per cell


(b) Nine cells per block


(c) Three cells per block

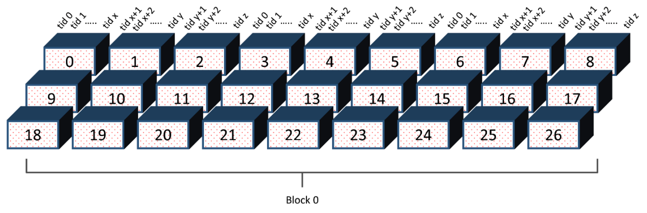

(d) Twenty seven cells per block

Figure 4.1: Different methods in assigning cells to thread blocks

Algorithm 9 CalculateParticlesContributionTPT function (cell list 9x3)

```
1: Input: three cells per block
2: //Initialize shared memory (x3)
3: /For the particle in the old location
4: //Find SourceCurrentCell
5: //Fetch three Neighboring cells From texture
6: if Non-empty cell then
7:     //For each ParticleID in SourceCurrentCell or neighboring cells
8:     //Assign one thread per particle
9:     //Determine the true distance between particles
10:    //applying periodic boundary conditions.
11:    //Calculate RadialDistance between particles.
12:    if RadialDistance within cutoff then
13:        //store interaction results in shared memory
14:    end if
15: end if
16: //For the particle in the new location
17: //Find DestCurrentCell
18: //Fetch three Neighboring cells From texture
19: if Non-empty cell then
20:    //For each ParticleID in DestCurrentCell or neighboring cells
21:    //Determine the true distance between particles
22:    //Applying periodic boundary conditions.
23:    //Calculate RadialDistance between particles.
24:    if RadialDistance within cutoff then
25:        //Store interaction results in shared memory
26:    end if
27: end if
28: syncthreads()
29: //Apply reduction algorithm
30: //return $\Delta E$ to caller via global memory
```

4.2.3 Assigning Cells to Blocks

In MD simulations, all particles move in the same randomly chosen direction at once. It is more efficient to assign one thread per particle, and one block per cell to take the advantage of caching all 26 neighboring cells. However, this is not necessarily true for MC simulations where only one particle is to be moved in each simulation step, and more blocks need more synchronization through atomic operations on global memory. Our implementation considers multiple options, with different numbers of cells per block and reports the difference in performance. This is depicted in Figure 4.1 where each thread block may compute 1, 3, 9 or 27 cells. This shows more ways to manage GPU resources according to the domain under study.

Algorithm 10 CalculateParticlesContributionTM function (cell list 9x3)

```
1: Input: three cells per block
2: //Initialize shared memory (x3)
3: //For the selected particle
4: //Find CurrentCell
5: //Fetch three Neighboring cells From texture
6: if Non-empty cell then
7:     //For each ParticleID in CurrentCell or neighboring cells
8:     //Assign one thread per particle
9:     //Determine the true distance between particles
10:    //Applying periodic boundary conditions.
11:    //Calculate RadialDistance between particles.
12:    if RadialDistance within cutoff then
13:        //Store interaction results in shared memory
14:    end if
15: end if
16: syncthreads()
17: //Apply reduction algorithm
18: //return $\Delta E$ to caller via global memory
```

One cell per block This is the most straight forward implementation expressed in Figure 4.1(a). In this scheme a kernel launches with 27 blocks, and the block size is set as the number of particles per cell. However, more synchronization between blocks and more total shared memory is needed. Yet, for systems with a uniform distribution of particles per cell, this works the best.

Three and nine cells per block In Figures 4.1(b) and 4.1(c), two different ways for resource management of the device are considered, one with nine blocks per cell, and the other with three cells per block, respectively. In both cases, one thread is assigned to one particle. These two mechanisms use more total local memory per kernel call, with less need for block synchronization.

Twenty-seven cells in one block Figure 4.1(d) shows that assigning all 27 cells to one block is a possible option too in this code. However, due to the huge number of threads per block for this implementation, more shared memory and other GPU resources are required, see Table 4.3. This means that only one SM is used to compute the entire simulation and the other SMs are idle. Although this is slower for one simulation, it would support the option to run multiple simulations simultaneously.

4.2.4 Assigning Threads to Particles

Previous work did study the performance of one cell per block and one thread per particle [121, 112]. However, for very large systems that were too large to be simulated prior to this work, it is worthwhile to study the performance of more than one particle per thread. An algorithm has been developed to assign multiple particles in more than one cell to a thread. For example, for a kernel of nine blocks, each block has been called with the size of one thread per particle in a cell, then each thread will calculate the pairwise energy for three particles from three cells (9x1x3). Another example is when only three blocks are called and each thread is calculating one particle from nine different cells (3x1x9). The reason for using more than one thread is to allow larger system sizes to fit to one SM. In addition, eliminating block synchronization are the reasons for using more than one particle per thread.

4.2.5 Adding Cell List Implementation to the Parallel Grand Canonical Algorithm

Adding the cell list implementation to the above CUDA implementation described in § 4.2 requires modifications in two steps. First, in calculating the pairwise energy of the system; for each type of move, only the current cell that a particle belongs to and the 26 neighboring cells' particles are considered. Second, for each type of move, slightly different steps are required as follows: Extra steps are required to update the cell list if a particle displacement move is accepted. The list of particles for both the source and the destination cells should be updated. To remove a particle from the cell list, an exhaustive search for the selected particle index in the source cell is executed. Then the last particle's index in that cell replaces the memory location in which the particle was stored, unless the particle of interest is the last particle in the list. In both cases, the counter of particles for that cell is decremented.

The destination cell update requires fewer operations. The particle is appended to list of particles in the destination cell, and the counter for that cell is incremented. If the particle is moving within the same cell, no action is required to the cell contents other than updating the coordinates of that particle.

4.3 Performance Results

In this work, we implement a fast parallel Monte Carlo simulation for the grand canonical ensemble using CUDA toolkit 5.0 [87] and evaluate the end-to-end application wall clock time against a single core

CPU implementation. The serial and CUDA implementations have been executed on a PC with an Intel Core i5-2500k CPU that has 8 GB of RAM running Linux kernel build 2.6.32 and compiled with the Intel 13.0.0 compiler. Parallel results are collected from running the code on the same machine using an NVIDIA GeForce GTX 480 graphics card. Relevant specs for this card can be found in Table 3.3. All code has been run with the full optimization flag passed to the compiler (-O3). All measurements have used one million simulation steps (attempts) and particle radius cut (r_{cut}) of 2.5. Furthermore, the statistical accuracy for both the serial and the CUDA code have been compared to those in [100] and show very close agreement. The Mersenne twister algorithm [77] has been used to generate the pseudo-random numbers used in our simulations.

Table 4.1 reports the performance of the sequential grand canonical code and the CUDA code for a number of particles ranging from 2^9 to 2^{18} with a corresponding volume of the simulation box ranging from 853.3 to 436905.6, doubling as the number of particles doubles. For small problem sizes of less than 4096 particles, no speedup has been achieved due to the low utilization of the device. However, when the simulation box has 4096 particles, the CUDA code started to outperform the serial code and about 2 times of speedup is shown. As the system size increases, more intensive mathematical operations are executed and the GPU code continues to show more speedup. For the largest problem size, we see a 15.8 fold speedup.

From Figure 4.2 we observe that the break-even point (dotted line) for the CUDA code to show any speedup over the serial code (black circles) is when the simulation box contains 4096 particles, which meets our expectation for this kind of problems. The main reason for this is because the kernel call overhead for smaller systems exceeds the gain of parallelism. Moreover, there is not enough arithmetic intensity for small systems. But, as the system size grows, the CUDA code shows more speedup, up to around 16 times for the largest problem size. The large number of particles and the associated arithmetic operations are enough to hide the cost of kernel calls for such large systems.

Since many variations of the parallel cell list algorithm have been evaluated, Table 4.2 explains the meaning of different notations. For a minimum cell size of $r_{cut} = 2.5$ and a density of $\rho = 1.0$, on average there are 10 particles per cell for systems of size 4096, 27 locations per cell are occupied for medium system sizes and 64 for large systems of more than 131072 particles. However, since the smallest execution unit on the device is the warp of 32, we chose as an upper limit the nearest multiple of 32. Also, the system size lower limit is given by the requirement to have at least three cells per dimension.

Table 4.1: Execution times in seconds for different algorithm implementations.

N	**Serial code**	**CUDA**	**Speedup**
512	2.8	22.3	0.13
1024	7.9	22.5	0.35
2048	14.2	22.8	0.62
4096	52.8	23.4	2.25
8192	116.3	26.7	4.36
16384	237.7	36.7	6.48
32768	502.2	56	8.96
65536	991.8	91.6	10.83
131072	2061	154.6	13.33
262144	4534.8	287	15.8

Table 4.2: Legend of blocks, cells, and threads per kernel call.

Notation	Block/Kernel	Cell/Block	Thread/Particle
27x1	27	1	1
9x3	9	3	1
3x9	3	9	1
1x27	1	27	1
9x1x3	9	1	3
3x1x9	3	1	9
1x1x27	1	1	27

As shown in Figure 4.3, different performance results for the cell list code over the original code have been shown for cells of size 2.75. It is clear that the best performance gain of the cell list is achieved when there are 262144 particles in the system. Moreover, a very important observation of these results is that the total amount of work per kernel is not increasing rapidly with the size of the system. In fact, what we are seeing here is the increased time in memory transfers between the host and the device, as we can also see from the end-to-end application execution time as shown Figure 4.4.

From the results in Table 4.4, we can see that when a kernel calls with 27 or 9 blocks, the performance is the highest with speedup of 8.31 and 8.32, respectively. In the case of 27 blocks only one cell is handled by each block. However, the extra overhead of synchronization and reduction among blocks is what makes 9x3 beats 27x1 execution time, even if the difference is negligible. Although the implementation of 9x3, in

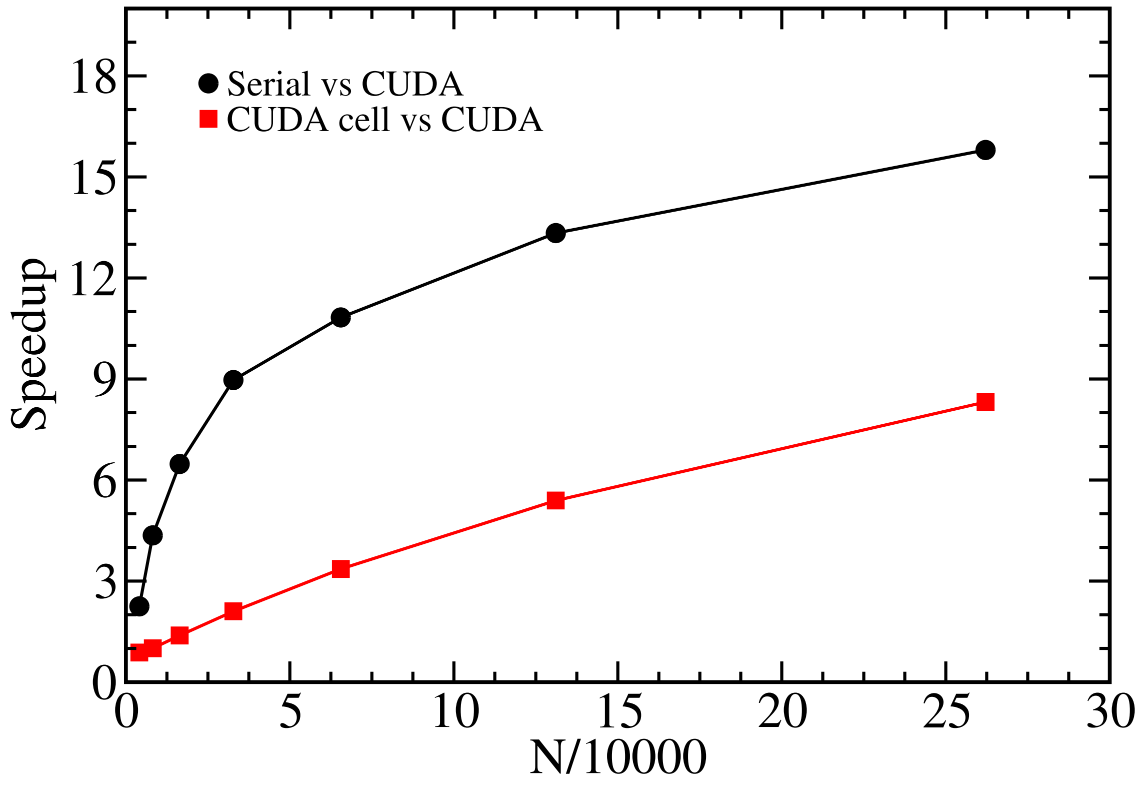


Figure 4.2: Speedup of 3 algorithms: Serial vs original CUDA and original CUDA vs CUDA with cell list.

Table 4.3, is using about three times the total shared memory of 27x1, the execution time doesn't exceed 4% difference. Moreover, from the same table we can see that managing GPU resources affect the overall performance of the CUDA code.

In Table 4.3, a comparison between different uses of GPU resources shows that efficient resource management can affect the performance advantageously. Although the average total amount of shared memory for the kernel call without cell list is almost the same as 27x1, a speedup of about one and a half times can be observed. As a result, we would recommend the use of 9x3 algorithm with a cell of size 2.75 for this problem under study.

A set of performance evaluation tests have been conducted using different cell sizes with a large number of particles in the box to evaluate the execution time of the CUDA implementation with cell lists against that of the CUDA implementation without cell list. From Figure 4.5 we note that the best results are achieved when the cell size was the minimum. Also, we observe that even with a different cell size, the algorithms of 27x1 and 9x3 both did the best.

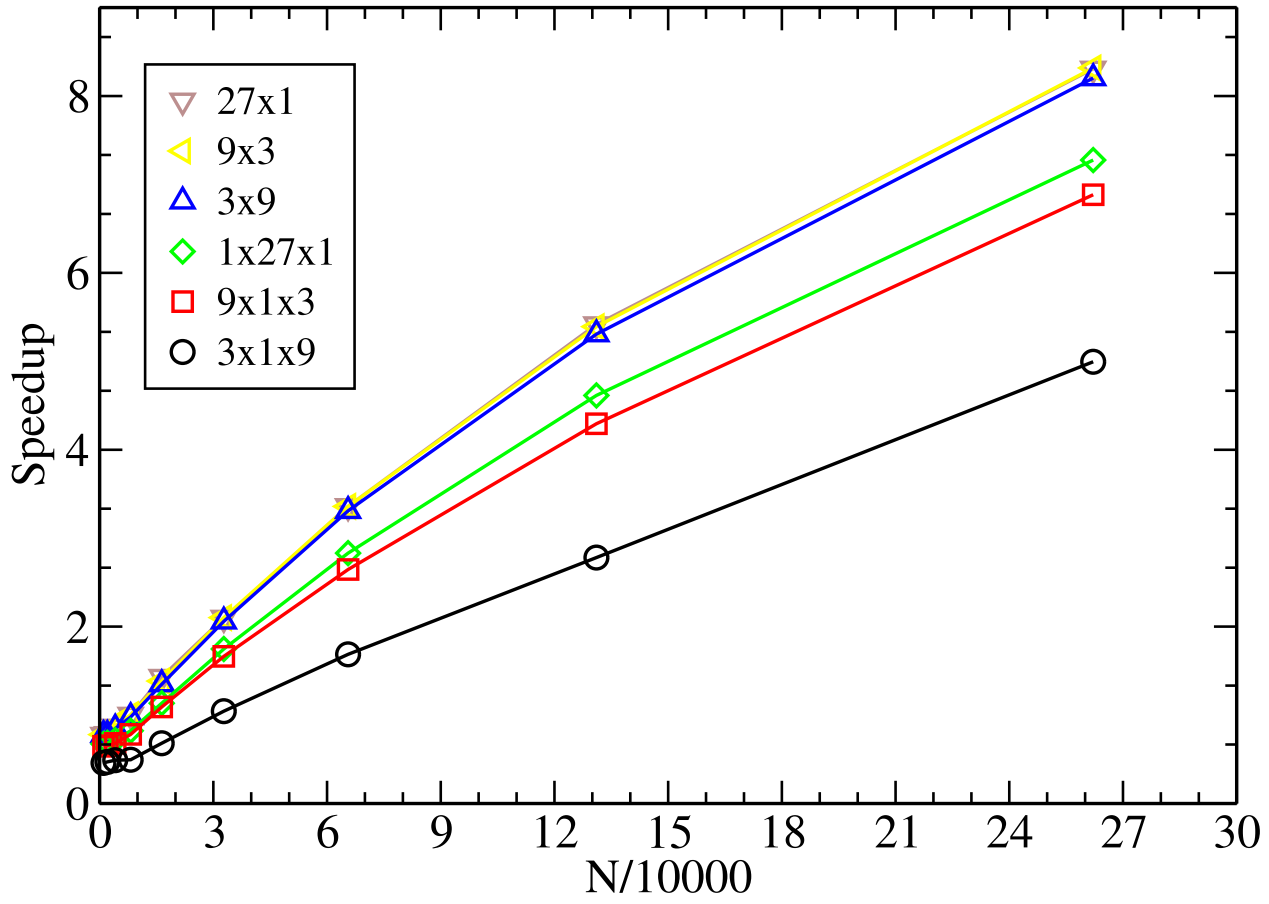


Figure 4.3: Speedup of CUDA code with different cell list codes over CUDA with no cell list.

