An Efficient Cell List Implementation for Monte Carlo Simulation on GPUs

Loren Schwiebert
Dept. of Computer Science
Wayne State University
Detroit, MI 48202
loren@wayne.edu

Eyad Hailat
Dept. of Computer Science
Wayne State University
Detroit, MI 48202
eyad@wayne.edu

Kamel Rushaidat
Dept. of Computer Science
Wayne State University
Detroit, MI 48202
ed3457@wayne.edu

Jason Mick
Dept. of Chemical Engineering
Wayne State University
Detroit, MI 48202
dw2413@wayne.edu

Jeffrey Potoff
Dept. of Chemical Engineering
Wayne State University
Detroit, MI 48202
jpotoff@wayne.edu

August 19, 2014

Abstract

Maximizing the performance potential of the modern day GPU architecture requires judicious utilization of available parallel resources. Although dramatic reductions can often be obtained through straightforward mappings, further performance improvements often require algorithmic redesigns to more closely exploit the target architecture. In this paper, we focus on efficient molecular simulations for the GPU and propose a novel cell list algorithm that better utilizes its parallel resources. Our goal is an efficient GPU implementation of large-scale Monte Carlo simulations for the grand canonical ensemble. This is a particularly challenging application because there is inherently less computation and parallelism than in similar applications with molecular dynamics. Consistent with the results of prior researchers, our simulation results show traditional cell list implementations for Monte Carlo simulations of molecular systems offer effectively no performance improvement for small systems [5, 14], even when porting to the GPU. However for larger systems, the cell list implementation offers significant gains in performance. Furthermore, our novel cell list approach results in better performance for all problem sizes when compared with other GPU implementations with or without cell lists.

1 Introduction

Graphics acceleration hardware has been commercially available to consumers since the early 1980s. The highly-parallel architecture has encouraged researchers to port many different applications

to these devices. With the availability of the CUDA API [17, 25] writing code that runs on NVIDIA GPUs became a much easier task. Although significant speedups by the GPU over serial implementations have been achieved for many applications, not all applications are easily ported to the GPU. In particular, problem domains with serial behavior such as Markov chain algorithms [7] often benefit little from the SIMD parallelism available on the GPU. In this class of problems, one simulation step depends on the results of the previous step. Furthermore, due to the nature of the problem, predicting the results of the previous step creates a bias and invalidates the sampling distribution of the algorithm.

The main goal of simulating molecular systems using Monte Carlo (MC) and molecular dynamics (MD) methods 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 feasible to simulate open systems that MD cannot simulate, because the MD algorithms are not designed for systems that permit the addition or deletion of particles [7]. For example, MC can be applied to the grand canonical ensemble method, which is useful in adsorption studies, where the amount of material adsorbed is a function of the chemical potential and temperature of the reservoir with which the material is in contact. Moreover, in simulations of two phase systems, the properties of the system vary widely in the interfacial region, such as between the gas and adsorbent, which are a strong function of system size. Hence, it is necessary to simulate large systems to minimize the influence of the interface on the properties of the corresponding bulk phases [7, 21].

Other simulation techniques, such as molecular dynamics, typically require significantly more computation for each step of the simulation, and so are better-suited for parallel implementation. Even so, some previous simulations have used the GPU to implement the MC method [10]. Their implementation depends on an embarrassingly parallel algorithm that runs several concurrent simulations with small systems of 128 particles. Instead, our work uses the energy decomposition method (farm algorithm), which enables us to support configurations with over a million particles.

In [18], a parallelization method for the canonical MC simulations via domain decomposition technique has been presented, where each domain can be assigned to a separate processor and multiple moves can be simulated in parallel. Interprocess communication is required only when moving particles near the edge of a domain, since this requires interactions between adjacent domains. To limit this communication, each domain is partitioned into three subdomains. The size of the middle subdomain is chosen as large as possible to minimize interprocess communication. Although well-suited for a multi-core CPU, this approach does not expose the fine-grained parallelism required for an efficient GPU implementation.

Each time a particle is displaced in, removed from, or inserted into the simulation region, energetic decomposition requires that pairwise energy be calculated between this particle and all other particles. A radial cutoff is typically chosen to reduce the execution time by limiting the calculation of inter-molecular forces to only those particles within the cutoff. The forces due to interactions with particles outside of the cutoff can be approximated using tail corrections.[1] Since interactions within only a small radius are considered, it is possible to create either a cell list or a neighbor list to organize particles based on their relative locations and ignore particles that are beyond the cutoff. In this way, not only are the energy and pressure computations of more distant pairwise interactions avoided, but also the calculation of distances between these particles.

