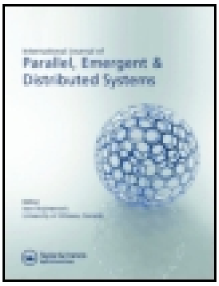

International Journal of Parallel, Emergent and Distributed Systems
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/gpaa20

Parallel Monte Carlo simulation in the canonical ensemble on the graphics processing unit

Eyad Hailat[a], Vincent Russo[a], Kamel Rushaidat[a], Jason Mick[b], Loren Schwiebert[a] & Jeffrey Potoff[b]

[a] Department of Computer Science, Wayne State University, Detroit, MI48202, USA

[b] Department of Chemical Engineering and Materials Science, Wayne State University, Detroit, MI48202, USA

Published online: 02 Oct 2013.

To cite this article: Eyad Hailat, Vincent Russo, Kamel Rushaidat, Jason Mick, Loren Schwiebert & Jeffrey Potoff , International Journal of Parallel, Emergent and Distributed Systems (2013): Parallel Monte Carlo simulation in the canonical ensemble on the graphics processing unit, International Journal of Parallel, Emergent and Distributed Systems, DOI: 10.1080/17445760.2013.833617

To link to this article: http://dx.doi.org/10.1080/17445760.2013.833617

International Journal of Parallel, Emergent and Distributed Systems, 2013
http://dx.doi.org/10.1080/17445760.2013.833617

Parallel Monte Carlo simulation in the canonical ensemble on the graphics processing unit

Eyad Hailat[a]*, Vincent Russo[a], Kamel Rushaidat[a], Jason Mick[b], Loren Schwiebert[a] and Jeffrey Potoff[b]

[a]*Department of Computer Science, Wayne State University, Detroit, MI 48202, USA;* [b]*Department of Chemical Engineering and Materials Science, Wayne State University, Detroit, MI 48202, USA*

(*Received 10 February 2013; accepted 6 August 2013*)

Graphics processing units (GPUs) offer parallel computing power that usually requires a cluster of networked computers or a supercomputer to accomplish. While writing kernel code is fairly straightforward, achieving efficiency and performance requires very careful optimisation decisions and changes to the original serial algorithm. We introduce a parallel canonical ensemble Monte Carlo (MC) simulation that runs entirely on the GPU. In this paper, we describe two MC simulation codes of Lennard-Jones particles in the canonical ensemble, a single CPU core and a parallel GPU implementations. Using Compute Unified Device Architecture, the parallel implementation enables the simulation of systems containing over 200,000 particles in a reasonable amount of time, which allows researchers to obtain more accurate simulation results. A remapping algorithm is introduced to balance the load of the device resources and demonstrate by experimental results that the efficiency of this algorithm is bounded by available GPU resource. Our parallel implementation achieves an improvement of up to 15 times on a commodity GPU over our efficient single core implementation for a system consisting of 256k particles, with the speedup increasing with the problem size. Furthermore, we describe our methods and strategies for optimising our implementation in detail.

Keywords: graphics processing unit; Compute Unified Device Architecture; high-performance computing; Monte Carlo simulations; canonical thermodynamic ensemble; Lennard-Jones potential

1. Introduction

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 optimised algorithms and code to utilise these devices remains time-consuming and intensive. In this paper, we describe the development of an efficient GPU implementation for the Monte Carlo (MC) simulation of molecular systems. The programming abstraction utilised in this paper is the Compute Unified Device Architecture (CUDA™) application programming interface [30]. This architecture supports the parallel programming model and the instruction set that allows substantial speedup over general purpose CPUs for data parallel applications. Moreover, CUDA allows the developer to use their familiarity with languages such as C, C++ or Fortran within the framework of the programming environment. This reduces the learning curve for developers and promotes

*Corresponding author. Email: eyad@wayne.edu rapid application development. The result is that more applications are being re-implemented for execution on the GPU every day and computer simulation is one such key application.

Potential functions have long been used in physical simulations to describe the collective or local behaviour 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 laboratory experiments to get information on the liquid or gas states of chemical compounds and mixtures [11]. Two approaches have been of particular interest to a number of researchers, MC and molecular dynamics (MD) simulations [6,14]. 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 MC simulation include the following:

- *Prediction of physical properties and phase behaviour*. 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.
- *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 [13].
- *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 [17]. 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.
- *The use of nanoparticles to stabilise 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.

MC 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 millions of iterations required for the MC simulation to converge to a solution.

An optimised sequential version of the canonical MC algorithm was written in C/C++, which significantly outperforms the open source Fortran-based MC software package MCCCS Towhee [25]. 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. A similar effort of redesigning a large-scale system is the highly optimised object-oriented many particle dynamics (HOOMD) engine. It was created by Ames Lab [3] in collaboration with

Iowa State University and, later, adopted by the University of Michigan (HOOMD-blue) to perform MD simulations utilising GPUs. HOOMD-blue utilises CUDA at its core and, additionally, showcases many of the innovations expected of a modern reworking for a simulation engine – executing a simulation with performance equivalent to that of using a 30 processor core cluster [3].

Until now, no available GPU-based MC engine has been developed for standard thermodynamic ensemble simulations of Lennard-Jones particles to this scale. However, GPU-driven MC simulations of chemical systems have been performed, using lattice gauge theory [10], Ising models [32] and simulations of hard spheres [15]. Calculations of the Lennard-Jones potential are significantly more computationally expensive than the Ising or hard sphere models, because the interactions between all particles within a certain cut-off radius, r_{cut}, must be calculated. Recently, a work was published using lookup tables for the canonical ensemble simulation, which focuses on a small-size system ($N = 128$) [19] using the embarrassingly/pleasingly parallel algorithm [1] of multiple identical lightweight single-thread simulations. The authors suggest that 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 [41] suggested, but did produce insufficient performance. However, the GPU architecture requires a re-examination of the older algorithm that is deemed inefficient on CPUs.