Table 4.3: Different block sizes have different effect on resource utilization. For these experiments N is equal to 65536.

	Cells per block				
	27x1	9x3	3x9	1x27	No cell list
Total Shared Memory	524	1548	4620	11676	518
Occupancy(%)	16.7	50.0	37.5	47.9	16.7
Average exec time	25.7	26.5	27.2	32.40	36.7

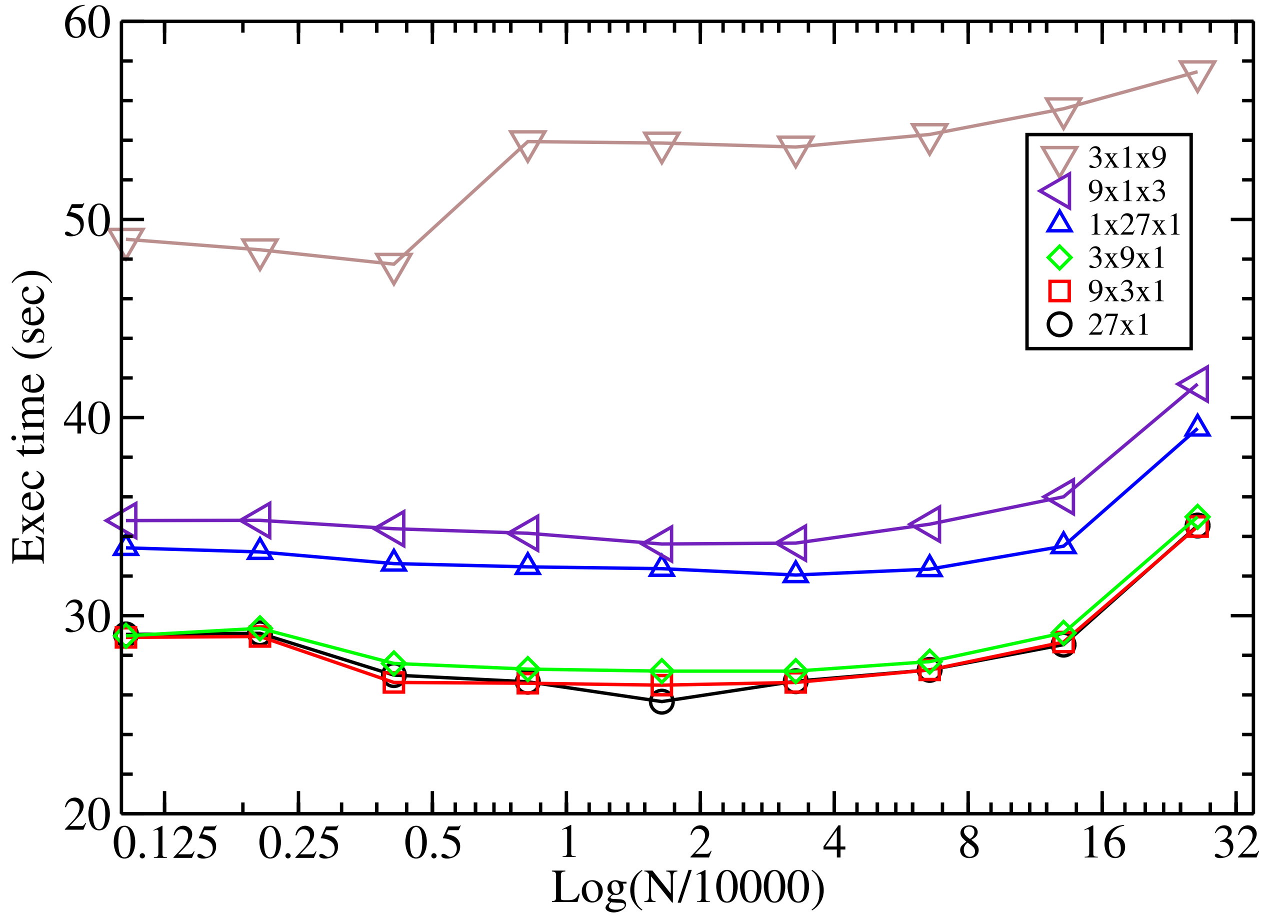


Figure 4.4: Execution times for different algorithms with CUDA cell list.

Table 4.4: Speedup of different cell list implementations over no cell list CUDA code.

N	27x1	9x3	3x9	1x27	9x1x3	3x1x9
1024	0.77	0.78	0.78	0.67	0.65	0.46
2048	0.78	0.79	0.78	0.69	0.65	0.47
4096	0.87	0.88	0.85	0.72	0.68	0.49
8192	1.00	1.00	0.98	0.82	0.78	0.50
16384	1.43	1.39	1.35	1.13	1.09	0.68
32768	2.10	2.10	2.06	1.75	1.66	1.04
65536	3.36	3.36	3.31	2.83	2.65	1.69
131072	5.42	5.39	5.31	4.61	4.29	2.78
262144	8.31	8.32	8.20	7.28	6.88	4.99

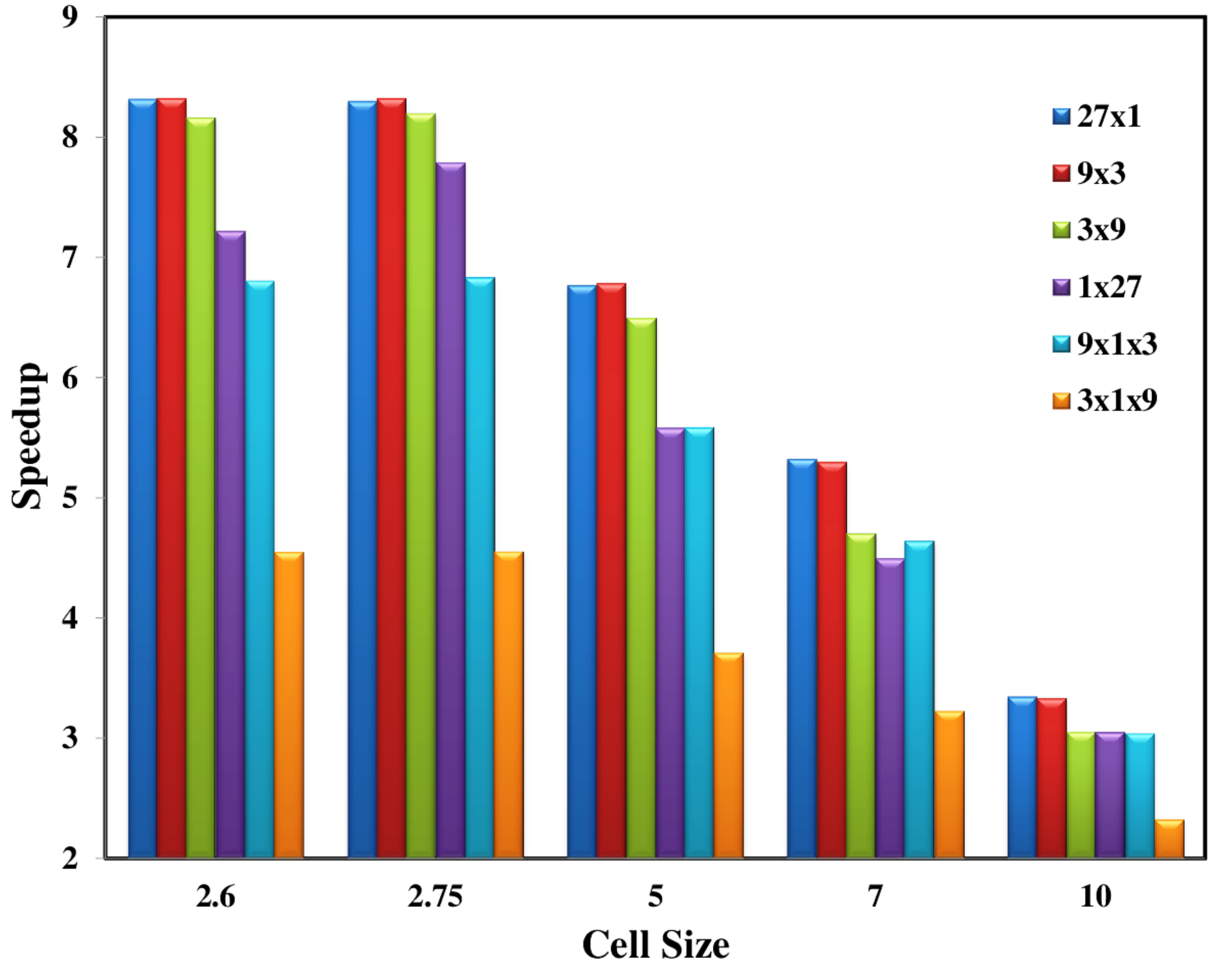


Figure 4.5: Cell size and speedup. of particles per cell per thread for N = 262144.

CHAPTER 5 Gibbs Ensemble: Two Simulation Boxes

In this chapter, we consider the Gibbs ensemble simulation in vapor-liquid phase coexistence, which can be determined from running a single simulation with two boxes [93, 39]. The MC method has been crucial in studying such systems and is necessary in order to allow for the number of atoms to vary with applied chemical potential via the Gibbs ensemble simulations. In high density systems, calculations may require a significant amount of computation power, with weeks to months of running time. For these systems to run in a reasonable amount of time, one should consider the use of massively parallel devices such as GPUs.

We first developed an efficient CPU code for MC simulation of the Gibbs ensemble. Then, we transformed the algorithm to run on the GPU and applied many tuning and optimization techniques specific to the GPU. Any general coding improvements that were made to the GPU code were back-ported to the CPU code, so both codes are highly optimized for their respective platform. In addition, both implementations produce identical results when choosing the same random seed, and are in close statistical agreement with the literature. Although the parallel version of the code shows speedup for small system sizes, larger systems show higher speedup, because larger systems have larger computation volume and the benefits of parallelism are greater.

5.1 MC Simulation of the Gibbs Ensemble

In a molecular system, there are several important correlated variables: volume, temperature, number of atoms, system energy, and pressure. For each *ensemble*, specific variables are fixed, and others remain independent. For instance, when the number of atoms, the volume, and the temperature are fixed through out the entire simulation, this is the Canonical ensemble or NVT. Another widely used ensemble is the Gibbs ensemble [93]. The Gibbs ensemble method directly simulates the average densities of coexisting fluid phases for bulk systems and confined fluids in equilibrium with a bulk region, and avoids the time consuming calculation of the chemical potentials. This method allows simulating phase equilibria under conditions where the pressure, temperature, and chemical potential(s) of the coexisting phases are fixed [39].

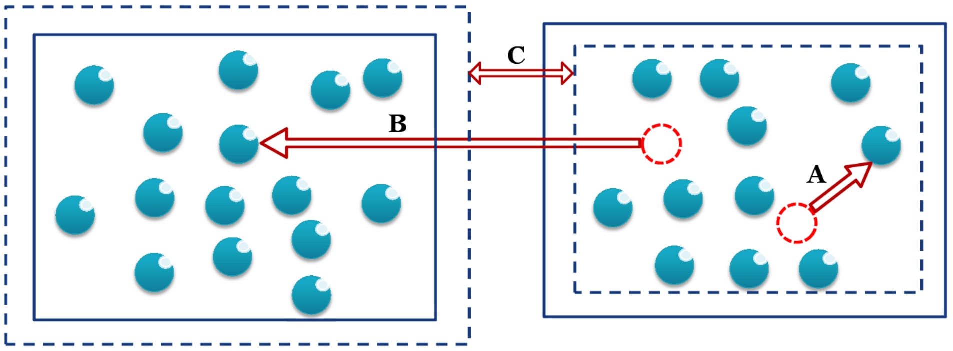


Figure 5.1: An illustration of the three move types for the Gibbs ensemble method. **A** represents atom displacement, **B** represents atom transfer between boxes, and **C** represents volume swap.

The Gibbs ensemble simulation model consists of two three-dimensional simulation boxes, with the atoms placed in them. Moreover, three trial moves are considered [93, 39], each of which has an acceptance rule. The criterion of acceptance is based on the weighting of the Boltzmann factor. where N is the total number of atoms in both boxes, s is the position of the atom, and U is the potential. Figure 5.1 illustrates the three types of moves conducted by an atom. The types of moves are:

A. **Atom Displacement:** A randomly selected atom attempts displacement within the same box. The acceptance rule for this move is given by:

$$acc(o \rightarrow n) = \min\left(1, B_F\right) \tag{5.1}$$

where $acc(o \rightarrow n)$ is the change in the configuration from o (the previous configurations) to n (the configuration after this displacement).

B. **Atom Transfer:** Figure 5.1 shows an attempt to transfer a randomly selected atom from one of the two boxes to the other box. Without lose of generality, assume that we generate configuration n from configuration o by removing one atom from box 1 and inserting this atom into box 2. This move is accepted according to the following rule:

$$acc(o \rightarrow n) = \min\left(1, \frac{n_1(V - V_1)}{(N - n_1 + 1) - V_1} \times B_F\right) \tag{5.2}$$

where V_1 is the volume for box 1 and n_1 is the number of atoms for the same box.

C. **Volume Swap:** An equal and opposite random change in the volume of the two boxes, such that the total volume remains constant. This move uses the acceptance rule:

$$acc(o \to n) = \min\left(1, \left(\frac{V_1^n}{V_1^o}\right)^{n_1+1}\left(\frac{V - V_1^n}{V - V_1^o}\right)^{N-n_1+1} \times B_F\right) \tag{5.3}$$

where V_1^o is the volume in the original state, V_1^n is the volume in the new state, and N is the total number of atoms in the system. For the new trial system, the scaled coordinates are defined as

$$s_n^N = r^N \times \left(1 + \frac{\sqrt[3]{V_n - V_o}}{V_o}\right) \tag{5.4}$$

As the simulation approaches the equilibrium state, the distance that we displace an atom is updated periodically so that an acceptance rate close to 50% is achieved for the displace and volume swap moves.

Based on the value of the acceptance rule, a random exponential function is used to determine whether or not the move is accepted. If the move is accepted, the new system configuration replaces the old one, otherwise, the old system configuration is retained.

The pseudocode in Algorithm 11 shows how the selection of the move is performed. Although all simulation moves are kernel calls, the main loop is executed on the host side, which involves many configuration updates. Moreover, a periodic flush of the data to the disk is required to write system status permanently. Hence, the simulation data should be transferred to the CPU to be written to the disk. These are limiting factors and overhead to the GPU use.