[1]This is a reasonable approximation because the atomic forces decrease at the rate of $O(d^6)$ or $O(d^{12})$ with the distance, d.

A number of researchers have proposed efficient techniques for maintaining these lists. One approach, the Verlet list [24] or neighbor list, maintains a list of neighboring particles, particles within the radial cutoff, for each particle. This minimizes the number of interactions that must be computed, but requires frequent updating. To reduce the frequency of updating, the Verlet list can be expanded to include particles slightly beyond the cutoff, so as particles move they do not enter or exit another particle's neighbor list until n simulation steps. Although this is a reasonable option for MD simulations, where all the particles move with each step, this is not practical for MC simulations. In MC simulations, typically only a single particle is moving in each step and the displacement of this particle can be arbitrarily large. In addition, inserting or deleting a particle from the simulation, which is not possible in MD simulations, would require rebuilding the Verlet list after this move.

Another approach is to use a cell list, where the simulation box is partitioned into cells (squares in 2D, cubes in 3D) such that each cell has dimensions not much larger than the cutoff and perhaps smaller. The only particles that need to be considered are those in the same cell and the adjoining cells. For larger systems, as the simulation box grows relative to the cutoff, an increasing portion of the particles can be ignored. Because many of the particles in adjacent cells will be outside the cutoff, more potential pairwise interactions are computed than with a Verlet list. However, a cell list can be maintained with less overhead than a Verlet list. To reduce the number of extraneous particles processed in the cell list implementation, one can sort the particles in each cell [8, 26]. Or the cells can be made smaller, with more cells processed for each move [2, 14]; this can be taken to the extreme of a cell size large enough for only a single particle. However, this requires many more cells to be examined, many of which will be empty, which also introduces some overhead. Since the cell list approach is more promising for MC simulations on the GPU, for the rest of this paper we consider only cell lists.

A third option is to use both a Verlet list and a cell list [8]. For instance, Proctor *et al.* [20] show cell lists on the GPU allow a fast approximation of whether or not two particles are within the cutoff, which performs better than immediately traversing the neighbor list. They do not create or maintain a cell list, but calculate the cell of each particle based on its coordinates, with cell dimensions larger than the cutoff, and use this calculation to determine whether or not two particles are in neighboring cells.

There are many examples of using a cell list implementation for the MD simulations [3, 6, 7, 22, 23]. On early GPUs, an efficient implementation of cell list on the GPU was not viable due to the lack of atomic operations on the GPU [10]. Instead, implementations such as [3, 22, 23] use the CPU to construct the cell list and then copy it to the GPU. These cell lists are then used to construct a neighbor list. Note that in molecular dynamics simulations, all molecules are moved in each step, requiring the cell list to be updated after nearly every simulation step. The frequency of updates depends on how far a molecule moves in each step, how much extra distance beyond the cutoff is used in defining the neighbors, and how much inaccuracy can be tolerated in the computations. A state-of-the-art implementation is described in [1].

In MD simulations, since all the particles of the system are moving at the same time, the overhead of maintaining the cell list can be eclipsed by the computation cost of each step of the simulation. On the other hand, in a grand canonical MC simulation, only one particle is moved in each step. So, there is less computation. Therefore, prior work on cell lists for MC simulation showed that there is no performance gain for these applications with relatively small systems [5, 14]. Since MC simulations typically have run on systems with at most a few thousand particles, cell

lists have been considered impractical for MC simulations.

In this paper, we reexamine this assumption for MC simulations on the GPU. There are two aspects to this reconsideration. First, with the GPU, it is feasible to simulate much larger systems in a reasonable amount of time. So, it is appropriate to ask whether cell lists might offer a performance advantage for much larger systems. Second, the GPU is a massively multithreaded architecture, so we evaluate a novel cell list implementation designed specifically for a manycore architecture such as the GPU.