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 towards manycore technology that provides more parallelism for this algorithm. We present a novel optimised GPU-based MC 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 with 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. In other words, the 20 times speedup did not come from a faster GPU run, rather, it is a result of running a slower CPU; the running time of the parallel code should be the same in both cases.

The structure of this document is as follows. Related work and different applications of the GPU are introduced in Section 2. Section 3 provides a brief overview of MC simulations and methods used to utilise this simulation. Relevant features of the CUDA framework are presented in Section 4. Discussion of the key factors that affect the design of the parallel algorithm, in addition to illustration of the proposed algorithm, can be found in Section 5. Section 6 discusses the performance impact of these key design factors. In addition, a detailed performance comparison between the parallel and serial algorithms is presented. Finally, Section 7 elaborates on future extensions to this work in regard to simulating other systems in parallel using the CUDA framework.

2. Related work

Recently, a wide variety of applications have reported substantial performance gains from the use of GPUs. Many scientific 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 have started utilising the power of this inexpensive technology. For example, data clustering [21], protein folding [39], macromolecular simulation [38], stock pricing [22], speech recognition [18], sorting and searching [4], standard query language (SQL) queries [5], kernel machines [35], magnetic resonance imaging (MRI) reconstruction [37] and Euclidean distance map [24].

MD codes exist, some of which have been modified to utilise the GPU, including large-scale atomic/molecular massively parallel simulator (LAMMPS) [9], nanoscale molecular dynamics (NAMD) [31], assisted model building with energy refinement (AMBER) [34] and HOOMD-blue [3], which was developed from scratch to support the GPU. Existing GPU-enabled MD codes are inadequate for many biomolecular systems of interest, which requires the simulation of an open system. The MC method is the ideal technique for this class of biomolecular systems. However, to the best of our knowledge, there is only one open-source MC code (Towhee) [25], and there are no open-source MC codes that utilise GPUs in any form. As a result, only small problem sizes can be run in a reasonable amount of time and this constrains the size of MC simulations. While systems containing more than 100,000 atoms are routinely simulated with MD, MC simulations are typically limited to systems containing less than 2000 atoms. Another recent work on MC simulation on the GPU for systems of hard disks can be found in [2]. In this method, a spatial decomposition technique is used, in which 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. Maintaining a detailed balance in this algorithm adds extra overhead to the original algorithm and to the process of verifying results. In addition, while this algorithm has been conducted only for 2D systems, scaling from 2D checkerboarding to 3D checkerboarding is a very difficult task.

3. Markov chain MC simulations

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

3.1 Metropolis method and thermodynamic ensembles

While there are many approaches to applying MC methods to molecular systems, the most popular one is called the *Metropolis method* [27]. In general, the Metropolis MC method [33] is a computational approach to generate a set of C configurations of the system.

An *ensemble* (also statistical ensemble or thermodynamic ensemble) is an idealisation consisting of a large number of mental copies (sometimes infinitely many) of a system, considered all at once, each of which represents a possible state that the real system might be in [14]. Figure 1 shows one of the most common ensembles used in the literature that is the *canonical ensemble* in which the number of particles (N), volume (V) and temperature (T) are fixed. However, the system energy (E) and pressure (P) are variables. Sometimes, this system is referred to as the *NVT* ensemble. Using MC trials of different configurations,

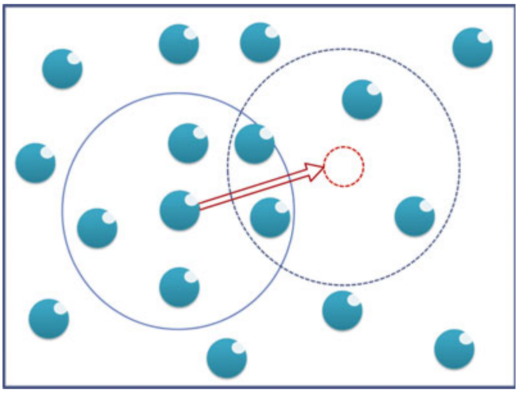

Figure 1. A particle displacement attempt in Metropolis MC method.

as per the *Boltzmann's ergodic hypothesis* [7], 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

$$e^{(-\beta \Delta E)}, \tag{1}$$

where ΔE is the change in energy from the previous state to the tested state, $\beta = (1/k_B T)$, k_B the *Boltzmann constant* and T the temperature of the system. The result of this equation is typically compared with 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 [14].

3.2 Lennard-Jones potential

The Lennard-Jones potential is a frequently used short-range interaction model to simulate interactions between a pair of particles [12]. The potential equation is given by:

$$U_{LJ} = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], \tag{2}$$

where ϵ is the depth of the potential well, σ the collision diameter for interacting particles and r 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 is given by the equation:

$$F_{LJ} = 24\epsilon \left[2\left(\frac{\sigma^{12}}{r^{13}}\right) - \left(\frac{\sigma^{6}}{r^{7}}\right) \right]. \tag{3}$$

This portion of the simulation is responsible for nearly all of the execution time [23]. 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 [14] for the proof of the validity of this method and further chemical details.

4. Parallel model

4.1 CUDA architecture

We use the Fermi GPU architecture from NVIDIA® [20,29,30,40] for our experiments. Early GPU architectures had none or limited double-precision floating-point capabilities, which restricted the use of GPUs for scientific applications. GPUs built on Fermi have advanced double-precision capabilities that enhance the performance of programs that require the use of double-precision operations. Moreover, Fermi GPUs introduced faster atomic operations, which enhanced many operations such as sorting and reduction. It also provides faster context switching, support for concurrent kernel execution, improved thread block scheduling, improved branch prediction, the addition of an L2 cache, and more registers and shared memory [23,40].