5.2 Method

Different MC simulations have different implementations due to the nature of the application being simulated. Applications such as the one implemented in this work have proprieties that make the process of creating an efficient parallel algorithm challenging. One of these properties is that this is a *Markov Chain* application where one simulation step depends on the results of previous steps. This prevents the execution of multiple steps in parallel.[1] So, the alternative is to create methods and techniques to solve these and other

[1]It might be possible to run multiple steps simultaneously if they are not interacting, such as two displacement moves, one in each box. However, the fraction of such non-overlapping moves is relatively low and would introduce significant overhead to determine whether or not moves are overlapping.

Algorithm 11 Gibbs Ensemble MC Algorithm

```
1: Input: Two boxes of equal size (N) and volume (V)
2: //Main simulation loop
3: for i= 1 to N$_{steps}$, step=1 do
4:     //Randomly select a move type
5:     $R \leftarrow$ rand()
6:     if ($R \leq$ N$_{disp}$) then
7:         //Attempt atom displacement
8:         //Randomly select a box
9:         Source $\leftarrow$ rand()
10:        //Pick a box
11:        if (Source $< 0.5$ ) then
12:            //(K) Attempt to displace an atom in box 1
13:        else
14:            //(K) Attempt to displace an atom in box 2
15:        end if
16:    else if ( $R \leq ( N_{disp} + N_{vol} )$ ) then
17:        //(K) Volume Transfer
18:    else
19:        //Attempt atom transfer
20:        //Randomly select a box
21:        Source $\leftarrow$ rand()
22:        //Pick a box
23:        if (Source $< 0.5$ ) then
24:            //(K) Source box is box 1
25:        else
26:            //(K) Source box is box 2
27:        end if
28:    end if
29:    //Solve if the system is in equilibrium
30:    //Periodically write system status to disk
31: end for
```

encountered parallel cases. Some tricks and optimization techniques used to get the results in § 5.3 include:

1. Mapping from serial operations to equivalent parallel operations with one order of magnitude reduction, assuming the GPU has enough resources to run N threads concurrently. For example, operations on all atoms such as the initialization of atom positions is reduced from $O(N)$ to $O(\log N)$. Another example, when scaling atoms in a volume swap move, the complexity can be reduced from $O(N^2)$ to $O(N)$; such operations are easily parallelized.

2. Block size has been chosen very carefully. The number of atoms in a thread block plays a crucial role for this application. In general, the massively threaded hierarchy has been supported by the

thread block structure that CUDA devices provide. However, thread blocks add more overhead to the kernel launch in two ways. First, thread blocks communicate through global memory that has very high latency compared to other memory types; Second, synchronizing thread block execution is not as straightforward as synchronizing threads using *__synchthreads()* nor synchronizing grids using *cudaDeviceSynchronize()*. However, block synchronization is not hard to implement using atomic operations, which have efficient performance in the Fermi architecture.

3. The use of loop unrolling and instruction-level parallelism are two common optimization techniques for such platforms. The CUDA compiler (nvcc) by default will try to unroll any loop for potential speedup. However, this is not fully optimized in the current nvcc compiler [85]. Moreover, the degree of unrolling should be examined by the developer for best performance. This technique may also eliminate the need for loop overhead when calculating the index and a dependent array index or offset.

4. The use of an efficient reduction technique. This particular application domain is dealing with hundreds of thousands of atoms that need to calculate the pairwise energy interaction for atoms. This is a very long and costly process with thread dependency and many synchronization barriers. Algorithm 12 shows our method of solving this problem and how it is implemented. First, the irregular problem size, when we don't have an exact power of two number of atoms, is addressed. Line 4 shows the operation that is executed to find the closest power of two that is less than the block size.

   For example, if the number of atoms to be processed equals to 117 atoms in a box, the equation in Line 4 will evaluate to $2^{\lfloor \log_2(117) \rfloor} = 2^6 = 64$ which is the number of atoms that need to be resolved first. While the fragment of code between lines 6 and 11 will solve up to 117 atoms in this example, the inner if statement in line 7 is to make sure that we don't try to add thread 63 with thread 127 or thread 55 with thread 119. Afterward, the process of reducing to 64 atoms is executed in lines 13 to 18. Then, loop unrolling is implemented for the first 64 atoms. We replace many nested if-elseif-else structures with a single switch statement, as seen in lines follow Line 22. This is a very efficient technique for removing any conditional thread divergence that if statements usually cause. Note that this method of reduction reduces the need for synchronization barriers after the switch statement as well.

   This algorithm is a generalization of the algorithm presented in [108] which is restricted to problem sizes that are a power of two. Algorithm 12 provides significant performance enhancement because it is

Algorithm 12 Calculating Total Energy in one box

```
__syncthreads()
// Allocate Energy[] in shared memory
// lpt stores the largest power of 2 that is less than the block size
lpt ← $2^{log_2(BlockSize)}$
// Reduce to the largest power of two atoms
if  BlockSize < MaxThPerBlock then
    if  $t_{id}$ + lpt < NumOfAtoms  then
        Energy[$t_{id}$] ← Energy[$t_{id}$+ lpt] + Energy[$t_{id}$]
    end if
    __syncthreads()
end if
// Sum in shared memory all locations larger than 32
for  i= lpt to 32, step=i / 2 do
    if  $t_{id}$ < i  then
        Energy[$t_{id}$] ← Energy[$t_{id}$ + i] + Energy[$t_{id}$]
    end if
    __syncthreads()
end for
// Reduction for the first 64 locations
if  $t_{id}$ < 32  then
    est ← min(lpt, 64)
    switch (est) do
      case 64: Energy[$t_{id}$] ← Energy[$t_{id}$ + 32] + Energy[$t_{id}$]
      case 32: Energy[$t_{id}$] ← Energy[$t_{id}$ + 16] + Energy[$t_{id}$]
      case 16: Energy[$t_{id}$] ← Energy[$t_{id}$ + 8] + Energy[$t_{id}$]
      case 8: Energy[$t_{id}$] ← Energy[$t_{id}$ + 4] + Energy[$t_{id}$]
      case 4: Energy[$t_{id}$] ← Energy[$t_{id}$ + 2] + Energy[$t_{id}$]
      case 2: Energy[$t_{id}$] ← Energy[$t_{id}$ + 1] + Energy[$t_{id}$]
    end switch
end if
```

being used twice for each volume swap move and atom transfer move, and once when the displacement move is executed.

5. Balancing the load across threads. In a naïve implementation, when calculating the interacting energy between atoms, each thread calculates the total interacting energy for one atom with all other atoms (one row), non-repetitive. To illustrate, thread zero is assigned to calculate the energy for atom zero with all other $N - 1$ atoms. Thread one is assigned to calculate the energy for atom one and all other $N - 2$ atoms, etc. The last thread will only calculate the interaction between two atoms. This means, to aggregate the sum of partial energies from all blocks, all threads in a block will wait for the first

thread to finish. Instead, this method has been optimized by assigning four threads per row to allow memory coalescing and leverage resource utilization. A transposing/remapping technique has been developed where each thread will do an equal amount of computation of $(N-1)/4$ interactions. That significantly improves the speedup of this process and balances the load among threads.

Another example of balancing the load on threads can be noticed in the displacement move. As most of the move attempts are displacement moves, the randomly selected atom energy contribution should be calculated for the old and the new locations. The algorithm has been modified to do both steps at the same time, using the same global memory reads and same shared memory for reduction, the difference in the old and the new energies are calculated by deducting the partial energy as soon as it is calculated from the current interaction by the same thread. The result is the difference in energy for this selected atom. On the other hand, the serial algorithm executes in two steps, calculate the old system energy, then the new system energy, to deduct the former and add the latter one.

6. The reuse of mathematical results. Studying the problem in hand and potential mathematical optimizations resulted in significant execution time reduction. Many mathematical results have been saved in registers and reused for the entire thread lifetime. Also, equation simplification and the removal of redundant operations. Moreover, the use of CUDA hardware implementations of functions improves performance. These types of optimizations are domain dependent and may enhance the overall run time if applied.

7. The use of asynchronous kernel calls and memory transfers, and the use of streams. Streams are non-blocking kernel launches where two or more predefined streams can execute two or more different kernels at the same time if they are dependency free. This is also called *concurrent copy and execute*. For instance, in a volume move, while atoms in one box are being scaled according to the new volume, another kernel call would be calculating the new energy of the other box.

8. Other optimizations include the recalculation of some values instead of executing memory transfers when it is cost effective. For example, scaling atoms in a box on the fly instead of maintaining a scaled copy of all the atoms and the overhead of transferring them in a volume move. Furthermore, other optimizations such as the use of device-to-device data transfer instead of using the host for intermediate memory transfer and caching frequently used memory locations.

Table 5.1: A comparison of a high and a low end GPU

	GTX 480	**Tesla M2090**
Processor clock	1401 MHz	1150 MHz
Memory clock	1848 MHz	1566 MHz
Memory size	1536 MB	6 GB
Memory Bandwidth	177.4 GB/sec	177 GB/sec
CUDA cores per SM	32	32
CUDA cores	480	512
Release date	April 2010	March 2011

5.3 Results and Discussion

In this work we implement a parallel MC simulation for the Gibbs ensemble using CUDA toolkit 4.2 [85] and evaluate the end-to-end application run time against a single core CPU implementation. Parallel results are collected from running the code on an NVIDIA GeForce GTX 480 graphics card. In addition to the specifications in Table 5.1, this card has 15 streaming multiprocessors, and compute capability 2.0. The serial and CUDA implementations have been executed on a PC with an Intel® Core™ i5-2500k CPU that has 8 GB of RAM running Linux kernel build 2.6.32 and compiled with the gcc 4.4.6 compiler and a different run with the Intel 13.0.0 compiler.

To avoid the bias of using a high end GPU we compare the performance of a commodity GPU to a commodity CPU. To illustrate, experiments has been conducted with a GeForce GTX 480 card which has an average price of $400 at the current market price and an Intel® Core™ i5-2500k which has an average price of $220. However, a top of the line GPU such as a Tesla card can produce even better performance than what we are reporting here. But, due to the lack of access to a high end CPU on that machine, the results with the Tesla M2090 are not reported here. While Table 5.1 shows a comparison of a high end GPU to an average one, Table 5.2 shows the different specifications between a high end CPU Intel® Xeon® Processor E5-2690 to an average one that could be found in regular user desktops.

All codes have been compiled with compiler optimization (-O2) and 64-bit environment flags turned on for the first case of using gcc compiler, and (-xAVX) and (-O2) flags passed to the Intel compiler. The latter

Table 5.2: A comparison of a high and a low end CPU

	Intel® i5	**Intel® Xeon**
No. of cores	4	8
No. of threads	4	16
Clock speed	3.4 GHz	2.9 GHz
Cache	6 MB	20MB
Max Memory Bandwidth	21 GB/s	51.2 GB/s
Release date	Q1 2011	Q1 2012

compiler achieved slightly faster execution time for both the serial and the parallel implementations, as can be seen in Tables 5.3 and 5.4. Other parameters and configurations used for collecting these results are the cutoff distance, r_{cut}, is set to 3.0, initial system density is kept consistent by adjusting the volume based on the number of atoms, and temperature is set to 1.0 in agreement with [100]. Also, output energies have been verified against the results in [90] and show close agreement. Moreover, we are running a million simulation steps[2] (move attempts) where 89% of the simulation steps attempt to execute atom displacement, 1% of the steps attempt to execute volume swap, and 10% attempt atoms transfer between boxes. However, due to the long running time of the serial version of the code, we are unable to report results for more than 131072 atoms in the system, which is not the case for the parallel version of the code which can scale up to the memory limit of the GPU and still executes in a reasonable time.

The use of a pseudo-random number generator is necessary on both the CPU, to decide on the move type, and on the GPU, to operate on each individual move. Because of the long periodicity and the widespread use in the application field, the Mersenne Twister (MT) algorithm [77] was chosen. It is used on the host side to generate a queue of random numbers concurrently with kernel calls, then this queue is transferred to the device in the same way. Moreover, a pointer to the front of the queue is maintained and passed to the device to be used by all threads. Since this process is being executed asynchronously with kernel calls and done infrequently, no noticeable performance effect on the end-to-end application execution time has been observed.

[2]Although hundreds of millions of simulation steps are required to obtain scientifically accurate simulation results, one million steps is sufficient to show the relative speedup of the GPU code.

Table 5.3: Execution times in seconds and speedup using the icc compiler.

N	Serial code	Parallel code	Speedup
1024	56.8	28.6	2.0
2048	157.3	32.2	4.9
4096	559.3	44.9	12.5
8192	2100.7	83.8	25.1
16384	8172.7	235.7	34.7
32768	32168.4	734.0	43.8
65536	114872.2	2722.1	42.2
131072	505546.0	10749.0	47.0

Table 5.4: Execution times in seconds and speedup using the gcc compiler.

N	Serial code	Parallel code	Speedup
1024	54.5	28.5	2.0
2048	159.4	32.3	4.9
4096	560.7	45.1	12.4
8192	2114.3	83.9	25.2
16384	8417.4	234.1	35.9
32768	32228.3	738.6	43.6
65536	121750.4	2728.3	44.6
131072	540139.6	10873.0	49.6

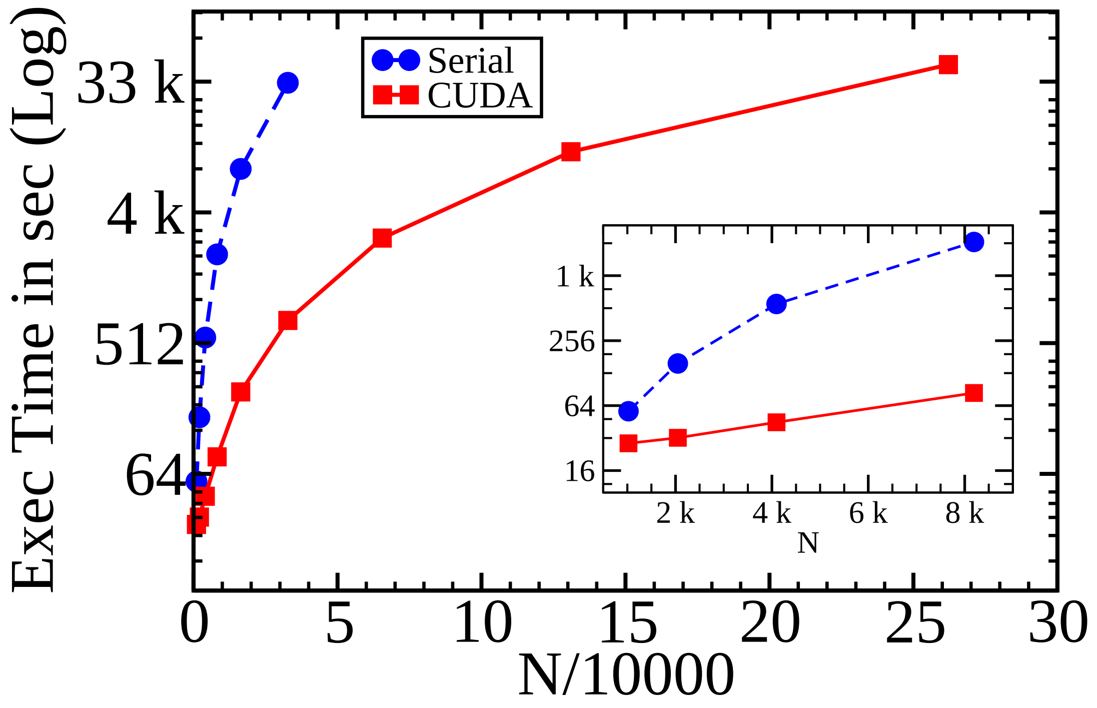


Figure 5.2: Execution time for the serial and the CUDA code

Table 5.3 presents the experimental results of the end-to-end application execution times for the CUDA and the serial code for different problem sizes, where N is the total number of atoms in the two boxes. Each data point is an average of five runs (except the last data point in the serial code), with a difference between this average and any single run always less than 3%. The minimum speedup achieved, 2x, occurs when there are not many atoms in the system; a total of less than 1024 atoms in both boxes does not provide speedup over the serial version due to the memory latency, block synchronization, and kernel launch overhead. However, more speedup is achieved rapidly when there is a chance to do more work and the amount of computation exceeds the parallel overhead. The highest recorded speedup is a factor of 47x when we have 131072 atoms in the system. In terms of optimizations, the best results were obtained by using 4 times loop unrolling and tuning the thread block size to 128 threads.