To develop our code, we started with a state-of-the-art sequential code developed by Chemical Engineers in our research group to ensure that scientifically valid results are produced. The GPU versions were then compared with the output of this sequential code to ensure that not only fast code, but also correct code was produced. Our purpose is not, however, to compare the sequential code with the GPU code. Instead, we look at various versions of the GPU code, focusing on different cell list implementations to investigate the best cell list implementation for the GPU. Our results clearly demonstrate that a cell list implementation tailored to a manycore architecture offers significant speedup over both the original GPU code and the traditional cell list for problem sizes ranging from 512 particles up to over one million particles.

The rest of this paper is organized as follows. Section 2 introduces the MC simulation for the grand canonical algorithm. Then, in section 3 we introduce techniques, optimizations, and decisions we have made to write the parallel algorithms with and without the cell implementation. Both a traditional cell list implementation and our novel cell list for the GPU are presented. Performance results and observations are discussed in section 4. Section 5 presents our conclusions and future work.

2 Grand Canonical Ensemble

The grand canonical ensemble extends the canonical ensemble by fixing the values of the temperature (T), volume (V), and the chemical potential (μ) [7]. Particles can interact with each other only when they exist inside the simulation box and are within a cutoff radius, r_{cut}, of each other. A reservoir is connected to this simulation box, allowing the particles and energy to be exchanged freely between them. Through this exchange of particles, the system and the reservoir will reach the equilibrium state, which can be determined by using the values of the temperature and the chemical potential.

This method can be applied to problems such as:

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

2. In numerical simulations to accurately predict properties of materials and their guest-adsorption characteristics.

3. Determining the equation of state of the Lennard-Jones fluid. One fixes the temperature and chemical potential and calculate the resulting density and pressure.

We study systems of particles interacting via the Lennard-Jones potential by calculating the

configurational energy of pairwise interaction, given by:

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right], \tag{1}$$

where r is the distance between two interacting particles, ϵ is the depth of the potential well, and σ is the particle diameter. Note that calculations of the Lennard-Jones potential are significantly more complex than the Ising or hard sphere models, since the system must calculate the interactions between all particles within a certain cutoff radius. On the other hand, the hard sphere system just requires checking for overlap. In the Ising model, we are only calculating interactions between nearest neighbors and those nearest neighbors are always known, since the spins do not change locations during the simulation.

Particles in this ensemble are moving inside the simulation box or between the box and the reservoir. The acceptance of a move is determined by computing the Boltzmann factor:

$$B_F = e^{-\beta \Delta E}, \tag{2}$$

where β is given by $(1/k_B T)$, k_B is the *Boltzmann constant*, and T is the temperature of the system. $\Delta E = [U(s'^N) - U(s^N)]$ is the difference between the new system energy and the old one, where U is the total energy of the system for a given configuration, N is the number of particles in the box, and s and s' represent the old and new positions of the particle, respectively.

The acceptance or rejection for the types of particle moves are given by the Metropolis acceptance criterion [15, 4]:

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

$$acc(s \to s') = \min\left[1, B_F\right] \tag{3}$$

Insertion A particle is inserted from the infinite reservoir into a random location in the simulation box. The acceptance probability of this move is giving by:

$$acc(C \to C+1) = \min\left[\, 1, \frac{V B_F}{\Lambda^3 (N+1)} \right], \tag{4}$$

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

Deletion The transfer of a random particle out of the simulation box into the reservoir is accepted with a probability:

$$acc(C \to C-1) = \min\left[1, \frac{\Lambda^3 N B_F}{V}\right], \tag{5}$$

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

Algorithm 1 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.

In each of the three moves, the computational cost is dominated by the overhead to calculate the pairwise energy with all interacting particles. Since this is a Markov chain algorithm, each step needs the current system status to calculate the probability of acceptance for the next one.

Algorithm 1 Serial Grand Canonical Ensemble Monte Carlo Algorithm

```
1: Input: One box of size (N) particles, (V) volume
2: Input: Infinite reservoir
3: //Initialize N particles positions inside the box
4: //Calculate total system energy
5: //Main simulation loop
6: for each step do
7:     //Randomly select a move type
8:     R ← 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 ← rand()
16:        if (Source < 0.5) then
17:            //Source box is the box (Deletion)
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 write system status to disk
24: end for
```

3 GPU Implementations

Due to the highly multithreaded architecture of graphics devices, fine-grained parallelism is needed to keep the processors in the device busy. For example, on NVIDIA GPUs, threads are scheduled in warps of 32 threads. For good performance, 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 GPU has high memory bandwidth, the relatively high latency of global memory accesses has to be hidden through sufficient parallelism to get efficient performance.