CUDA program development incorporates the use of threads that run a specified function called a *kernel*. Threads are organised into blocks, and each grid has multiple blocks of threads. Threads are organised into groups of 32 parallel threads called *warps*. A warp executes one common instruction at a time. Threads inside a block can communicate using the on-chip shared memory. Threads from different thread blocks can communicate using global memory, which is slower than the on-chip shared memory. Figure 2 illustrates the GPU architecture and how the GPU hardware is related to the CUDA threads. In this work, we are using CUDA release 4.0. Significant updates from prior CUDA versions include debugging tools, profiling tools, unified virtual addressing and N-copy pinning of system memory [29].

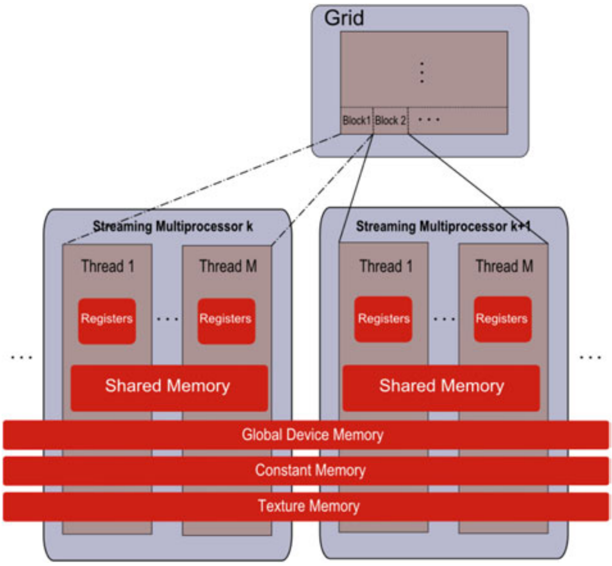

Figure 2. NVIDIA's Fermi architecture abstraction.

4.2 Structure of the code

In order to gain a better understanding of the code 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 results directly to an output device, all results have to be copied back to the CPU for further processing and for output to files. As 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 algorithm due to the serial nature of the MC algorithm. However, specific functions have been ported to the GPU. The flowchart in Figure 3 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 and the number of particles in the system.

Algorithm 1. Serial canonical ensemble Monte Carlo algorithm

```
input: Number of particles and Volume
input: Temperature
input: $\epsilon$, $\sigma$, $r_{\text{cut}}$
// 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 ⟵ rand()
   Old_particle_loc ⟵ particle_location(s)
   // Randomly move to a new location
   New_particle_loc ⟵ 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 ⟵ 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
```

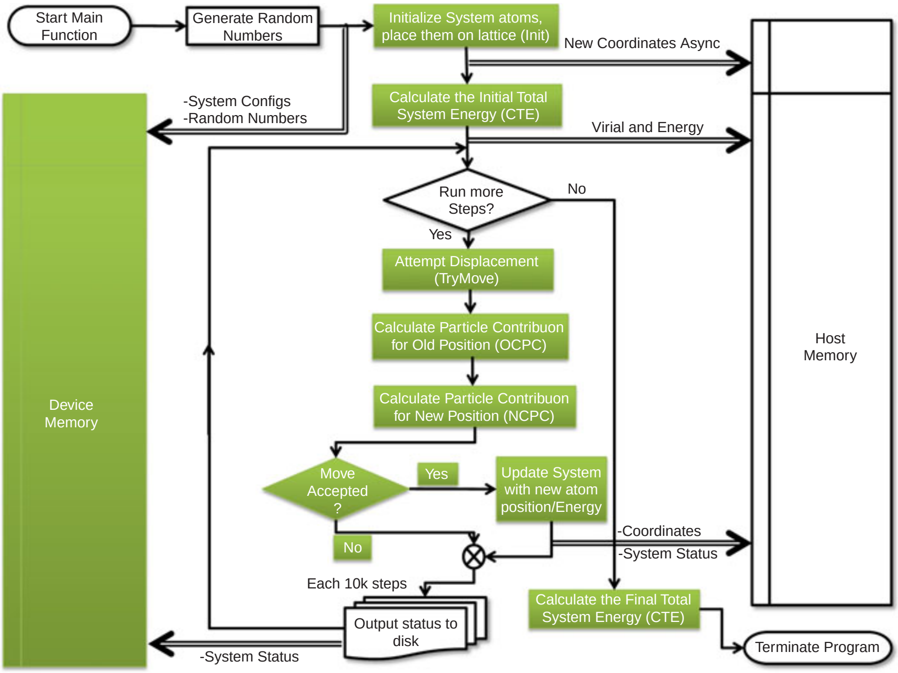
Start Main Function
Generate Random Numbers
Initialize System atoms, place them on lattice (Init)
New Coordinates Async
-System Configs
-Random Numbers
Calculate the Initial Total System Energy (CTE)
Virial and Energy
Run more Steps?
No
Yes
Attempt Displacement (TryMove)
Calculate Particle Contribuon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

Figure 3. MC simulation for the canonical ensemble method flowchart. Filled shapes represent operations executed on the GPU.

(2) Repeatedly perform trial move attempts within the main loop. For each trial, pick a random particle to move and calculate the difference in energy (ΔE) for the selected particle in the old and new locations. This includes the following:
   (a) Assign threads to particles
   (b) Calculate partial energy sums from all threads
   (c) Calculate partial energy sums from all blocks
   (d) Calculate ΔE
   (e) Calculate the Boltzmann factor
(3) Compare a random number with 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 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 labelled to illustrate the specific data being transferred for that particular step. Eventually, the host has the main loop that the simulation executes, in 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 *CalculateTotalEnergy()* 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 optimisations 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.