Figure 5.2 shows that as the system size grows, the execution time increases for both the serial and CUDA codes. However, the CUDA code execution time grows more slowly than the serial one. This relatively slow

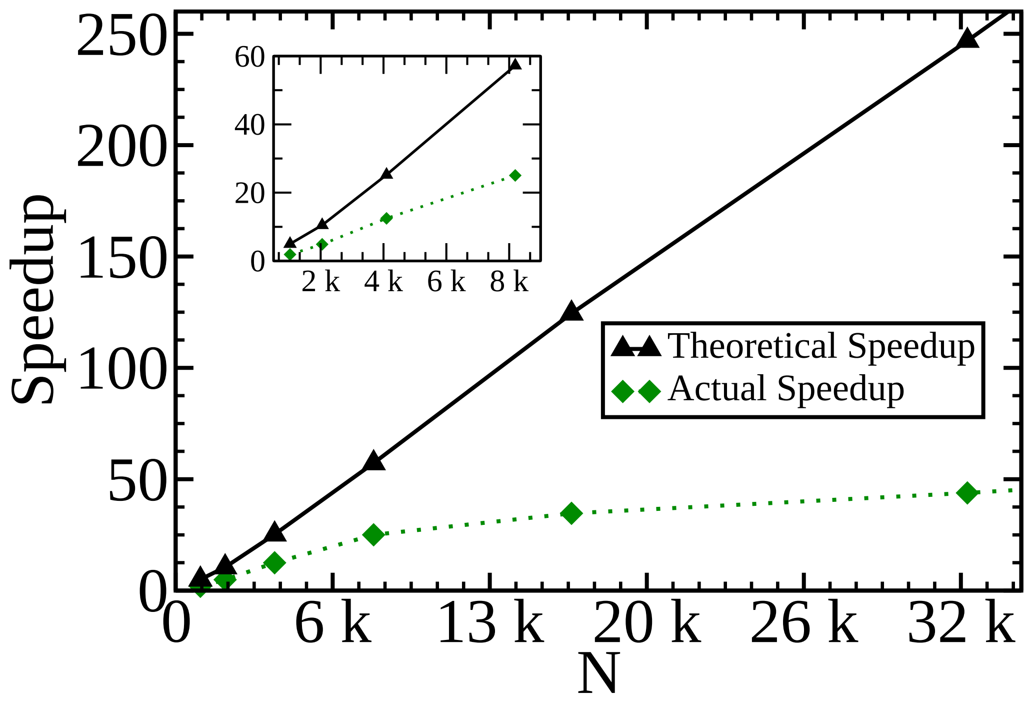


Figure 5.3: The actual speedup and theoretical speedup of the CUDA code

growth is a result of the higher utilization of the GPU. Moreover, Figure 5.2 shows the actual speedup of the CUDA code over the serial code with the speedup trajectory expected to keep rising as the problem size increases until the GPU resources reaches a saturation state. Note that compiling with the Intel compiler follow the same trajectory of execution times.

In addition, Figure 5.2 shows larger system sizes that the serial code can't scale to and supports this assumption. Also, for problems of size between 2K and 16K, the utilization of the GPU is relatively low for a single thread block, so we can use more of the GPU for a larger problem size (more blocks) without much overhead. Hence, the execution time increases very little between 2K and 16K. However, the speedup rate is expected to be slower than the region when we have 2K to 16K atoms since the overhead on memory transfer and block synchronization is increasing as well. Moreover, we notice that the speedup is not significant for small systems with less than 4K atoms, because we have low hardware utilization and high memory latency versus low parallel computation time.

Figure 5.3 shows the inherent parallelism of our code. The theoretical speedup was measured by replacing each kernel call with a simple kernel call having just one thread that did no work. This models an infinitely fast GPU with unbounded resources. Although this is an unrealistic measure of peak performance, it does allow us to estimate the inherent sequential percentage of the code. As Figure 5.3 shows, there is a great deal of parallelism in the code, which explains why we are able to obtain these speedups.

CHAPTER 6 Configurational Bias Gibbs Ensemble

6.1 Introduction

In the simulation of chain molecules, where there is an insertion of a particle into the simulation box such as the grand canonical and the Gibbs ensemble simulations, the probability of accepting the random trial is extremely low. Hence, the traditional MC techniques will fail to converge in a reasonable number of steps especially for complex fluids of dense chain systems and monolayers. However, an extension of the standard MC algorithm that allows us to overcome such limitations of MC methods is the configurational biased technique [104, 110, 37, 32]. This method biases the simulation moves so that they have an enhanced probability to fit into the existing configurations.

The original Gibbs ensemble method, discussed in § 5, executes three moves. Particle transfer between the simulation boxes is one of these moves. In the particle transfer move, a randomly selected particle is transferred from one box to an arbitrary location in the other box. For configurational bias, an extension is added to this type of move so the probability of this move is not entirely random, rather it has an enhanced probability suitable to the existing configuration. One advantage of using such techniques is that they help enhance the efficiency of the simulation. However, to satisfy detailed balance[1], we should update the acceptance rule in such a way that the bias is removed from the sampling scheme.

The Rosenbluth approach implements a self-avoiding random walk [104] that has been used later in many algorithms such as [47, 109, 36, 75] and others, to implement MC techniques to conduct large scale conformational changes of the chain molecule in a single trial move. The original and updated algorithms consist mainly of a generation of a chain conformation with a bias that ensures that these conformations have high probability of acceptance. Then the Rosenbluth weight is used to bias the acceptance of these trial conformations. This is the same technique that we use to implement the CB algorithm in this chapter.

[1]To satisfy detailed balance, the limiting distribution of the Markov chain should exist, is unique, and is the Boltzmann distribution [73].

6.2 CBGEMC Method and Implementation

The advantage of MC simulation over MD techniques is that MC techniques can be used for systems where it is not possible to change the conformation of macromolecules by successive small steps. Initially, the Gibbs ensemble MC technique used to be limited to systems containing atoms or small molecules. Yet, by combining the Gibbs ensemble method with configurational bias MC, acronym CBGEMC or for short CB, the Gibbs ensemble method can be made to work for much longer chain molecules. Combining these two methods allowed researchers to compute some of the first vapor-liquid coexistence curves for chain molecules, such as united-atom n-alkanes. The result is a large increase in the acceptance rate for insertions of polyatomic molecules into liquids.

To implement this method, we use the ratio of the 'Rosenbluth' weight factors of the new and old configurations to decide if a trial move should be accepted or not. Algorithm 13 shows the serial steps to execute a particle transfer from one box to the other in CB. Also, this can be seen in Figure 6.1 where the number of CB trials is five. In this figure, four attempts for four possible positions of the true particle in the source box are considered and the energy for each one is calculated, corresponding to Lines 4-14 in Algorithm 13. In the destination box, five locations are drawn randomly, and the associated energy is calculated, which maps to Lines 14-22 in Algorithm 13. Then, one of these locations is selected to be used in calculating the acceptance rate. Upon move acceptance, the system status will be updated and the new configurations take place.

This extension adds an extra overhead to the original algorithm by performing extra computations especially for higher K. On the other hand, the trials are independent and have the potential to run in parallel, such as the operations executed in Lines 4-22. The only serial step is when the results of all trials are needed to pick one random location and calculates the acceptance rate for it. That being said, this problem can map to the GPU exploiting two different levels of parallelism:

- **Low level parallelism**

  The lower level parallelism that has been explored here is similar to the one we applied before when we were running *CalculateParticlesContribution*(). In this case, multiple thread blocks calculate the energy of one particle with all other particles in the old and new locations asynchronously, see § 3.4 and the related text. In this model, thread zero of each block collects the block summary before the last thread of the last block to finish finds the sum of all block results. This operation requires GPU atomic operations to synchronize blocks since there is no way to predict which block will finish last.

On the other hand, an advantage to this mechanism is that all threads in a thread block can share information and be synchronized easily. The limitations on parallelism are device specifications on how many blocks could be scheduled and run simultaneously.

- **High level parallelism**

  On the GPU, this is kernel level parallelism. When multiple independent operations are required, such as multiple CB trials here, a higher level of parallelism can be implemented for running multiple trials in parallel by calling different kernel functions to be carried out on different streams. To achieve this higher level of parallelism, multiple simultaneous kernel calls must be executed, mapping streams to trials, as illustrated in Figure 6.2. This allows the device driver to schedule multiple streams to run concurrently on the device.

  Although in this level of parallelism kernel calls are independent to some extent, the need for a synchronization mechanism at the end to calculate final results is required, referred to as synchronization barrier in Figure 6.2. The final collected result from all streams is used to make the decision of whether or not to accept the move. This could be executed on the host side, and hence all device variables should be transferred to the host, which is undesirable due to the high memory latency associated. Or, we can design the synchronization barrier using atomic operation on the device and execute the last step of the algorithm on the device. However, when implementing this technique, we were unable to synchronize the output of the streams using atomic operations. Instead, we used the synchronization barrier on the host side followed by another kernel call to make the final decision based on the required output calculations from the previous step.

  Listing 6.1 shows the two kernel calls used for the particle transfer move with multiple streams using CUDA. The first kernel will be called once for each of the CB trials where the current stream ID is used to bind the kernel call to a specific stream. Using the function to synchronize kernel calls, line 5 of Listing 6.1, the execution will not proceed to the next kernel call until all results from all CB trials finish. Only then the second kernel will be called and data stored in global memory from previous kernel calls will be used.

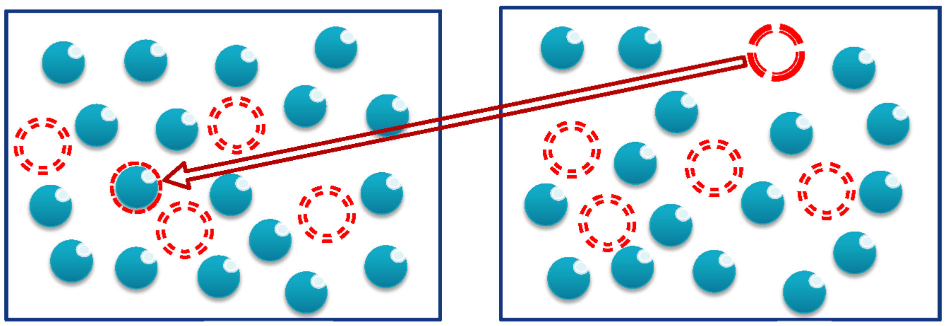


Figure 6.1: Particle transfer move with CB and K equals five.

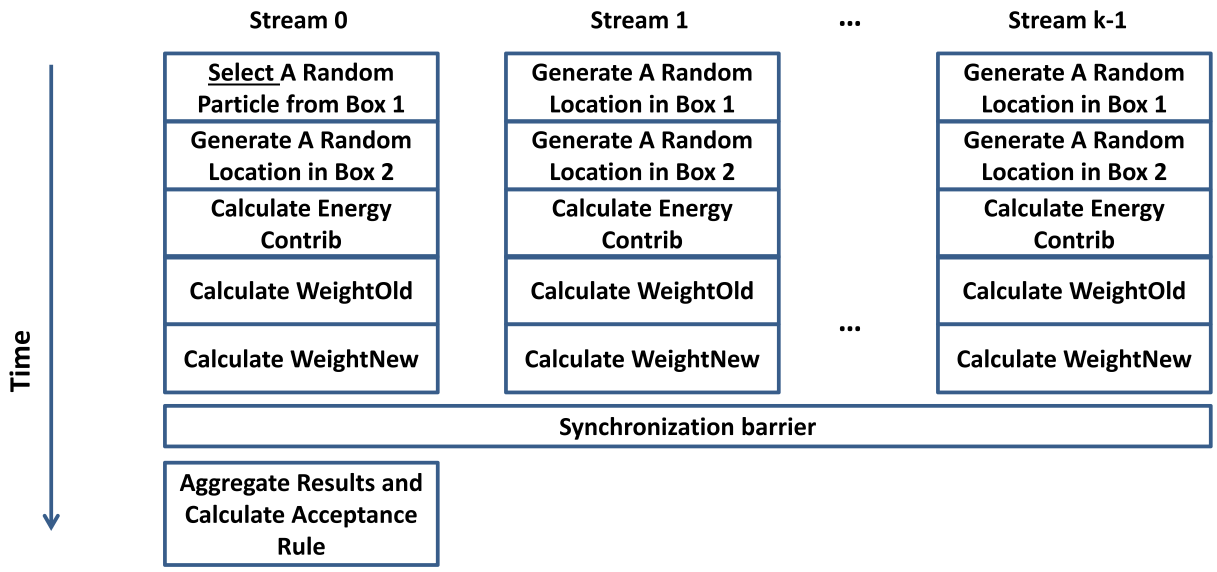

Stream 0
Stream 1
...
Stream k-1
Time
Select A Random Particle from Box 1
Generate A Random Location in Box 1
Generate A Random Location in Box 2
Calculate Energy Contrib
Calculate WeightOld
Calculate WeightNew
Synchronization barrier
Aggregate Results and Calculate Acceptance Rule

Figure 6.2: Timeline of particle transfer in CB execution using K-1 independent streams.

Listing 6.1: Code for running multiple stream kernels.

```
for(int StreamId = 0; StreamId< K; StreamId++){
    TryParticleTransfer<<<GridSize,BlockSize,
                                      SharedMem, StreamArr[streamId]>>>(...);
  }
cudaDeviceSynchronize();
AcceptTPTMove<<<1,1>>>();
```

Algorithm 13 Serial Particle Transfer Move in the CBGEMC Algorithm

```
1: **Input:** Two boxes of equal size (N) and volume (V)
2: **Input:** The number of CB trials K
3: // Assume source box is box 1
4: // Randomly select a particle from box 1
5: call **CalculateParticlesContribution**() for the selected particle
6: $W_{old_0} \leftarrow \exp(-\beta[U_0^1])$
7: $TotalW_{old} \leftarrow W_{old_0}$
8: // Assign new positions, calculate energies, and weights in box 1 for K-1 trials
9: **for** o = 1 **to** K-1 **do**
10:     $SelectedInsertionPosition_{o1} \leftarrow$ **randCoordinates**()
11:     call **CalculateParticlesContribution**() for $SelectedInsertionPosition_{o1}$
12:     $W_{old_o} \leftarrow \exp(-\beta[U_o^1])$
13:     $TotalW_{old} += W_{old_o}$
14: **end for**
15: // Assign new positions, calculate energies, and weights in box 2 for K trials
16: $TotalW_{new} \leftarrow 0.0$
17: **for** n = 0 **to** K-1 **do**
18:     $SelectedInsertionPosition_{n2} \leftarrow$ **randCoordinates**()
19:     call **CalculateParticlesContribution**() for $SelectedInsertionPosition_{n2}$
20:     $W_{new_n} \leftarrow \exp(-\beta[U_n^2])$
21:     $TotalW_{new} += W_{new_n}$
22: **end for**
23: // Determine probability of trial selection
24: **for** i = 0 **to** K-1 **do**
25:     $ProbSelectTrial_i \leftarrow W_{new_i}/TotalW_{new}$
26: **end for**
27: // Pick the best trial
28: $SelectedTrial \leftarrow 0$
29: $WeightDraw \leftarrow$ **rand**()
30: **repeat**
31:     $CumulativeWeight += ProbSelectTrial_{SelectedTrial}$
32:     $SelectedTrial$++
33: **until** $WeightDraw > CumulativeWeight$
34: $Coefficient \leftarrow \beta N/V$
35: // Remove sampling bias using Rosenbluth weight
36: $ProbOfAcceptance \leftarrow Coefficient * TotaolW_{new}/TotalW_{old}$
37: **if** ($ProbOfAcceptance <$ **rand**()) **then**
38:     // Accept the transfer of the particle
39:     // Apply new system configurations
40: **end if**
```

6.3 Results and Discussion

In this chapter we present an implementation of a parallel MC simulation for the Configurational Bias Gibbs ensemble using CUDA toolkit 5.0 [87] and evaluate the end-to-end application run time against a single core CPU implementation. Then we implement multi-stream kernel calls to carry out the multiple CB trials in parallel. Parallel results are collected from running the code on an NVIDIA GeForce GTX 480 graphics card and Tesla K20c. The Tesla K20c is the first GK110 card. It has the following specs [89]:

- 1 Kepler GK110 chip
- 5 GB of memory size (GDDR5)
- 2496 CUDA cores
- Processor core clock of 706 MHz
- 1.17 Tflops peak double precision floating point performance
- 2.6 GHz memory clock
- 208 GB/sec memory bandwidth
- 32 streaming multiprocessors
- Compute capability 3.5

The most important new features in the Kepler GK110 GPU architecture are the SMX, Dynamic Parallelism, and Hyper-Q. Further details can be found in the Kepler GK110 whitepaper[86]. However, specially tuned algorithms to utilize the new Kepler features are required to get peak performance.