3.1 Implementation without Cell List

Although the simulation steps have been implemented on the GPU, the CPU makes the decision on which move to execute next and calls the corresponding kernel. Moreover, the simulation periodically writes system status to disk, an operation not supported by current GPUs.

In this section, we describe in detail the simulation moves implemented on the GPU without the use of cell lists. Our implementation of the traditional cell list and our new microcell list are presented in sections 3.2 and 3.3, respectively.

3.1.1 Calculating Total System Energy

Although this is the most time consuming kernel call, since all pairwise interactions are being calculated, this function is called only once to calculate the initial system energy. The total number of unique pairwise energy calculations is potentially $N(N-1)/2$. Rather than create one thread for

Algorithm 2 Parallel particle displacement

```
1: Input: One box of size (N) particles, (V) volume
2: //Randomly select a particle to displace
3: P ← rand()
4: ΔE ← CalculateParticlesContributionTM()
5: if thread 0 in last block then
6:     Use ΔE to calculate the acceptance rule
7:     //Select a random number A in [0,1)
8:     A ← 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
```

each pair, we create $4N$ threads and use these threads to iterate over all unique pairs of particles using a method akin to the RB technique [16]. Since this is computed only once, minimal effort was expended in optimizing this function.

After each thread finishes calculating the pairwise energy, it writes the results to shared memory. A reduction operation is then executed on all threads in a block. Afterward, the summation of the thread values in the block is transferred to global memory to be visible to other blocks. The last block to finish copies the partial sums from all other blocks into its shared memory. Another iteration of reduction for these sums is executed in shared memory and the result is copied to global memory to be used by other kernel calls.

3.1.2 Particle Displacement within the Box

In this move, a randomly selected particle attempts to move to a random position 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 energy contribution from its original location, then calculating the additional energy for the new location. Since only one particle is moved, this approach allows us to update the system energy using an $O(n)$ operation. The reduction technique described in section 3.1.1 is used here. The difference in energy, ΔE, is used for calculating the Boltzmann factor in equation 3 to decide whether or not to accept the new position. Upon acceptance of the particle displacement, the system status is updated, which includes the new energy, new virial pressure, particle's new position, and other configuration variables.

In algorithm 2, the kernel function TryMove() calls a GPU device function, CalculateParticlesContributionTM(), that returns ΔE. Thread zero of the last block to finish decides whether or not to accept the move by computing the probability of acceptance using ΔE and comparing this with a random value. If the move is accepted, then the new particle position, current energy, etc. are updated. This information is maintained on the GPU and only copied to the CPU for periodic configuration dumps. Algorithm 3 shows the parallel algorithm for the function CalculateParticlesContributionTM().

Algorithm 3 CalculateParticlesContributionTM
(no cell list)

```
1: Input: One thread per particle
2: //Initialize shared memory
3: //Assign a particle for the current thread
4: for the ParticleID corresponding to this thread 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 ΔE
26: //return ΔE to caller via global memory
```

3.1.3 Insertion and Deletion of Particles

To insert a particle from the infinite reservoir, a random position is selected in the simulation box. The device function CalculateParticlesContributionTPT() calculates the energy contribution for the new particle in its new location. The change in system energy due to 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 section 3.1.1 is then executed as illustrated in algorithm 4. Algorithm 5 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 addition of this particle; otherwise, the system configuration remains unchanged. Since each kernel is called with one thread per particle, the number of particles in the box needs to be copied from the GPU to the CPU after each insertion or deletion move.

The deletion move is similar to the insertion move, except that the system is removing a particle and has to change the probability of acceptance as in equation 5. When the deletion step is accepted, the particle moves to the reservoir and the system configuration is updated accordingly.