As Figure 3 shows, almost all functions have been moved to the device. Besides some initialisation of variables and allocation of storage, those functions are as follows:

- *InitialiseAtoms()*, which is called once at the beginning to initialise atom positions on the grid.
- *CalculateTotalEnergy()* to calculate the system's initial energy at the beginning of the simulation and final energy at the end of the simulation.
- *TryMove*(), which is executed at each simulation step to attempt a displacement move.
- *CalculateParticleContribution()* for the energy of the selected particle in the old and the new locations in each simulation step.

On the other hand, the only two functions that do not execute on the device are the function that dumps the current system status to disk, *WriteSystemStatusToDisk()*, and the function that generates the random number sequence, *MTRandomSequence()*.

5. Optimising the MC algorithm for the GPU

In this section, we shall consider specific strategies implemented to optimise the MC code suite. This list mentions a number of significant optimisations that have boosted the performance of the MC simulation. As a parallel algorithm cannot be generalised to all problem domains, we have focused on the optimisations that enhanced the overall performance of this particular class of problems.

5.1 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 drawback to using smaller block sizes is the reduced sharing of data among threads.

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 synchronisation mechanisms. While threads in the same block can synchronise execution through lightweight CUDA statements such as `__syncthreads()`, threads in different blocks need other techniques to accomplish synchronisation such as the technique mentioned in Section 5.7. Some modest performance gain may be possible by tuning the block size separately for each function. As the vast majority of the running time (not counting memory transfers) is used to execute the TryMove kernel, we did not evaluate this idea. In this study, several block sizes are examined and we have reported the performance measurements. The same block size is used for all the GPU functions.

5.2 *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 MC simulation is the storage of generated pseudo-random numbers. Random numbers are needed for each step of the algorithm; we used the Mersenne Twister [26] 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. As the random numbers and the coordinates of the particles are the only large memory structures that are transferred between the CPU and the GPU, only these two structures use pinned memory.

5.3 *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 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 lists the different memory specifications for the cards used in this study.

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, which is used for the CalculateTotalEnergy (CTE) and TryMove functions. However, Section 5.8 shows an unavoidable use of global memory to synchronise 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.

5.4 *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 neighbouring 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

Table 1. Thread hierarchy and properties.

		Associated resources	
Coarse	Size	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