Performance numbers presented here have been conducted with two desktop PCs setup. The first run has been executed on a PC with an Intel® Core™ i5-2500k CPU that has 8 GB of RAM running Linux kernel build 2.6.32 and compiled with the gcc 4.4.6 compiler and loaded with GTX 480 card. The second PC, with the Kepler card, has a Core 2 Duo™ with 2 GB of RAM and running Windows 7. The code in the latter desktop computer was compiled with Microsoft Visual Studio 2010 C++ compiler.

All codes have been compiled with full optimization flags and 64-bit environment flags turned on. Other parameters and configurations used for collecting these results are the cutoff distance, r_{cut}, is set to 2.5, initial

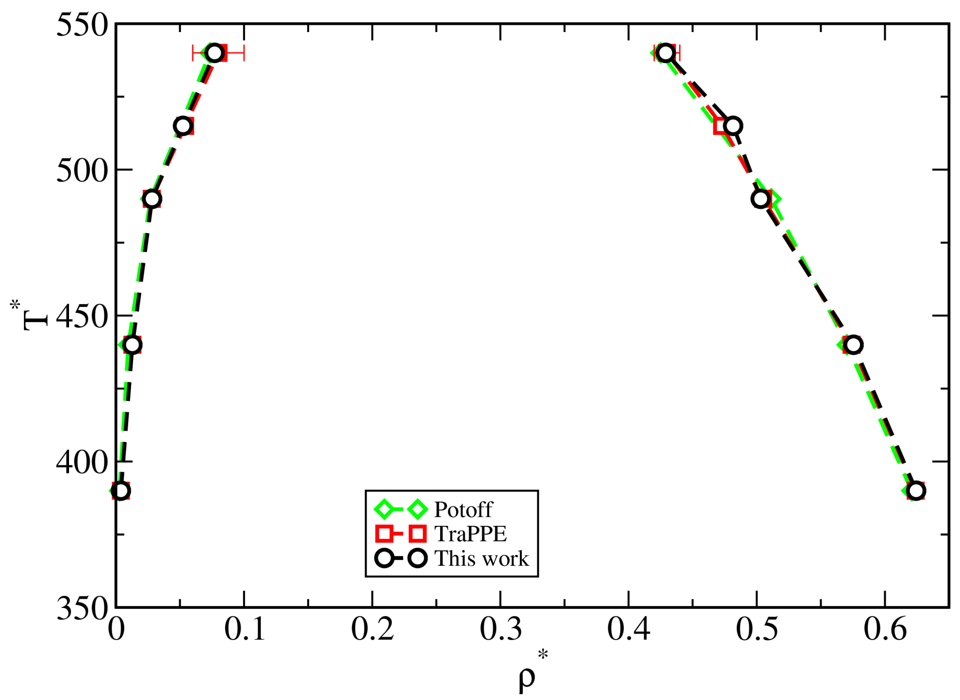
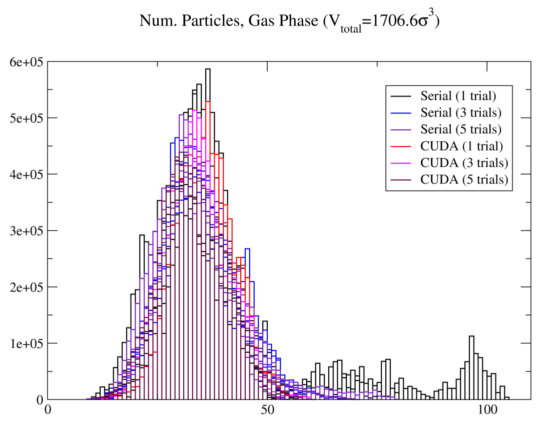


Figure 6.3: Vapor-liquid coexistence curves for Lennard-Jones fluid. Simulation runs are done by Jason Mick.

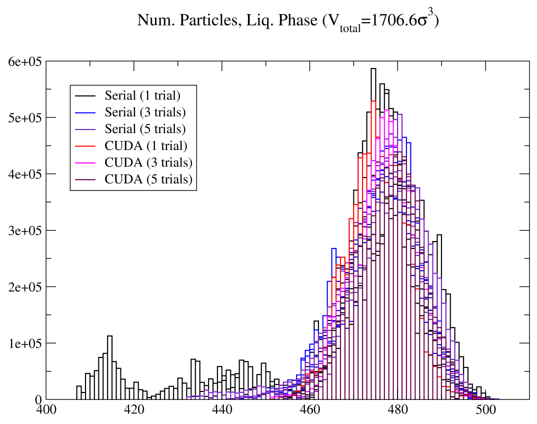
system density is kept consistent by adjusting the volume based on the number of atoms, and temperature is set to 1.0, in agreement with [100]. Moreover, we are running a million simulation steps[2] (move attempts) where 89% of the simulation steps attempt to execute atom displacement, 1% of the steps attempt to execute volume swap, and 10% attempt atoms transfer between boxes. However, due to the long running time of the serial version of the code, we are unable to report results more than 131072 atoms in the system, which is not the case for the parallel version of the code which can scale up to the memory limit of the GPU and still executes in a reasonable time.

To validate the correctness of the serial and parallel code, we compare our serial code results to those obtained from [99] and the transferable potentials for phase equlibria (TraPPE) force field in Gibbs ensemble, as can be seen in the Vapor-Liquid coexistence curve in Figure 6.3. All curves are in close agreement to each other in both the vapor and liquid states. Then, we compare the sample of particle distribution across the simulation runs. In Figure 6.4, the frequency with which the system is at a specific density is shown. The serial and CUDA results are in close agreement in sampling particle histogram for different values of K for

[2]Although hundreds of millions of simulation steps are required to obtain scientifically accurate simulation results, one million steps is sufficient to show the relative speedup of the GPU code.



(a) Histogram of gas phase



(b) Histogram of liquid phase

Figure 6.4: Histogram sampling the distribution of the gas and liquid phases resulted from our code (GOMC).The serial and CUDA code are in a close agreement. Simulation runs and the figure are done by Jason Mick.

both the gas and liquid phases. From this figure, we can see that the two states (gas and liquid) in the two simulation boxes reached a state of separation.

We implement a serial code and two algorithms suitable for the GPU. The serial algorithm is the basis for the other two implementations. It is used to validate the correctness of the algorithm under development and for performance comparisons. All these pieces of code are part of the GPU optimized Monte Carlo (GOMC) project [33]. Our focus here is on the system behavior with different numbers of CB trials, referred to by K, with and without streams for launching multiple kernels asynchronously. The higher the K, the sooner the system reaches an equilibrium state. However, researchers are interested in different values of K for different system configurations, so we study possible values of K, and the effect on the system performance.

Tables 6.1 and 6.2 show the end-to-end application execution times for the CB code for different configurations. We note that as N increases, the execution time increases rapidly for all cases. This is because with large systems, more operations are required, such as the time to calculate the total energy for the entire system for a volume swap, or a particle's energy contribution with all other particles in a box, and larger memory transfers are executed on the data.

It is clear from Tables 6.1 and 6.2, also, that as K increases the execution time of the serial algorithm increases. However, a lower rate of execution time increase can be seen in the parallel code execution times.

Table 6.1: Execution times in seconds for CB for different number of CB trials (K). The last two columns are runs on a different PC than the other three.

	N	**Serial**	**GTX 480**	**480/Streams**	**Tesla K20c**	**K20c/Streams**
K = 1	**1024**	46.1	31.6	32.0	64.9	72.9
	2048	148.7	35.3	35.0	69.5	75.2
	4096	487.8	47.2	47.0	79.5	85.9
	8192	1788.3	86.4	86.1	100.8	109.6
	16384	6913.2	233.0	233.6	183.2	197.6
	32768	27284.8	704.7	709.7	456.2	478.3
	65536	110941.0	2622.9	2649.4	1558.9	1656.0
	131072	526759.5	10694.2	9589.0	5858.0	6129.0
K = 3	**1024**	52.7	35.6	33.1	75.3	73.0
	2048	171.6	39.8	37.1	81.8	78.9
	4096	576.6	52.3	51.4	93.0	90.2
	8192	1851.5	93.7	90.9	116.0	114.2
	16384	7032.3	244.4	243.4	200.8	201.9
	32768	28363.6	713.0	728.2	480.8	493.9
	65536	108549.0	2619.7	2708.6	1568.0	1686.0
	131072	464814.0	9720.0	9589.0	6047.0	6162.0
K = 5	**1024**	62.5	39.0	33.9	82.3	77.7
	2048	200.4	43.8	38.7	88.4	83.7
	4096	577.3	57.6	52.7	99.4	94.6
	8192	2048.3	98.9	95.3	124.2	120.4
	16384	7072.0	254.7	247.7	210.4	208.3
	32768	28994.9	733.4	746.0	497.2	505.9
	65536	109060.0	2686.5	2705.0	1583.0	1684.0
	131072	453653.5	10359.4	10177.1	5856.6	6290.0
K = 7	**1024**	64.1	42.8	34.6	85.3	81.1
	2048	197.6	47.9	38.4	90.7	87.5
	4096	672.4	61.5	55.1	102.3	99.3
	8192	2178.9	104.6	96.7	128.9	125.1
	16384	6927.9	261.5	247.9	214.5	211.3
	32768	29477.2	747.7	778.3	508.1	505.9
	65536	117195.0	2698.0	2761.2	1616	1725.8
	131072	497626.5	10455.6	10645.5	5882.0	6404.0

Table 6.2: Execution times in seconds for CB for different number of CB trials (K). The last two columns are runs on a different PC than the other three.

	N	Serial	GTX 480	480/Streams	Tesla K20c	K20c/Streams
K = 10	**1024**	68.1	48.3	36.1	94.5	84.9
	2048	215.7	53.9	41.4	101.2	90.9
	4096	759.7	68.6	56.3	114.0	102.6
	8192	2151.6	113.6	102.8	141.2	130.8
	16384	7119.9	271.9	251.1	231.8	217.2
	32768	29320.5	769.4	794.0	531.8	521.2
	65536	117257.0	2782.8	2817.0	1624.0	1722.0
	131072	495616.0	11045.6	10874.9	5868.0	6218.0
K = 15	**1024**	74.2	57.6	38.5	111.3	91.9
	2048	229.6	63.8	43.9	118.2	98.4
	4096	728.6	79.3	61.5	132.2	110.8
	8192	2465.5	126.3	108.1	165.7	141.5
	16384	7489.2	291.5	262.2	256.2	229.5
	32768	31376.9	794.2	818.2	577.2	537.0
	65536	111810.0	2783.3	2713.9	1743.0	1786.0
	131072	449458.0	10711.8	11198.2	6248.0	6517.0
K = 20	**1024**	81.6	67.1	40.4	128.3	100.1
	2048	246.2	73.1	46.3	135.9	107.6
	4096	786.1	90.8	64.1	151.6	121.2
	8192	2737.8	138.1	113.5	188.3	162.2
	16384	8350.7	311.1	264.3	282.7	250.0
	32768	27847.4	815.5	823.4	610.1	563.4
	65536	112685.0	2782.6	2880.5	1819.0	1792.0
	131072	512504.0	10263.8	10613.5	6341.0	6335.0

Table 6.3: Execution time in seconds of running simulations with 131072 particles in parallel.

K	NoS 480	Streams 480	NoS K20c	Streams K20c
1	10694.2	9589.0	5858.0	6129.0
3	9720.0	9589.0	6047.0	6162.0
5	10359.4	10177.1	5856.6	6290.0
7	10455.6	10645.5	5882.0	6404.0
10	11045.6	10874.9	5868.0	6218.0
15	10711.8	11198.2	6248.0	6517.0
20	10263.8	10613.5	6341.0	6335.0

More precisely, the execution time of the parallel code is only doubling from when K is as small as one and as large as twenty for N equals 1024 for the Fermi card. Furthermore, Table 6.3 summarizes the execution times of the parallel code for a very large N of 131072, where columns represent CUDA code without streams (NoS) on the GTX 480 card and the Tesla K20c and code with streams on the GTX 480 and the Tesla K20c, respectively. From this table, we can see that the value of K doesn't have a large effect on the overall execution time for different values of K for the parallel algorithm. For example, it is always within a 10% range of difference between K=1 and K=20.

Running the parallel Nsight profiler [88], the average execution time of the *TryParticleTransferCB*() kernel call without using streams is 328μs. On the other hand, for the steams code, the first function that calculates Rosenbluth's weights is taking only 251μs on average. In addition to that, 125μs is the elapsed time for the kernel call that calculates the acceptance rate of the particle transfer. On average, the total elapsed time for a particle transfer move when using streams and a second kernel call to aggregate the results of all previous kernels is slightly smaller than that with no streams.

Table 6.4 shows the speedup of different code implementations for a large problem of 131072 particles. Columns represent speedup of the CUDA code with streams on the GTX 480 over the serial code, CUDA code with streams over without streams on the GTX 480 card, and the speedup of the stream code over without streams on Tesla K20c, respectively. The last two columns are the speedup of K20c over GTX 480.

As expected, the Kepler card provides better performance. Profiling the parallel code, we found that on average there are 11 and 16 active warps for the code with and without streams, respectively. Moreover, 480

Table 6.4: Speedup of running simulations with 131072 particles.

K	**Serial/S**	**NoS/S 480**	**NoS/S K20c**	**480/k20c NoS**	**480/k20c S**
1	54.9	1.1	0.96	1.8	1.6
3	48.5	1.0	0.98	1.6	1.6
5	44.6	1.0	0.93	1.8	1.6
7	46.7	1.0	0.92	1.8	1.7
10	45.6	1.0	0.94	1.9	1.7
15	40.1	1.0	0.96	1.7	1.7
20	48.3	1.0	1.00	1.6	1.7

cores are organized in 15 SMs of 32 cores each in the Fermi architecture. That is, half the number of SMs for the Kepler card, which in turn shows the two times speedup in Table 6.4 and is also plotted in Figure 6.6. In addition, the Kepler's Hyper-Q feature eliminates false data dependencies that could occur due to Fermi's single queue architecture.