Algorithm 4 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 ΔE to caller via global memory
```

Algorithm 5 Parallel Insertion/Deletion

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

3.2 Traditional Cell List Implementation

As depicted in figure 1, if we use a short radial cutoff, denoted r_{cut}, which is typical,[2] the simulation box can be decomposed into smaller domains, called cells, with the cell length along each dimension S greater than or equal to r_{cut}. For a given particle, all interacting particles are located in the same or adjoining cells along all axes. Figure 2 shows a 3D model of a simulation box where the dotted cells are the neighboring cells to the cell with grid lines. Each move has a constant upper bound on the number of particles that need to be considered for pairwise interactions, which is dependent on r_{cut} but independent of the system size. This property holds because of the physical reality underlying our simulation, which constrains the minimum distance between particles.

On the other hand, the cell list algorithm suffers from the associated overhead of constructing, storing and accessing, and maintaining the cell structure. For small systems, this overhead may exceed the advantages of the cell list because only a few particles are considered even without cells. In addition, systems with a small box or a large cutoff radius cannot be decomposed into more than three cells per dimension. So, the entire simulation box is included within the adjoining cells, but the overhead of the cell list is also present. Next, we discuss the algorithm for implementing

[2] A cutoff of 2.5σ to 3.0σ is not uncommon.

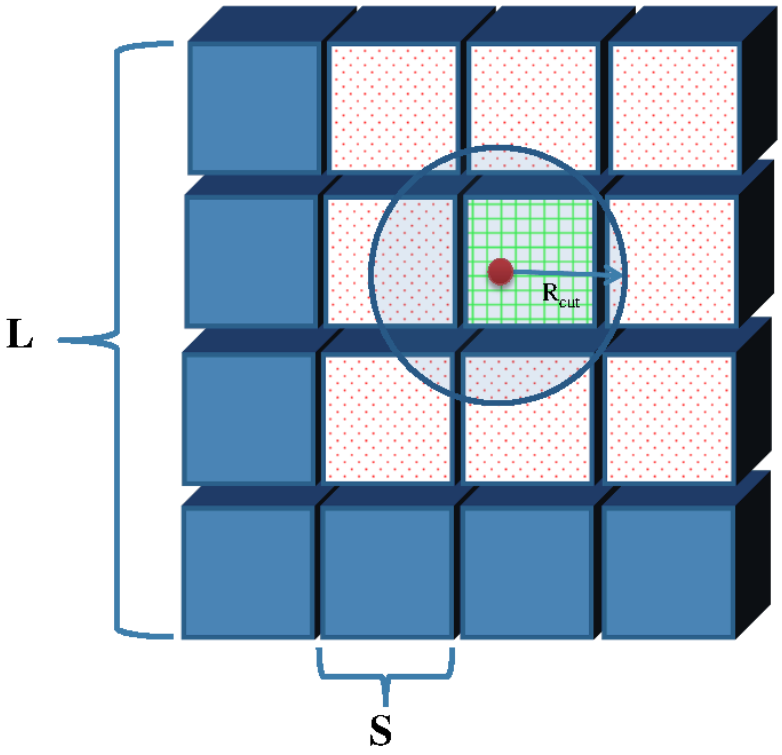


Figure 1: The volume $V = L^3$ is decomposed into $T \geq 3$ cells per dimension, with cell dimension of size $S \geq r_{cut}$.

the traditional cell list and various factors of the design.

3.2.1 Building the Initial Cell List

Many techniques have been proposed for implementing cell lists [7]. On the GPU, we must use a method that exposes a great deal of parallelism. Some approaches use a linked list to store the indices of the particles in each cell, while others assign a fixed sized array of placeholders to every cell. A significant disadvantage with a linked list is that parallel access to the particle indices is not possible. 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 in section 3.2.2.

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*) is constructed to hold the 26 adjacent cell IDs for each cell, of size $27\,T$, where T is the number of cells in the box. Then, *ParticlesInCells* is bound to texture memory to take advantage of caching. Later for the entire simulation run, values will be read through texture fetches. Using texture memory is efficient for this purpose, especially since one thread block will be reading the same cell indices for the entire kernel call, allowing for a high rate of cache hits.

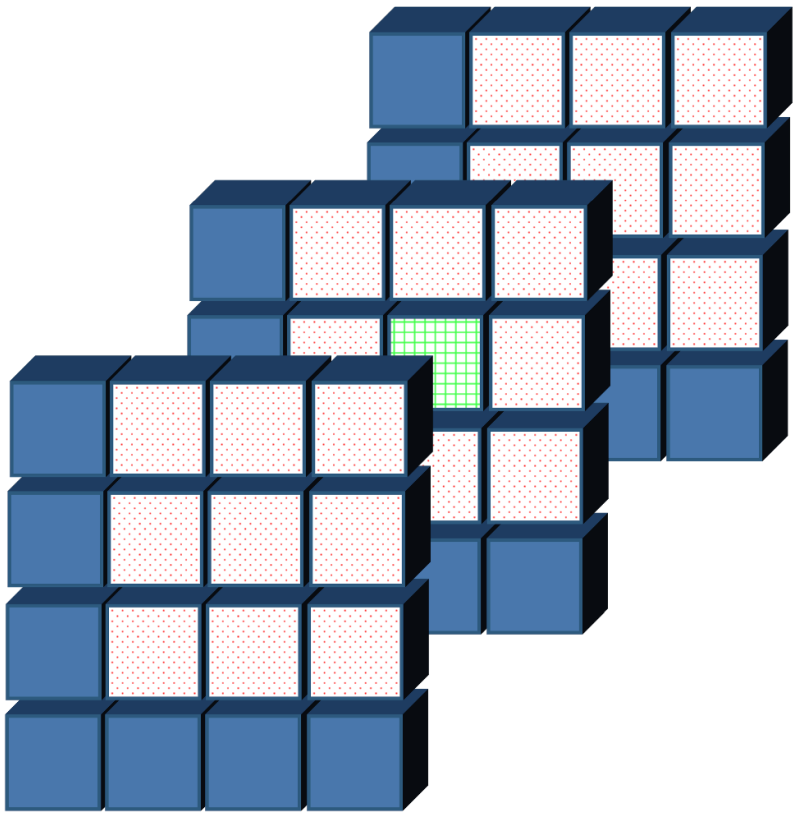


Figure 2: 26 cells adjacent to a particle in all axes.

Algorithm 6 CalculateParticlesContributionTM function (cell list 9×3)