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
$i \leftarrow$ offset
**while** $i > 32$ **do**
    **if** tid $< i$ **then**
        cachedEnergy[tid] $\leftarrow$ cachedEnergy[tid $+ i$] $+$ cachedEnergy[tid]
    **end if**
    _syncthreads()
    $i \leftarrow i/2$
**end while**
// Find the sum of the first 64 values
**if** tid $< 32$ **then**
    offset $\leftarrow$ min(off_set, 64)
    **switch** (off_set) **do**
        **case** *64*: cachedEnergy[tid] $\leftarrow$ cachedEnergy[tid $+ 32$] $+$ cachedEnergy[tid]
        **case** *32*: cachedEnergy[tid] $\leftarrow$ cachedEnergy[tid $+ 16$] $+$ cachedEnergy[tid]
        **case** *16*: cachedEnergy[tid] $\leftarrow$ cachedEnergy[tid $+ 8$] $+$ cachedEnergy[tid]
        **case** *8*:  cachedEnergy[tid] $\leftarrow$ cachedEnergy[tid $+ 4$] $+$ cachedEnergy[tid]
        **case** *4*:  cachedEnergy[tid] $\leftarrow$ cachedEnergy[tid $+ 2$] $+$ cachedEnergy[tid]
        **case** *2*:  cachedEnergy[tid] $\leftarrow$ cachedEnergy[tid $+ 1$] $+$ cachedEnergy[tid]
    **end switch**
**end if**
```

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.

5.5 Loop unrolling technique in finding total energy

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, that mentioned in [16], in which slightly more optimisations 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 [16], 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 8, then the case for 8, 4 and 2 will be executed. This technique is illustrated in Figure 4. For blocks with enough threads, applying this method to the kernel code enhances performance significantly.

5.6 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

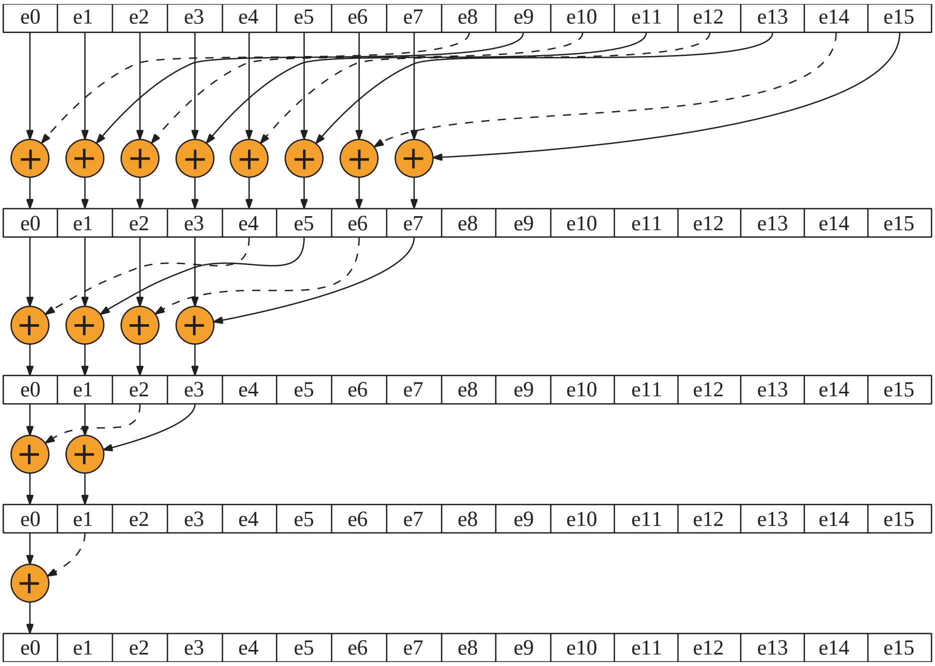
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14 e15
+ + + + + + + +
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14 e15
+ + + +
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14 e15
+ +
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14 e15
+
e0 e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13 e14 e15

Figure 4. Calculating the partial sum for values in shared memory that use adjacent memory locations. Each circle represents work done by a thread.

potential interaction. An $N \times N$ square matrix can be used to illustrate the pair interactions as seen in Figure 5. 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 4. With this technique, threads will access adjacent locations on each pass, which reduces the memory latency. Figure 5 provides further illustration of the mapping algorithm.

5.7 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 synchronise blocks in a grid, as blocks can share only global memory. This is discussed further in Section 5.8. On the other hand, atomic operations can lower performance as they serialise accesses to global memory, add extra instruction processing and require busy waiting. However, Fermi cards offer a more efficient implementation of atomic operations.

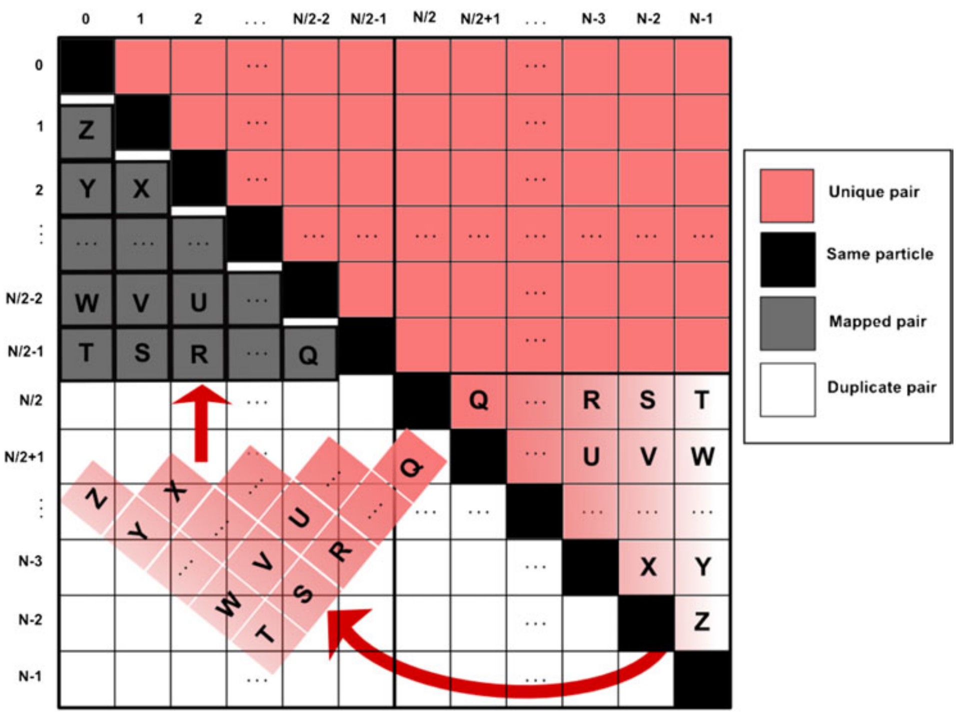

Figure 5. Mapping algorithm for work load balancing across threads.

5.8 Block synchronisation through global memory and atomic operations

The current structure of the CUDA architecture does not support explicit synchronisation 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 [30]. 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.

5.9 Numerical optimisations: tricks and tweaks

Several mathematical operations have been optimised 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 2, respectively. Since MC simulation does a significant number of repetitive mathematical operations, these optimisations gave the most performance improvement among all the optimisations tested.

Another optimisation technique is to use more efficient mathematical operations on the GPU. For example, a double-precision division operation such as $[A < (B/2.0)]$ is

Algorithm 3. Block synchronisation technique