Plotting the execution times for all algorithms and from Figure 6.5, we can see that:

- Despite the value of K, the execution time for the code with streams is the same for very large N.
- For small K less than 7, the Tesla K20c code shows no speedup for the streams code against the code without streams on the same card.
- For K larger than 5 and with between 16384 and 32768 particles in the system, the Tesla K20c code with streams is faster.
- Although we achieved two times speedup for executing the same parallel algorithm on the Tesla K20c, for both code with and without streams, these results came from running the code with no optimization techniques or customized tweaks done for the Kepler architecture. The reason for this performance improvement is because there are more resources available on the Kepler card. However, the number of streams in this case didn't affect the execution time of the code with streams significantly compared to that without streams. This is because in most cases the number of trials, denoted K, was less than the number of available streams, which is 16.

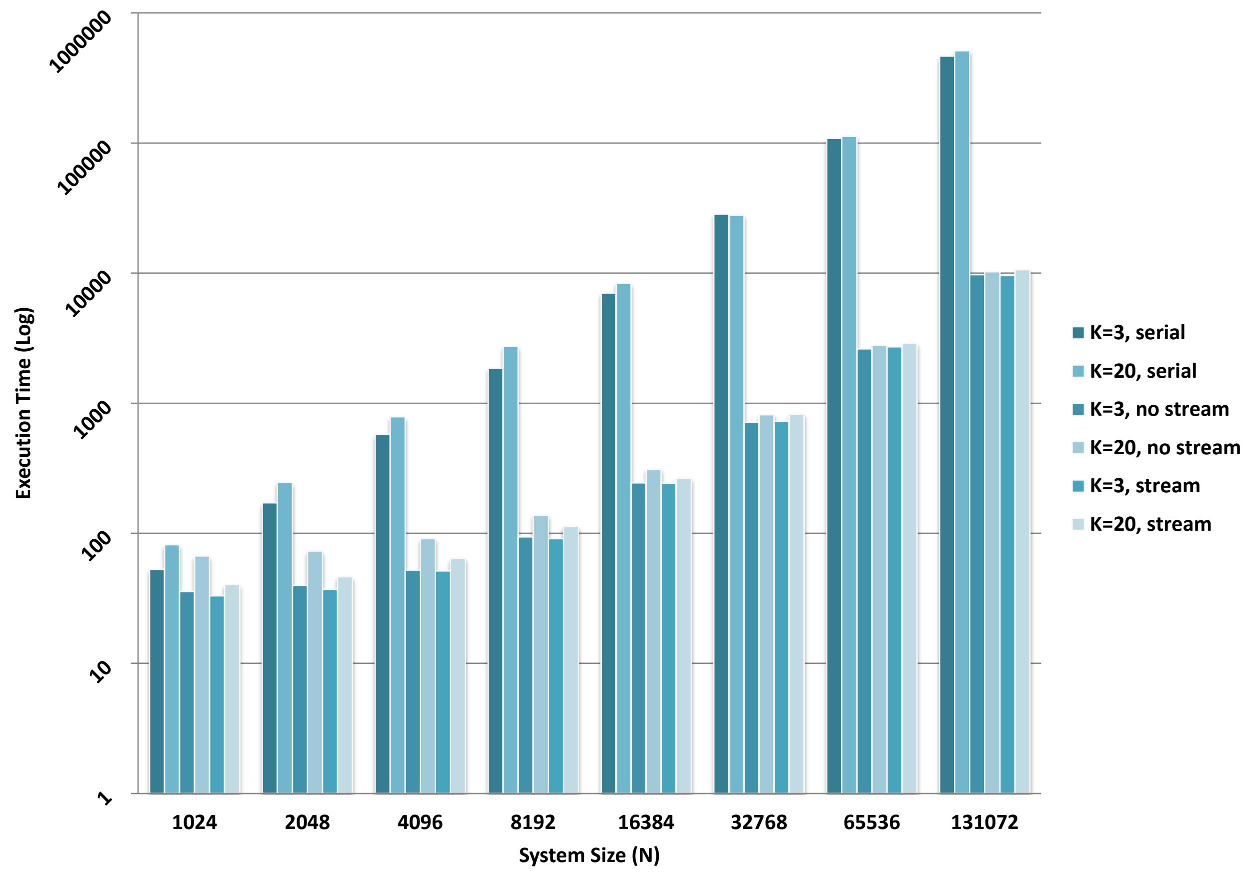


Figure 6.5: Execution times (Log Normalized) for K equals 3 and 20 with different algorithms executed on GeForece GTX 480.

- For K greater than 10 and N larger than 16384, the GTX 480 with no streams did better compared to the stream code.
- Code with streams is doing better for the GTX 480 for a relatively smaller size N.

The statistics in Table 6.5 are only for systems with 100% particle transfer move. For K = 1, the run time of the streams code exceeds the run time without streams, mainly because the code with streams has two kernel calls and uses more global memory than the other parallel code. On the other hand, for all values of K larger than one, the execution time for the stream code is larger than that for the code without streams. Although Figure 6.7 shows that as the system size increases, the execution time of all pieces of the code also increase, but the elapsed time of the code without streams is the highest.

Speedup results in Figure 6.8 shows different values of K for the stream code over the code without streams shows that a maximum speedup of about three times is observed when using the streams code and K

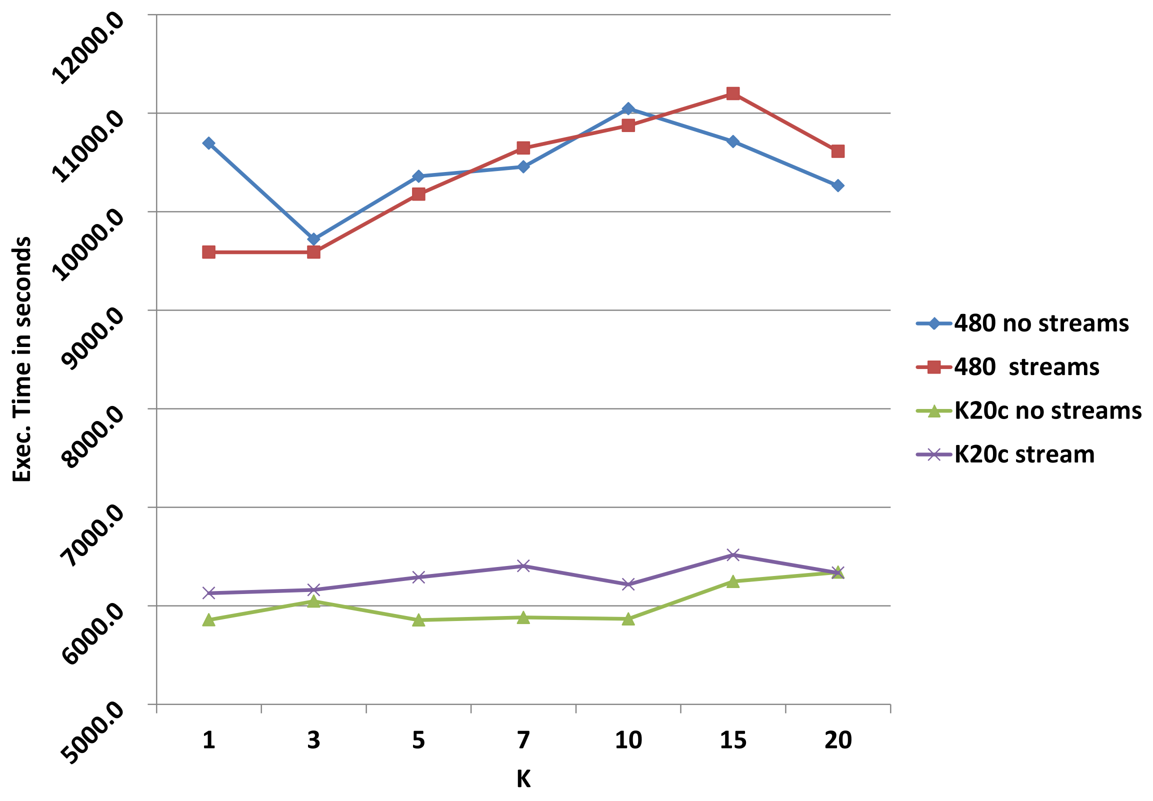


Figure 6.6: Execution times of CUDA code on GTX 480 and K20c for a system with 131072 particles.

Table 6.5: Execution times in seconds and speedup for CB for different number of CB trials (K) in parallel.

	N	**No Streams**	**Streams**	**Speedup**
K = 1	**1024**	49.8	47.8	0.96
	2048	51.0	48.7	0.97
	4096	52.6	51.4	0.96
	8192	54.7	53.77	0.94
	16384	68.8	64.4	1.02
	32768	91.6	92.3	0.93
	65536	128.6	138.6	0.88
	131072	204.4	244.1	0.82
K = 3	**1024**	92	61.7	1.49
	2048	93	63.3	1.47
	4096	101.3	66.6	1.52
	8192	103.4	80.5	1.28
	16384	150.2	107.3	1.40
	32768	212.5	169.6	1.25
	65536	318.5	278	1.15
	131072	531.3	508.3	1.05
K = 15	**1024**	320.8	118.4	2.71
	2048	331.9	126.1	2.63
	4096	356.7	143.5	2.49
	8192	388.8	190.2	2.04
	16384	614.9	285.3	2.16
	32768	916.7	515	1.78
	65536	1439.6	943	1.53
	131072	2454.7	1830.2	1.34
K = 20	**1024**	417.7	141	2.96
	2048	436.9	148.6	2.94
	4096	461	177.6	2.6
	8192	505.8	218.8	2.31
	16384	808.2	343.4	2.35
	32768	1208.1	619.7	1.95
	65536	1899.2	1159.5	1.64
	131072	3252.4	2214.8	1.47

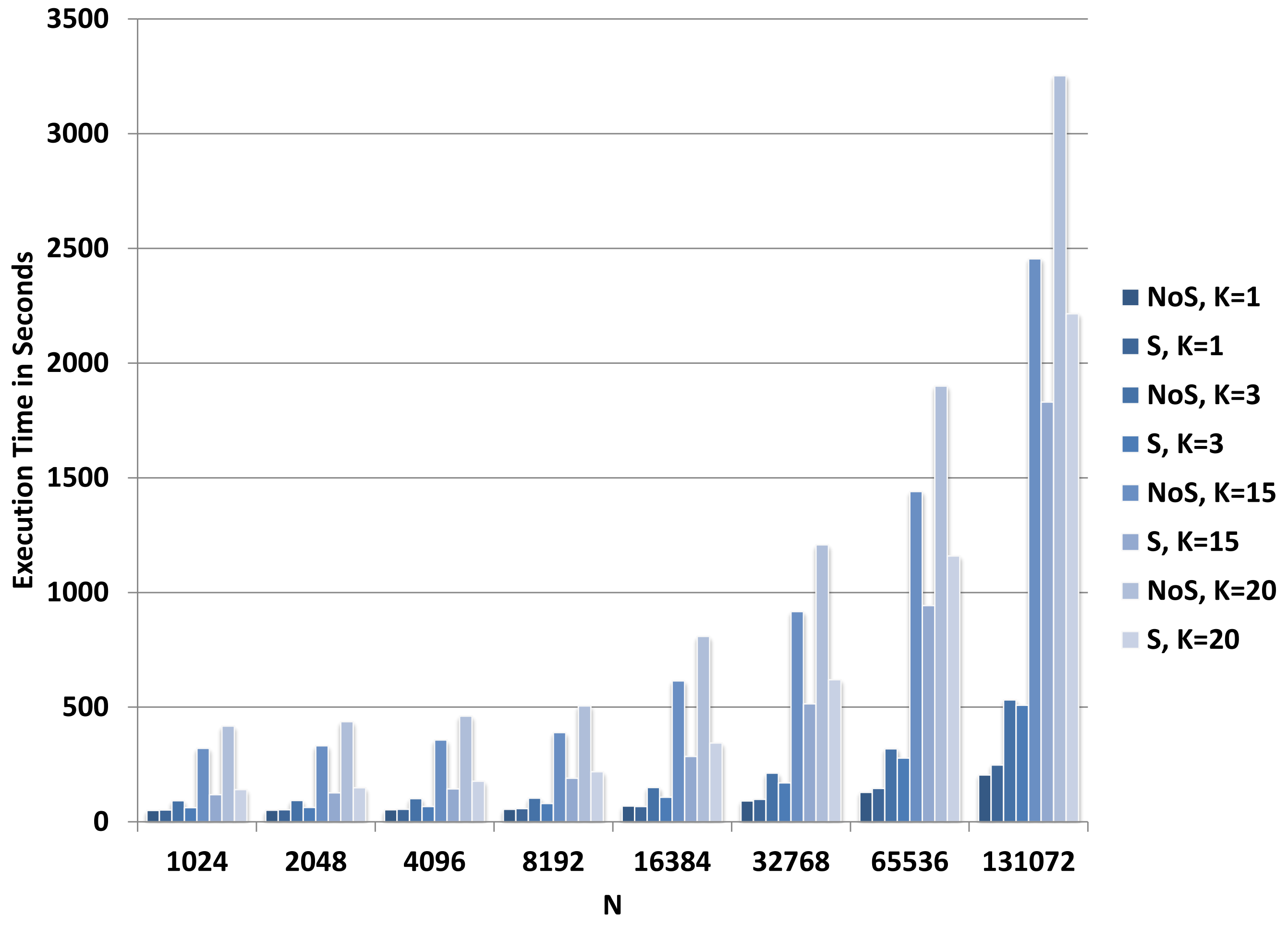


Figure 6.7: Execution time in seconds for parallel code with only particle transfer move for different values of K and N.

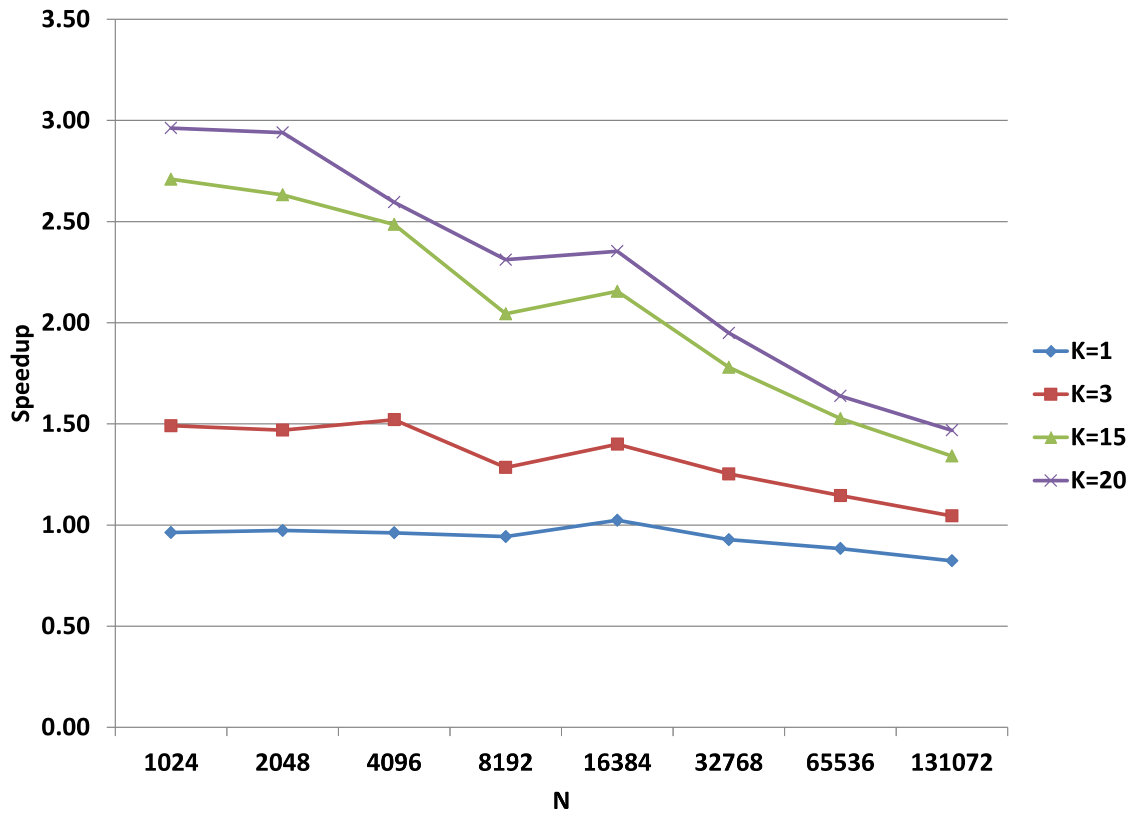


Figure 6.8: Speedup for parallel code with streams and no streams with only particle transfer move for different values of K and N.