```
1: Input: Three cells per block
2: //Initialize shared memory (x3)
3: //For the old position of the particle
4: //Find SourceCurrentCell
5: //Fetch three neighboring cells from texture
6: if Non-empty cell then
7:       //For each ParticleID in a fetched cell
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 new position of the particle
17: //Find DestCurrentCell
18: //Fetch three neighboring cells from texture
19: if Non-empty cell then
20:      //For each ParticleID in a fetched cell
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 ΔE to caller via global memory
```

Algorithms 6 and 7 show how the cell list implementation is used for finding ΔE for both particle displacement and particle insertion/deletion, respectively.

The process of placing particles in cells is called *binning* the particles. The binning process involves looping through the N particles and placing each into one of the T cells. The particle binning is done when we first construct the simulation box and generate the initial locations of all particles. When a particle is added to a cell, an atomic operation is performed on the relevant cell counter in order to avoid a race condition, so that each particle added to a cell is placed in a unique array location. Since atomic operations have become faster with the Fermi and Kepler architectures, and since this is done just when the system configuration is initialized, the overhead is minimal.

3.2.2 Cell Size

Using the notation in figure 1, we select S to maximize the integer $T = L/S$ with the constraint that $S \geq r_{cut}$. For instance, if $r_{cut} = 2.5$ and $L = 23.9$, then $S = 2.656$ with $T = 9$ cells per dimension. If $L < 3\,r_{cut}$, we set $S = L/3$. The use of periodic boundary conditions means that the entire simulation box will always be contained in these 27 cells, so cells smaller than r_{cut} are acceptable.

Algorithm 7 CalculateParticlesContributionTPT function (cell list 9×3)

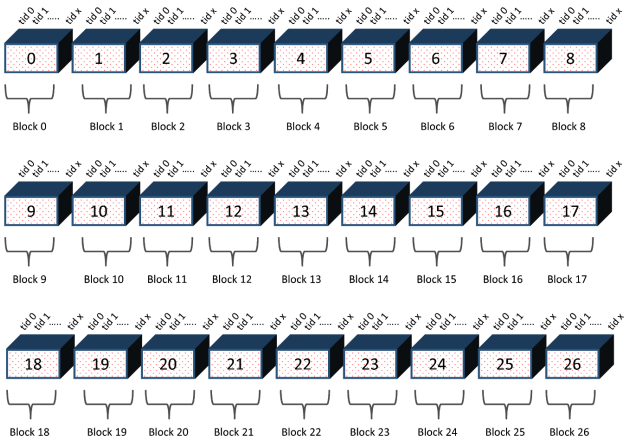
```
1: Input: Three cells per block
2: //Initialize shared memory (x3)
3: //For the selected particle (deletion) or
4: // selected position (insertion)
5: //Find CurrentCell
6: //Fetch three neighboring cells from texture
7: if Non-empty cell then
8:       //For each ParticleID in a fetched cell
9: 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