```
// Thread 0 of each block has the sum of all values for that block
if tid $= 0$ then
    GlobalEnergy[blockId] $\leftarrow$ cachedEnergy[0]
    LastBlock $\leftarrow$ 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] $\leftarrow$ GlobalEnergy[tid]
    end if
    // If you have more blocks than threads, reduce the extra values.
    $i \leftarrow$ ThreadsPerBlock
    while $i <$ gridDim do
        if tid $+ i <$ BlocksPerGrid then
            cachedEnergy[tid] $\leftarrow$ GlobalEnergy[tid $+$ i] $+$ cachedEnergy[tid]
        end if
        $i \leftarrow i +$ 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
```

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 optimisation is when calculating the Boltzmann factor in Equation (1). As the denominator is a constant, we instead compute the reciprocal once and replace a division with a multiplication.

6. Results and discussion

While we are using the CUDA architecture as an extension to the C language to implement the parallel algorithm, other MC 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 agreement with publicly available canonical ensemble simulation results from the National Institute of Standards and Technology [36]. 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 [25] serial code using the same configurations, shown in Table 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, as 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 2 are 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

Table 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	32,150.3	438.3

(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 optimisations. 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 16 hundred runs, only 9 deviated 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. 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 with the other two cards. Although 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

Table 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
Multiprocessors	7	7	15
Max shared mem. per block (kB)	48	48	48
Global mem. (GDDR 5) (GB)	1	1	1536
Processor clock (MHz)	1300	1700	1401
Max. block size	1024	1024	1024
Mem. bandwidth (GB/s)	108.8	128	177.4
Compute capability	2.1	2.1	2.0

Table 4. Desktop computers used for the experiments.

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

Intel® Xeon® processor. The results achieved in this work were obtained with the commodity desktop processors described in Table 4.

Table 5 shows different block sizes and their effect on the overall simulation speedup compared with the single-core serial code. Performance-wise, the following can be noted:

(1) When the number of threads per block is small, the need for more global memory accesses for synchronisation 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 4096 blocks.

(2) Sixty-four threads per block offers the best performance when there are less than 8192 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 [30]. However, end-to-end execution time is the worst when the system size is larger than 8192 particles, as seen in Table 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 8192 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 5, also plotted in Figure 6, 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 6, 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 7 plots the execution times in seconds for the serial code against Towhee on an Intel® Core™ i5, and Figure 8 compares the execution time of the serial algorithm with the

Table 5. Large versus small block size and system performance.

	Number of particles									
	256	512	1024	2048	4096	8192	16,384	32,768	65,536	131,072
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

Note: Numbers shown are speedup.

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 6. Memory transfers and parallel function invocation are the main causes of this overhead. The speedup ratio seen in Figure 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 9 shows that the speed of the CPU (host) has a negligible impact on the execution

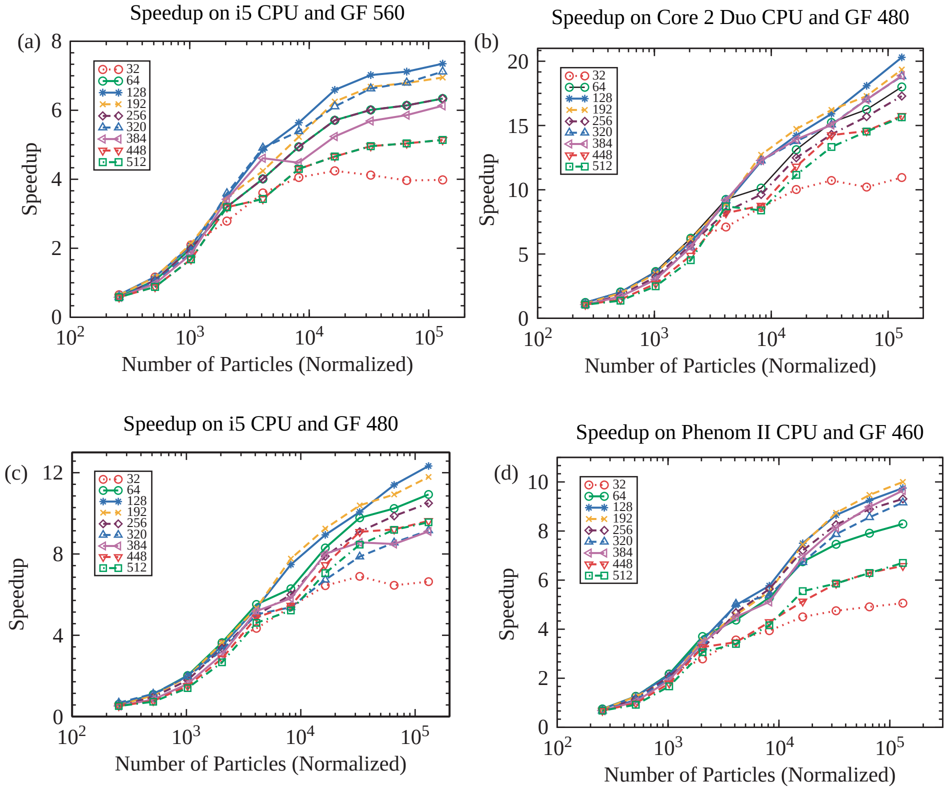

Figure 6. Plots of speedup for different block sizes on different platforms.

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.

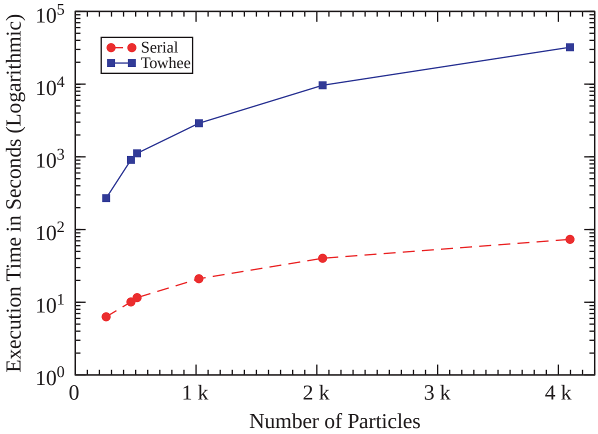

Figure 7. Developed serial code versus Towhee elapsed times (logarithmic normalisation).

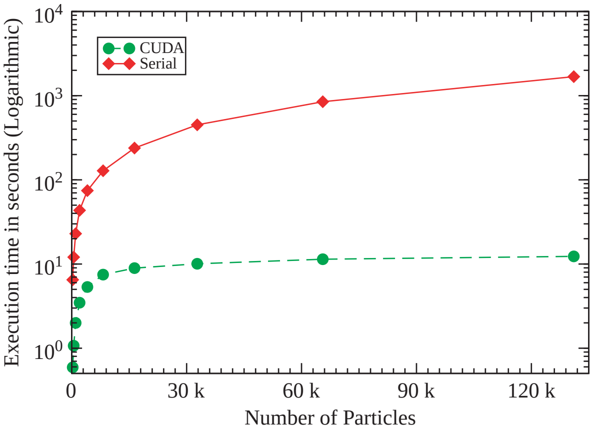

Figure 8. Serial versus CUDA execution times for MC simulation on i5 and GeForce 480.

Table 6. Average program execution times (in seconds) and speedup for a million steps on i5 and GeForce 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
16,384	238.8	26.7	9.0
32,768	450.6	44.7	10.1
65,536	847.0	74.3	11.4
131,072	1681.9	136.4	12.3
262,144	3659.8	243.7	15.0