= 20. But. the speedup is lower than that when K is smaller. Also, the highest speedup of the streams code is when K is the largest and the system size is relatively small. As we explained before, when the system size is relatively small, more streams can be scheduled to run concurrently. However, as the system size increases, there are more resources required to run a stream, so and not all streams can be scheduled to run in parallel.

CHAPTER 7 Conclusion and Future work

Given the serial nature of the Markov Chain Monte Carlo simulations, we present a very large scale parallel implementation fully ported the GPU for the canonical, Gibbs, and grand canonical ensemble methods with and without the configurational bias extension. We present a highly optimized serial code that runs on a single core CPU as a comparison base, a fast parallel implementation using CUDA, and a third parallel implementation using cell lists for the former two ensembles.

Research has been conducted to map the problem under study, and similar ones, to better harness the massively parallel nature of the GPU. The techniques that have been used throughout this dissertation can be used in developing code that enhance the overall GPU coding experience and show best performance practice. Other MC applications can benefit from the optimization techniques reported here, such as the problems of Bias Monte Carlo Methods in Environmental Engineering and Modeling of Biochemical Processes, as we have seen in § 1.8.

Optimization techniques have been applied to the original serial algorithm, such as the use of shared memory and load balancing among threads. In addition, synchronization techniques have been tested and several cases have been considered. All key components of the parallel algorithm have been tailored to the GPU for execution in a single kernel, which reduces overhead significantly. Moreover, high and low levels of parallelism are considered to adjust the serial algorithm to use new available features of the GPU. With the used parallel optimization techniques, doors are open to apply more algorithms and reduce the execution time per simulation step. For example, running more than one simulation at the same time is now possible after applying domain decomposition technique.

Different MC methods have different characteristics and simulation models. Each of which requires different optimization techniques and careful GPU resource management. Those methods have been moved to run on the GPU to reduce the execution time of the simulation. The evaluation of the parallel canonical algorithm on an affordable graphics processing unit shows a speedup of up to 15 times compared to the optimized serial implementation, and 2303 times speedup compared to *Towhee* for a small problem of size 4,096. Through the process of developing the parallel algorithm, an empirical optimization approach has been applied. This approach centers on selecting optimal block sizes for kernel invocation, which was 128 threads per block for large problem sizes. An implementation of cell list for this algorithm shows an additional factor of 11 times of speedup over the parallel implementation of the same algorithm. The reason for this decrease in execution time when using cell lists is due to the avoided calculations for particles that are out of cutoff range.

The cell list implementation does not offer any performance benefits until we have more than 4,096 particles. Traditionally, simulations using serial codes have not used more than about 2000 particles because of the long running time, so a cell list implementation has not been considered efficient.

Due to the lack of current serial cell list implementation in our code, and for a performance comparison, we report speedup results of our GPU algorithm compared to a single core CPU code as evidence of performance benefits of the CUDA code with no cell list implementation. Also, we compare the CUDA code with an efficient cell list implementation. Different cell sizes have been studied and performance results has been explained. Moreover, different algorithms have been developed for implementing the cell list and the performance results have been discussed.

In this work, we developed a MC code for the Gibbs ensemble simulation of Lennard-Jones atoms that utilizes graphics processors (GPUs) achieving a factor of over 45 times speedup in comparison to a single core serial code. This will open the door for new research since the Gibbs ensemble is the best method for determining the phase diagram of fluid mixture.

An extension to the Gibbs ensemble method has been implemented, configurational bias. In this extension, the acceptance rate of moving particles from one box to the other is enhanced allowing the simulation to reach equilibrium faster by attempting multiple similar moves at the same time. However, this creates a sampling bias that needs to be removed by applying extra calculations. Multiple attempts can be conducted in parallel, which minimizes the execution time significantly. The speedup gain of using the GPU to execute this algorithm shows a factor of up to 50 times speedup over the efficient single core implementation and a slightly better execution time over that when using GPU streams.

The grand canonical algorithm has been implemented and optimized to run on the GPU as well. However, the speedup achieved with this implementation was less than the Gibbs algorithms and more than that for the canonical one. This is mainly because most moves in the grand canonical algorithm are displacement moves, which is less computation intensive than the Gibbs ensemble method. On the other hand, the overhead of maintaining a dynamic box is more than that for the canonical ensemble. That being said, the parallel grand canonical algorithm shows about 16 times speedup over the single core implementation and an additional factor of 8 times speedup when using the cell list implementation, adding up to 128 times speedup over the single core code.

While our implementation features only the simple Lennard-Jones potential, it is trivial to replace this potential by other pair-potentials such as the coulombic interactions. Moreover, it is possible to implement many other techniques, such as Ewald sum methods and chain molecules.

REFERENCES

[1] Hoomd-blue web page. `http://codeblue.umich.edu/hoomd-blue`, Nov 2012.

[2] D.J Adams. The implementation of fluid phase monte carlo on the dap. *Journal of Computational Physics*, 75(1):138 – 150, 1988.

[3] E. Alerstam, T. Svensson, and S. Andersson-Engels. Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration. *J Biomed Opt*, 13(6):060504, 2008.

[4] Amos G. Anderson, William A. Goddard III, and Peter Schröder. Quantum Monte Carlo on graphical processing units. *Computer Physics Communications*, 177(3):298–306, 2007.

[5] J. Anderson, C. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

[6] U. Assarsson and E. Sintorn. Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008.

[7] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 94–103, New York, NY, USA, 2010. ACM.

[8] I. Beichl and F. Sullivan. The metropolis algorithm. *Computing in Science Engineering*, 2(1):65 –69, jan.-feb. 2000.

[9] Kurt Binder. *Monte Carlo and molecular dynamics simulations in polymer science*. Oxford University Press, USA, 1995.

[10] G.D. Birkhoff. Proof of the ergodic theorem. *Proceedings of the National Academy of Sciences of the United States of America*, 17(12):656, 1931.

[11] B. Block, P. Virnau, and T. Preis. Multi-GPU accelerated multi-spin Monte Carlo simulations of the 2D Ising model. *Computer Physics Communications*, 181(9):1549–1556, 2010.

[12] U. D. Bordoloi and S. Chakraborty. GPU-based Acceleration of System-level Design Tasks. *International Journal of Parallel Programming*, 38(3-4):225–253, 2010.

[13] Randy P. Broussard and Robert W. Ives. Using a commercial graphical processing unit and the cuda programming language to accelerate scientific image processing applications. volume 7872, page 787202. SPIE, 2011.

[14] W.M. Brown, S. Hampton, P. Agarwal, P. Wang, P. Crozier, and S. Plimpton. Porting lammps to gpus. Technical report, Sandia National Laboratories, 2010.

[15] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *Acm Transactions on Graphics*, 23(3):777–786, 2004.

[16] J. Buckner, J. Wilson, M. Seligman, B. Athey, S. Watson, and F. Meng. The gputools package enables GPU computing in R. *Bioinformatics*, 26(1):134–5, 2010.

[17] A. Buluc, J. R. Gilbert, and C. Budak. Solving path problems on the GPU. *Parallel Computing*, 36(5-6):241–253, 2010.

[18] Tahir Cagin and B. Montgomery Pettitt. Grand molecular dynamics: A method for open systems. *Molecular Simulation*, 6(1-3):5–26, 1991.

[19] Nuno Cardoso and Pedro Bicudo. Su (2) lattice gauge theory simulations on fermi gpus. *Journal of Computational Physics*, 230(10):3998 – 4010, 2011.

[20] D. Cha, S. Son, and I. Ihm. GPU-Assisted High Quality Particle Rendering. *Computer Graphics Forum*, 28(4):1247–1255, 2009.

[21] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel Distrib. Comput.*, 68:1370–1380, October 2008.

[23] P.G. Ciarlet, C. Le Bris, and J.L. Lions. *Handbook of Numerical Analysis: Special volume: Computational chemistry*, volume 10. North Holland, 2003.

[24] Kevin B. Daly, Jay B. Benziger, Pablo G. Debenedetti, and Athanassios Z. Panagiotopoulos. Massively parallel chemical potential calculation on graphics processing units. *Computer Physics Communications*, 183(10):2054 – 2062, 2012.

[25] D. De 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*, 52(3):116–122, 2010.

[26] M. de Greef, J. Crezee, J. C. van Eijk, R. Pool, and A. Bel. Accelerated ray tracing for radiotherapy dose calculations on a gpu. *Medical Physics*, 36(9):4095–4102, 2009.

[27] Lorenzo Dematta and Davide Prandi. GPU computing for systems biology. *Briefings in Bioinformatics*, 11(3):323–333, 2010.

[28] Peter Deuflhard. *Computational Molecular Dynamics: Challenges, Methods, Ideas: Proceedings of the 2nd International Symposium on Algorithms for Macromolecular Modellin*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 4th edition, 1999.

[29] F. Devillard, S. Gobron, and B. Heit. Retina simulation using cellular automata and GPU programming. *Machine Vision and Applications*, 18(6):331–342, 2007.

[30] Tina Duren, Youn-Sang Bae, and Randall Q. Snurr. Using molecular simulation to characterise metal-organic frameworks for adsorption applications. *Chem. Soc. Rev.*, 38:1237–1247, 2009.

[31] Michael Engel, Joshua A. Anderson, Sharon C. Glotzer, Masaharu Isobe, Etienne P. Bernard, and Werner Krauth. Hard-disk equation of state: First-order liquid-hexatic transition in two dimensions with three simulation methods. *Phys. Rev. E*, 87:042134, Apr 2013.

[32] K. Esselink, L. D. J. C. Loyens, and B. Smit. Parallel monte carlo simulations. *Phys. Rev. E*, 51:1560–1568, Feb 1995.

[33] Kamel Rushaidat Yunazhi Li Loren Schwiebert Jeffery Potoff Eyad Hailat, Jason Mick. GPU optimized Monte Carlo. `http://site:gomc-website`, 2013.

[34] Q. Fang and D. A. Boas. Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units. *Opt Express*, 17(22):20178–90, 2009.

[35] Qianqian Fang and David A. Boas. GPU Accelerated Monte Carlo Simulation for 3-D Photon Migration. In *Biomedical Optics*, page BME5. Optical Society of America, 2010.

[36] D. Frenkel and B. Smit. Unexpected length dependence of the solubility of chain molecules. *Molecular Physics*, 75:983–988, 1992.

[37] D. Frenkel and B. Smit. *Understanding molecular simulation: from algorithms to applications*, volume 1. Academic Pr, 2002.

[38] Daan Frenkel. Speed-up of monte carlo simulations by sampling of rejected states. *Proceedings of the National Academy of Sciences of the United States of America*, 101(51):17571–17575, 2004.

[39] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation (Second Edition)*. Academic Press, San Diego, second edition edition, 2002.

[40] A. Frezzotti, G.P. Ghiroldi, and L. Gibelli. Direct solution of the boltzmann equation for a binary mixture on gpus. In *American Institute of Physics Conference Series*, volume 1333, pages 884–889, 2011.

[41] Mark S. Friedrichs, Peter Eastman, Vishal Vaidyanathan, Mike Houston, Scott Legrand, Adam L. Beberg, Daniel L. Ensign, Christopher M. Bruns, and Vijay S. Pande. Accelerating molecular dynamic simulation on graphics processing units. *Journal of Computational Chemistry*, 30(6):864–872, 2009.

[42] Y. Frishman and A. Tal. Multi-level graph layout on the GPU. *Ieee Transactions on Visualization and Computer Graphics*, 13(6):1310–1317, 2007.

[43] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *Ieee Micro*, 28(4):13–27, 2008.

[44] David Geer. Taking the Graphics Processor beyond Graphics. *Computer*, 38:14–16, September 2005.

[45] S. Gorbunov, D. Rohr, K. Aamodt, T. Alt, H. Appelshauser, A. Arend, M. Bach, B. Becker, S. Bottger, T. Breitner, H. Busching, S. Chattopadhyay, J. Cleymans, C. Cicalo, I. Das, O. Djuvsland, H. Engel, H. A. Erdal, R. Fearick, O. S. Haaland, P. T. Hille, S. Kalcher, K. Kanaki, U. W. Kebschull, I. Kisel, M. Kretz, C. Lara, S. Lindal, V. Lindenstruth, A. A. Masoodi, G. Ovrebekk, R. Panse, J. Peschek, M. Ploskon, T. Pocheptsov, D. Ram, T. Rascanu, M. Richter, D. Rohrich, F. Ronchetti, B. Skaali, O. Smorholm, C. Stokkevag, T. M. Steinbeck, A. Szostak, J. Thader, T. Tveter, K. Ullaland, Z. Vilakazi, R. Weis, Z. B. Yin, P. Zelnicek, and Alice Collaboration. ALICE HLT High Speed Tracking on GPU. *Ieee Transactions on Nuclear Science*, 58(4):1845–1851, 2011.

[46] Shen Guobin, Gao Guang-Ping, Li Shipeng, Shum Heung-Yeung, and Zhang Ya-Qin. Accelerate video decoding with generic GPU. *Ieee Transactions on Circuits and Systems for Video Technology*, 15(5):685–693, 2005.

[47] Jonathan Harris and Stuart A. Rice. A lattice model of a supported monolayer of amphiphile molecules: Monte carlo simulations. *The Journal of Chemical Physics*, 88(2):1298–1306, 1988.

[48] Mark Harris. *Optimizing Parallel Reduction in CUDA*. NVIDIA Developer Technology, 2.3 edition, 2008.

[49] A. Heimlich, A. C. A. Mol, and C. M. N. A. Pereira. GPU-based Monte Carlo simulation in neutron transport and finite differences heat equation evaluation. *Progress in Nuclear Energy*, 53(2):229–239, 2011.

[50] A. Herrera. GPU computing uncovered. *Computer Graphics World*, 30(6):34–, 2007. 183DE Times Cited:0 Cited References Count:0.

[51] José R. Herrero, Enrique S. Quintana-ortí, and Robert Strzodka. Special Issue: GPU computing. *Concurrency and Computation: Practice and Experience*, 23(7):667–668, 2011.

[52] Q. M. Hou, K. Zhou, and B. N. Guo. BSGP: Bulk-synchronous GPU programming. *Acm Transactions on Graphics*, 27(3), 2008.

[53] Q. M. Hou, K. Zhou, and B. N. Guo. Debugging GPU Stream Programs Through Automatic Dataflow Recording and Visualization. *Acm Transactions on Graphics*, 28(5), 2009.

[54] Vassil Hristov. Performance Evaluation of Query Pro cessing Algorithms on GP GPUs. Master of science, University of Edinburgh, 2010.

[55] NVIDIA Inc. NVIDIA Developer Zone. `http://developer.nvidia.com`, Jan 2013.

[56] Zeena K. Issa, Charles W. Manke, Bhanu P. Jena, and Jeffrey J. Potoff. Ca2+ bridging of apposed phospholipid bilayers. *The Journal of Physical Chemistry B*, 114(41):13249–13254, 2010.

[57] Prateek K. Jha, Rastko Sknepnek, Guillermo Iva'n Guerrero-Garci'a, and Monica Olvera de la Cruz. A graphics processing unit implementation of coulomb interaction in molecular dynamics. *Journal of Chemical Theory and Computation*, 6(10):3058–3065, 2010.

[58] X. Jia, X. Gu, J. Sempau, D. Choi, A. Majumdar, and S. B. Jiang. Development of a GPU-based Monte Carlo dose calculation code for coupled electron-photon transport. *Phys Med Biol*, 55(11):3077–86, 2010.