A fixed size array of placeholders for every cell has been used to support cell lists in our code. This parameter depends on the density of the simulation box, the radial cutoff, and the minimum distance between particles. A larger array size means extra wasted memory locations. On the other hand, a small array size might prevent valid configurations from being realized. Our implementation considers an array size large enough to encompass all particles within range to avoid overflow. The code also outputs an error message if array overflow would occur and the simulation must then be rerun with a larger array size. To avoid this problem, we selected sufficiently large array sizes through experimentation. The number of particles in a cell for a simulation with a small cutoff can be limited to 48 particles per cell. For cutoffs greater than 4.0σ, we require a limit of 96 particles per cell.

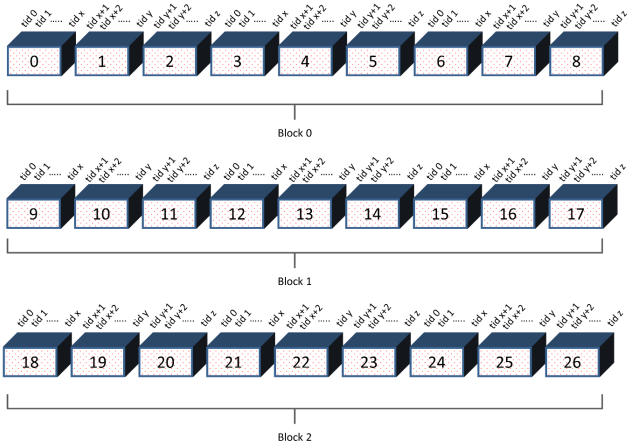
Note that move attempts where a particle stays in the same cell can be processed more efficiently. Whenever the particle moves within the same cell, the neighboring cells are the same for both the old and new locations for this particle. Hence, calculating the pairwise energy for both the old and the new location will consider the same set of particles. The coordinates of these particles are cached when calculating the energy at the old location and these cached coordinates can be reused when recalculating the energy at new location. Furthermore, if the move is accepted, since the list of particles in the cell is unchanged, no updates need to be made to any cells.

3.2.3 Assigning Cells to Blocks

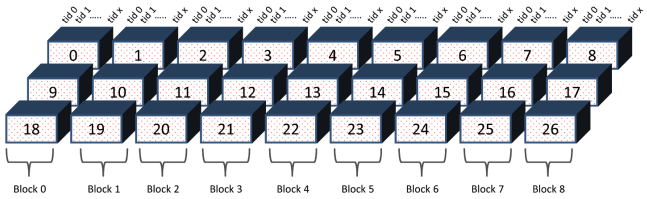
To the best of our knowledge, previous implementations of cell lists on the GPU have used a mapping where each cell is assigned to a different thread block. In MD simulations, where all particles are moved at the same time, it is more efficient to assign one thread per particle and one block per cell to take advantage of caching all 26 neighboring cells. However, this is not necessarily true for MC simulations where only one particle moves in each simulation step, and more blocks need more synchronization through atomic operations on global memory. Our implementation considers multiple options, with differing numbers of cells per block and reports the performance



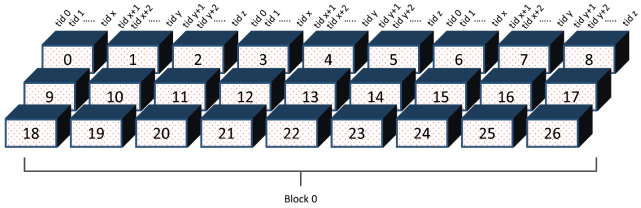
(a) One cell per block



(b) Nine cells per block



(c) Three cells per block



(d) Twenty-seven cells per block

Figure 3: Different methods of assigning cells to thread blocks.

differences. This is depicted in figure 3, where each thread block may compute 1, 3, 9, or 27 cells.

One Cell per Block This is the most straightforward implementation expressed in figure 3(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 best.

Three and Nine Cells per Block Due to the nature of the MC simulation, only one particle is examined each simulation step. This leads to relatively low device utilization. In figures 3(b) and 3(c), two different ways of GPU resource management are considered, one with nine cells per block and the other with three cells per block, respectively. In both cases, a separate thread is assigned to each particle. These two options use more total local memory per kernel call, with less need for block synchronization.