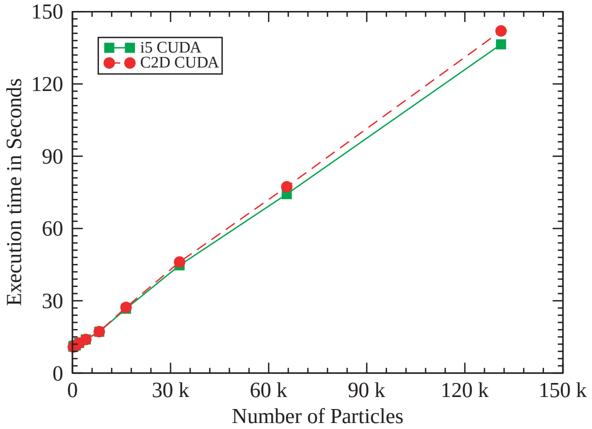

Figure 9. Execution times on two different platforms with GTX 480.

7. Conclusions and future work

In this paper, an optimised implementation of a parallel CUDA algorithm for Markov chain MC simulations has been presented. Optimisations have been applied to the algorithm, such as the use of shared memory and load balancing among threads. In addition, synchronisation 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.

The evaluation of the proposed algorithm on an affordable graphics processing unit shows a speedup of up to 15 times compared with the optimised serial implementation, and 2303 times speedup compared with Towhee for a small problem of size 4096. Through the process of developing the parallel algorithm, an empirical optimisation approach has been applied. This approach centres on selecting optimal block sizes for kernel invocation, which was 128 threads per block for large problem sizes.

The dramatic increase in speed achieved with GPU-based applications opens the door for the design and analysis of experiments that were previously not feasible, running an experiment in hours compared to days of computing time on a desktop computer. This will assist researchers in simulating a number of large biomolecular systems that require simulation of an open system (constant chemical potential).

The presented implementation of the MC simulation algorithm using CUDA can be generalised. Therefore, it would be interesting to see whether this algorithm or its extensions with other ensembles can be integrated into widely used scientific applications. In addition, systems with more than one top of the line GPU are being used in research laboratories and data centres. Yet, developing a highly flexible multi-ensemble code capable of fully utilising the multiple devices on these high-performance clusters is still a work in progress. Hence, in addition to expansion of the simulation capabilities, a multi-GPU algorithm [8] for use in supercomputing clusters would allow for faster performance and the handling of more particle data, opening the way to simulations of even larger systems (e.g. a microparticle–nanoparticle open constant chemical potential simulation).

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 neighbouring particles for such low computation intensive applications was not promising before. Now, with the possibility of simulating very large systems, a neighbour list algorithm [14] could show decent speedup for systems with 10,000 particles or more. This would be a natural extension to the work presented here.

Acknowledgements

The authors thank the anonymous reviewers for their insightful comments and invaluable feedback. This work has been supported by Wayne State University's Research Enhancement Program (REP) and National Science Foundation (NSF) grants NSF CBET-0730768 and OCI-1148168. The authors also thank NVIDIA® for donating some of the graphics cards used in this study.

Notes

1. Threads in the same warp do not need to be synchronised.
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.

References

[1] D. Adams, *The implementation of fluid phase Monte Carlo on the dap*, J. Comput. Phys. 75 (1988), pp. 138–150. Available at http://www.sciencedirect.com/science/article/pii/0021999188901039

[2] J.A. Anderson, E. Jankowski, T.L. Grubb, M. Engel, and S.C. Glotzer, *Massively parallel Monte Carlo for many-particle simulations on GPUs*, ArXiv e-prints (2012).

[3] J. Anderson, C. Lorenz, and A. Travesset, *General purpose molecular dynamics simulations fully implemented on graphics processing units*, J. Comput. Phys. 227 (2008), pp. 5342–5359.

[4] U. Assarsson and E. Sintorn, *Fast parallel gpu-sorting using a hybrid algorithm*, J. Parallel Distrib. Comput. 68 (2008), pp. 1381–1388.

[5] P. Bakkum and K. 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, Pittsburgh, PA, ACM, New York, NY, 2010, pp. 94–103.

[6] K. Binder, *Monte Carlo and Molecular Dynamics Simulations in Polymer Science*, Oxford University Press, New York, 1995.

[7] G. Birkhoff, *Proof of the ergodic theorem*, Proc. Natl. Acad. Sci. USA 17 (1931), pp. 656–660.

[8] B. Block, P. Virnau, and T. Preis, *Multi-gpu accelerated multi-spin Monte Carlo simulations of the 2d Ising model*, Comput. Phys. Commun. 181 (2010), pp. 1549–1556.

[9] W. Brown, S. Hampton, P. Agarwal, P. Wang, P. Crozier, and S. Plimpton, *Porting lammps to gpus*, Tech. Rep., Sandia National Laboratories, 2010.

[10] N. Cardoso and P. Bicudo, *Su(2) lattice gauge theory simulations on fermi gpus*, J. Comput. Phys. 230 (2011), pp. 3998–4010.

[11] P. Ciarlet, C. Le Bris, and J. Lions, *Handbook of Numerical Analysis, Special Volume: Computational Chemistry*, Vol. 10, North Holland, Amsterdam, 2003.