[59] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *CoRR*, abs/1005.2581, 2010.

[60] Emmett Kilgariff and Randima Fernando. The GeForce 6 series GPU architecture. In *ACM SIGGRAPH 2005 Courses*, page 29. ACM, 2005.

[61] Jihan Kim, Jocelyn M. Rodgers, Manuel Athnes, and Berend Smit. Molecular monte carlo simulations using graphics processing units: To waste recycle or not? *Journal of Chemical Theory and Computation*, 7(10):3208–3222, 2011.

[62] C. Kolb and M. Pharr. Options Pricing on the GPU. *GPU Gems*, 2:719–731, 2005.

[63] Jingfei Kong, Martin Dimitrov, Yi Yang, Janaka Liyanage, Lin Cao, Jacob Staples, Mike Mantor, and Huiyang Zhou. Accelerating matlab image processing toolbox functions on gpus. In *Proceedings*

of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU '10, pages 75–85, New York, NY, USA, 2010. ACM.

[64] J. Kruger and R. Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum*, 24(3):685–693, 2005.

[65] Fang-An Kuo, Matthew R. Smith, Chih-Wei Hsieh, Chau-Yi Chou, and Jong-Shinn Wu. GPU acceleration for general conservation equations and its application to several engineering problems. *Computers & Fluids*, 45(1):147–154, 2011.

[66] Frédéric Kuznik, Christian Obrecht, Gilles Rusaouen, and Jean-Jacques Roux. LBM based flow simulation using GPU computing processor. *Computers & Mathematics with Applications*, 59(7):2380–2392, 2010.

[67] Sandia National Labs. Lammps: A fix to perform grand canonical monte carlo. `http://lammps.sandia.gov/doc/fix_gcmc.html`, July 2013.

[68] Aaron E Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D Owens. Glift: Generic, efficient, random-access gpu data structures. *Acm Transactions on Graphics*, 25(1):60–99, 2006.

[69] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Ieee Micro*, 28(2):39–55, 2008.

[70] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig. Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA. *Computer physics communications*, 179(9):634–641, 2008.

[71] L.D.J.C. Loyens, B. Smit, and K. Esselink. Parallel gibbs-ensemble simulations. *Molecular Physics*, 86(2):171–183, 1995.

[72] David Luebke and Greg Humphreys. How GPUs Work. *Computer*, 40(2):96–100, feb. 2007.

[73] V. I. Manousiouthakis and M. W. Deem. Strict detailed balance is unnecessary in Monte Carlo simulation. *jcp*, 110:2753–2756, February 1999.

[74] Marcus G. Martin and Mary J. Biddy. Monte Carlo molecular simulation predictions for the heat of vaporization of acetone and butyramide. *Fluid Phase Equilibria*, 236(1-2):53–57, 2005.

[75] Marcus G. Martin and J. Ilja Siepmann. Novel configurational-bias monte carlo method for branched molecules. transferable potentials for phase equilibria. 2. united-atom description of branched alkanes. *The Journal of Physical Chemistry B*, 103(21):4508–4517, 1999.

[76] MG Martin, B. Chen, CD Wick, JJ Potoff, J.M. Stubbs, and JI Siepmann. Mcccs towhee. `http://towhee.sourceforge.net/`, Nov 2012.

[77] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998.

[78] Simon C. Mcgrother and Keith E. Gubbins. Constant pressure gibbs ensemble monte carlo simulations of adsorption into narrow pores. *Molecular Physics*, 97(8):955–965, 1999.

[79] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21:1087, 1953.

[80] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[81] Charles J. Mode. *Applications of Monte Carlo Method on Science and Engineering*, volume 1. InTech, 2011.

[82] A. Munshi et al. The opencl specification. *Khronos OpenCL Working Group*, 2011.

[83] J. Nickolls and W. J. Dally. The Gpu Computing Era. *Ieee Micro*, 30(2):56–69, 2010.

[84] NVIDIA. *CUDA C Programmin Guide 4.0*, 4.0 edition, Sep 2011.

[85] NVIDIA. *CUDA C Programming Guide 4.2*, 4.2 edition, Feb 2012.

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

[87] NVIDIA. *CUDA C Programming Guide 5.0*, 5.0 edition, Feb 2013.

[88] NVIDIA. NVIDIA Nsight Visual Studio Edition. `https://developer.nvidia.com/nvidia-nsight-visual-studio-edition`, 2013.

[89] NVIDIA. *TESLA GPU ACCELERATORS FOR SERVERS*, 5.0 edition, May 2013.

[90] The National Institute of Standards and Technology (NIST). Benchmark results for lennard-jones fluid-nvt monte carlo results at both liquid- and vapor-like densities. `http://cstl.nist.gov/srs/LJ_PURE/mc.htm`, 2012.

[91] The National Institute of Standards and Technology (NIST). Simpatico - simulation package for polymer and molecular liquids. `http://research.cems.umn.edu/morse/code/simpatico/old/index.html`, July 2013.

[92] C. J. O'Keeffe and G. Orkoulas. Parallel canonical monte carlo simulations through sequential updating of particles. *The Journal of Chemical Physics*, 130(13):134109, 2009.

[93] A. Z. Panagiotopoulos. Direct determination of phase coexistence properties of fluids by monte carlo simulation in a new ensemble. *Molecular Physics*, 61:813–826, 1987.

[94] A. Z. Panagiotopoulos and J. J. Potoff. Critical point and phase behavior of the pure fluid and a Lennard-Jones mixture. *Journal of Chemical Physics*, 109(24):10914–10920, 1998.

[95] Vijay S. Pande, Ian Baker, Jarrod Chapman, Sidney P. Elmer, Siraj Khaliq, Stefan M. Larson, Young Min Rhee, Michael R. Shirts, Christopher D. Snow, Eric J. Sorin, and Bojan Zagrovic. Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. *Biopolymers*, 68(1):91–109, 2003.

[96] Aphrodite Papadopoulou, Ezra D. Becker, Mark Lupkowski, and Frank van Swol. Molecular dynamics and monte carlo simulations in the grand canonical ensemble: Local versus global control. *The Journal of Chemical Physics*, 98(6):4897–4908, 1993.

[97] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kal, and Klaus Schulten. Scalable molecular dynamics with namd. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[98] V. Podlozhnyuk and M. Harris. Monte carlo option pricing. *nVidia Corporation Tutorial*, 2008.

[99] 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*, 113(44):14725–14731, 2009.

[100] Jeffrey J. Potoff and Athanassios Z. Panagiotopoulos. Critical point and phase behavior of the pure fluid and a lennard-jones mixture. *The Journal of Chemical Physics*, 109(24):10914–10920, 1998.

[101] Department of Chemical Engineering Potoff Research Group and Wayne State University Materials Science. Trappe-related models and applications. http://potoff1.eng.wayne.edu/trappe-force-field-publications.

[102] T. Preis, P. Virnau, W. Paul, and J.J. Schneider. Gpu accelerated monte carlo simulation of the 2d and 3d ising model. *Journal of Computational Physics*, 228(12):4468–4477, 2009.

[103] O. K. Rice. On the statistical mechanics of liquids, and the gas of hard elastic spheres. *The Journal of Chemical Physics*, 12(1):1–18, 1944.

[104] Marshall N. Rosenbluth and Arianna W. Rosenbluth. Monte carlo calculation of the average extension of molecular chains. *The Journal of Chemical Physics*, 23(2):356–359, 1955.

[105] M.N. Rosenbluth and A.W. Rosenbluth. Further results on monte carlo equations of state. *The Journal of Chemical Physics*, 22:881, 1954.

[106] D. Salomon. Graphics Devices. *The Computer Graphics Manual*, pages 1203–1285, 2011.

[107] Romelia Salomon-Ferrer, David A. Case, and Ross C. Walker. An overview of the amber biomolecular simulation package. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, pages n/a–n/a, 2012.

[108] Shubhabrata Sengupta, Mark Harris, Michael Garland, and John D Owens. Efficient parallel scan algorithms for many-core gpus. *Scientific Computing with Multicore and Accelerators*, 2011.

[109] J. I. Siepmann. A method for the direct calculation of chemical potentials for dense chain systems. *Molecular Physics*, 70:1145–1158, August 1990.

[110] Jrn Ilja Siepmann and Daan Frenkel. Configurational bias monte carlo: a new sampling scheme for flexible chains. *Molecular Physics*, 75(1):59–70, 1992.

[111] 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. *J Comput Chem*, 28(16):2618–40, 2007.

[112] 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*, 28(16):2618–2640, 2007.

[113] S.S. Stone, J.P. Haldar, S.C. Tsao, W.-m.W. Hwu, B.P. Sutton, and Z.-P. Liang. Accelerating advanced MRI reconstructions on GPUs. *Journal of Parallel and Distributed Computing*, 68(10):1307–1318, 2008.

[114] S.S. Stone, H. Yi, J. Haldar, W. Hwu, B. Sutton, and Z.P. Liang. How GPUs can improve the quality of magnetic resonance imaging. In *The First Workshop on General Purpose Processing on Graphics Processing Units*. Citeseer, 2007.

[115] Magnus Strengert, Thomas Klein, and Thomas Ertl. A hardware-aware debugger for the opengl shading language. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 81–88, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association.

[116] Jeff Stuart, Michael Cox, and John Owens. Gpu-to-cpu callbacks. In Mario Guarracino, Fredric Vivien, Jesper Traff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knapfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 365–372. Springer Berlin / Heidelberg, 2011.

[117] M. P. H. Stumpf, Y. X. Zhou, J. L. Liepe, X. Sheng, and C. Barnes. GPU accelerated biochemical network simulation. *Bioinformatics*, 27(6):874–876, 2011.

[118] A. Sunarso, T. Tsuji, and S. Chono. GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows. *Journal of Computational Physics*, 229(15):5486–5497, 2010.

[119] L. Szirmay-Kalos, T. Umenhoffer, G. Patow, L. Szecsi, and M. Sbert. Specular Effects on the GPU: State of the Art. *Computer Graphics Forum*, 28(6):1586–1617, 2009.

[120] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, and S. Miki. *The OpenCL Programming Book*. Fixstars Corporation, 2009.

[121] J.A. van Meel, A. Arnold, D. Frenkel, S.F.P. Zwart, and R.G. Belleman. Harvesting graphics power for md simulations. *Molecular Simulation*, 34(3):259–266, 2008.

[122] Vincent A. Voelz, Gregory R. Bowman, Kyle Beauchamp, and Vijay S. Pande. Molecular Simulation of ab Initio Protein Folding for a Millisecond Folder NTL9(1-39). *Journal of the American Chemical Society*, 132(5):1526–1528, 2010. PMID: 20070076.

[123] W.H. Wen-mei. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[124] Shengqi Yang, Ronghui Cheng, and Li Zou. Case study of programmable video post processing: Cuda-based novel edge directed video scaling. In *Multimedia and Expo (ICME), 2010 IEEE International Conference on*, pages 884 –889, july 2010.

[125] Stephen J. Zara and David Nicholson. Grand canonical ensemble monte carlo simulation on a transputer array. *Molecular Simulation*, 5(3-4):245–261, 1990.

ABSTRACT

ADVANCED OPTIMIZATION TECHNIQUES FOR MONTE CARLO SIMULATION ON GRAPHICS PROCESSING UNITS

by

EYAD HAILAT

August 2013

Advisor: Dr. Loren Schwiebert

Major: Computer Science

Degree: Doctor of Philosophy

The objective of this work is to design and implement a self-adaptive parallel GPU optimized Monte Carlo algorithm for the simulation of adsorption in porous materials. We focus on Nvidia's GPUs and CUDA's Fermi architecture specifically. The resulting package supports the different ensemble methods for the Monte Carlo simulation, which will allow for the simulation of multi-component adsorption in porous solids. Such an algorithm will have broad applications to the development of novel porous materials for the sequestration of CO_2 and the filtration of toxic industrial chemicals.

The primary objective of this work is the release of a massively parallel open source Monte Carlo simulation engine implemented using GPUs, called **GOMC**. The code will utilize the canonical ensemble, and the Gibbs ensemble method, which will allow for the simulation of multiple phenomena, including liquid-vapor phase coexistence, and single and multi-component adsorption in porous materials. In addition, the grand canonical ensemble and the configurational-bias algorithms have been implemented so that polymeric materials and small proteins may be simulated.

This simulation engine is the only open source GPU optimized Monte Carlo code available for the generalized simulation of adsorption and phase equilibria on a very large scale. As a result of conducting many optimization techniques and allowing the system to adjust for the change of simulation state, the

original MC algorithm has been rewritten based on an existing serial algorithm to suit the massive parallel devices resulting in reductions in computational time. This large time reduction allow for the simulation of significantly larger systems for longer timescales than is currently possible with existing implementations.

Results of the extensive research and applying device specific optimizations resulted in significant speedup. First, for the NVT method, a fully optimized serial algorithm has been implemented and the performance results has been compared to Towhee. A speedup of about 438 times has been achieved for a relatively small size problem of 4096 particles. In addition, two algorithms to run on the GPU with and without cell list structure have been implemented. The total speedup of the parallel code with cell list over the serial code was more than 160$\times$ faster. Moreover, for the grand canonical ensemble, a serial and two parallel algorithms have been developed. The simulation box in this method can be resized, which added a change to the algorithm that needed to adapt with the box size and adjust itself. The performance of running the CUDA code with cell list versus the serial code that doesn't have a cell list structure is a factor of 130 times faster.

More MC ensembles have been transferred to the GPU. The Gibbs ensemble method has two simulation boxes and three types of moves. This method has been studied carefully and the GPU algorithm has been implemented to port the computation intensive functions to the GPU. The performance of the GPU code was about 50$\times$ faster than the serial code. Finally, an extension of the Gibbs method has been implemented on the GPU. The particle transfer from one box to the other is the affected move type by this extension. CUDA streams are used to parallelize K trials for this method. A factor of three times speedup for the particle transfer move has been achieved for the best case. However, due to the low execution rate of the particle transfer move, just 10% of the total moves, the speedup has minimal effect on overall execution time of the simulation. Furthermore, a different run with all move types on Kepler K20c card has been executed, and a factor of 2 times speedup has been reported over the CUDA code on the GeForce GTX 480 card.

The main contribution of this work to society is when the above implementations become open source to the public through http://gomc.eng.wayne.edu. Also, other researchers can take advantage of the lessons learned with advanced optimizations and self-adapting mechanisms specific to the GPU. On the application level, the current code can be used by the chemical engineering community to explore accurate and affordable simulations that were not possible before.

AUTOBIOGRAPHICAL STATEMENT

Eyad Hailat earned a Master's and a Bachelor's degrees in Computer Science and Information Technology from Yarmouk University, in Jordan in 2002 and 2004, respectively. He is a graduate student member of the IEEE Communications, and Association for Computing Machinery chapter at Wayne State University. From 2006 to 2012, Mr. Hailat worked as a Teaching Assistant in the department of Computer Science at Wayne State University as well as instructor for a variety of undergraduate level classes. After that, and until he joined Effyis, Inc. as a data scientist in May 2013, he was a research assistant at the department of Computer Science at Wayne State University..

During the journey to earn his Ph.D., Mr. Hailat served as a reviewer for several peer-reviewed journals and conferences. Moreover, he was honored to accept the following awards:

- First place award. The 4th annual graduate exhibition, Wayne State University, presented March 19, 2013. A hundred participants from all majors in school.
- Graduate Research Award (NSF), 2012/2013.
- Graduate Research Award (REP - Wayne State University), 2011/2012.
- Outstanding Teaching of the year Award, Wayne State University, 2009.
- Graduate Professional Scholarship, Wayne State University, 2007. The highest prestigious graduate scholarship at Wayne State University.
- Graduate Teaching Award, Wayne State University, 2006-2010.
- Graduate Teaching Award, Yarmouk University, 2001.