[12] P. Deuflhard, *Computational molecular dynamics: Challenges, methods, ideas*, in *Proceedings of the 2nd International Symposium on Algorithms for Macromolecular Modelling*, 4th ed., Springer-Verlag New York, Inc., Secaucus, NJ, 1999, pp. 98–115.

[13] T. Duren, Y.S. Bae, and R.Q. Snurr, *Using molecular simulation to characterise metal–organic frameworks for adsorption applications*, Chem. Soc. Rev. 38 (2009), pp. 1237–1247.

[14] D. Frenkel and B. Smit, *Understanding Molecular Simulation*, 2nd ed., Academic Press, San Diego, CA, 2002.

[15] A. Frezzotti, G.P. Ghiroldi, and L. Gibelli, *Direct solution of the Boltzmann equation for a binary mixture on GPUs*, AIP Conf. Proc. 1333 (2011), pp. 884–889.

[16] M. Harris, *Optimizing Parallel Reduction in Cuda*, 2008. Available at http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[17] Z.K. Issa, C.W. Manke, B.P. Jena, and J.J. Potoff, *Ca^{2+} bridging of apposed phospholipid bilayers*, J. Phys. Chem. B 114 (2010), pp. 13249–13254.

[18] J. Kim, J. Chong, and I.R. Lane, *Efficient on-the-fly hypothesis rescoring in a hybrid GPU/CPU-based large vocabulary continuous speech recognition engine*, in *Proceedings of 13th Annual Conference of the International Communication Association (Interspeech),* Portland, OR, USA, 2012.

[19] J. Kim, J.M. Rodgers, M. Athènes, and B. Smit, *Molecular Monte Carlo simulations using graphics processing units: To waste recycle or not?* J. Chem. Theory Comput. 7 (2011), pp. 3208–3222.

[20] D. Kirk and W. Wen-mei, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, Boston, MA, 2010.

[21] K.J. Kohlhoff, M.H. Sosnick, W.T. Hsu, V.S. Pande, and R.B. Altman, *Campaign: An open-source library of gpu-accelerated data clustering algorithms*, Bioinformatics 27 (2011), pp. 2322–2323.

[22] C. Kolb and M. Pharr, *Options pricing on the GPU*, GPU Gems 2 (2005), pp. 719–731.

[23] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig, *Accelerating molecular dynamics simulations using graphics processing units with CUDA*, Comput. Phys. Commun. 179 (2008), pp. 634–641.

[24] D. Man, K. Uda, Y. Ito, and K. Nakano, *Accelerating computation of Euclidean distance map using the gpu with efficient memory access*, Int. J. Parallel Emergent Distrib. Syst. 28 (2012), pp. 1–24.

[25] M. Martin, B. Chen, C. Wick, J. Potoff, J. Stubbs, and J. Siepmann, Mcccs towhee (2012). Available at http://towhee.sourceforge.net/

[26] M. Matsumoto and T. Nishimura, *Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM Trans. Model. Comput. Simul. 8 (1998), pp. 3–30.

[27] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, *Equation of state calculations by fast computing machines*, J. Chem. Phys. 21 (1953), pp. 1087–1092.

[28] J. Nickolls and W. Dally, *The gpu computing era*, IEEE Micro 30 (2010), pp. 56–69.

[29] NVIDIA, *Best Practices Guide*, 2011. Available at http://www.nvidia.com/cuda

[30] NVIDIA, *CUDA C Programming guide 4.0*, 2011. Available at http://www.nvidia.com/cuda

[31] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kal, and K. Schulten, *Scalable molecular dynamics with namd*, J. Comput. Chem. 26 (2005), pp. 1781–1802.

[32] T. Preis, P. Virnau, W. Paul, and J. Schneider, *Gpu accelerated Monte Carlo simulation of the 2d and 3d Ising model*, J. Comput. Phys. 228 (2009), pp. 4468–4477.

[33] M. Rosenbluth and A. Rosenbluth, *Further results on Monte Carlo equations of state*, J. Chem. Phys. 22 (1954), pp. 881–885.

[34] R. Salomon-Ferrer, D.A. Case, and R.C. Walker, *An overview of the amber biomolecular simulation package*, Wiley Interdiscip. Rev. Comput. Mol. Sci. 3 (2013), pp. 198–210.

[35] B. Srinivasan, Q. Hu, and R. Duraiswami, *Gpuml: Graphical processors for speeding up kernel machines*, Workshop on High Performance Analytics-Algorithms, Implementations, and Applications, Columbus, OH, USA, 2010.

[36] T.N.I. Standardsof, T. (NIST), Benchmark Results for Lennard-Jones Fluid-nvt Monte Carlo Results at Both Liquid- and Vapor-Like Densities (2012). Available at http://cstl.nist.gov/srs/LJ_PURE/mc.htm

[37] S. Stone, J. Haldar, S. Tsao, W. Hwum, B. Sutton, and Z.P. Liang, *Accelerating advanced MRI reconstructions on GPU*, J. Parallel Distrib. Comput. 68 (2008), pp. 1307–1318.

[38] M. Taufer, N. Ganesan, and S. Patel, *Gpu-enabled macromolecular simulation: Challenges and opportunities*, Comput. Sci. Eng. 15 (2013), pp. 56–65.

[39] V.A. Voelz, G.R. Bowman, K. Beauchamp, and V.S. Pande, *Molecular simulation of* ab initio *protein folding for a millisecond folder NTL9*, J. Amer. Chem. Soc. 132 (2010), pp. 1526–1528.

[40] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu, *Fermi gf100 gpu architecture*, IEEE Micro 31 (2011), pp. 50–59.

[41] S.J. Zara and D. Nicholson, *Grand canonical ensemble Monte Carlo simulation on a transputer array*, Mol. Simul. 5 (1990), pp. 245–261.