27 Cells in One Block Figure 3(d) shows the option of assigning all 27 cells to one block. Due to the many threads in this block with this alternative, more shared memory and other GPU resources are required. This also means that only one SM is used to compute the entire simulation and the other SMs are idle. This option is useful if one wishes to run several large simulations concurrently.

3.2.4 Assigning Threads to Particles

Previous work has studied the performance of one cell per block and one thread per particle [22, 23]. However, for very large systems that were too large to be simulated prior to this work, it is worthwhile to study the performance of assigning multiple particle per thread. An algorithm has been developed to assign multiple particles from more than one cell to the same thread. For example, for a kernel of nine blocks, if each block has been called with one thread per particle in a cell, then each thread will calculate the pairwise energy for a particle from three different cells ($9\times1\times3$). Another example is when only three blocks are called and each thread is calculating one particle from nine different cells ($3\times1\times9$).

3.2.5 Cell List Implementation Details

Incorporating the cell list implementation into the CUDA implementation described in section 3 requires the modification of 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, unless the source and destination cells are the same. 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 ID is appended to the destination cell and the counter for that cell is incremented. In addition, the maximum number of particles

per cell is used by the CPU so that the correct number of threads per cell are spawned. So, some information is copied from the GPU to the CPU after each move.

3.3 Microcell List Implementation

Although we show in section 4 that, for large enough problem sizes, a traditional cell list implementation outperforms a GPU implementation without cell lists, there are a few drawbacks to this approach. First, the maximum number of particles in a cell can vary with system parameters such as density and cutoff, so we either use extra memory to store each cell's data or we need to make run-time adjustments based on input parameters, which can make code optimization difficult. Second, efficient load balancing of the threads is difficult, because the number of particles in a cell could differ significantly across cells. Finally, the 27 cells encompass a volume significantly larger than the volume of interest. For example, with a cutoff of 2.5σ and a cell size of 2.75σ per dimension, the sphere surrounding a particle has a volume of 65.45, but the volume of the corresponding cells is more than 561.5. In general, the cell dimensions will exceed the cell size by a small amount because the simulation box must be subdivided into cells of the same size. Even if an exact cutoff of 2.5σ is possible, the corresponding volume is 421.875, which means less than 16% of the volume being processed in the cells could hold a particle within the cutoff.

Dividing the volume of the box into smaller cells, for instance, with a cell size of $r_{cut}/2$, partially addresses some of these drawbacks, but not all of them. Instead, we propose a novel and highly efficient cell list structure that shows *better performance for all problem sizes*. The simulation box is partitioned into microcells, each of which has a volume of σ^3, except for cells on the boundary, which could be smaller. For instance, if the volume of the simulation box is $50.23\sigma^3$, then the last cell along an axis will be 0.23σ in that dimension. We then assign each microcell to a unique thread. This organization has several advantages:

- The total volume being processed is smaller. Even allowing for boundary cases, only 343 ($7 \times 7 \times 7$) threads are needed for a cutoff of 2.5σ or 2.75σ.
- The microcell that contains a particular particle can be computed, without any division, directly from the integer portion of each coordinate.
- The number of threads per block is fixed and independent of the number of particles in the box, so we do not need to copy results to the CPU prior to each kernel call.
- The specific microcell assigned to each thread in the cube of cells can be mapped directly from the thread IDs by defining an $n \times n \times n$ dim3D block of threads for the kernel call.
- Because of the physical properties of the system, we can guarantee that there cannot be more than five particles in a cell. So, we can bound the maximum cell contents regardless of the input parameters.
- The load balancing is straightforward. For typical densities of 0.5 or 0.6 particles per microcell, we expect on average about half the microcells to have no particles and the other half to have one particle. So, processing is relatively fast.
- Since only one thread is used per microcell, the entire move runs in one block, eliminating all inter-block synchronization.

3.3.1 Building the Initial Microcell List

The initial cell list is created when the system, including the particle positions, is initialized. There are two arrays used for the cell list. One contains a counter of the number of particles in a cell. This array is initialized to zero using cudaMemset() and updated when a particle is added to a cell. There is also an array that contains the particle numbers, indices into the array of particle coordinates. This array can hold a maximum of five particles per cell. To improve memory coalescing, the array is organized so that we store the first particle of each cell, followed by the second, etc. In other words, if there are 1000 cells, then array location 0 holds the index of the first particle in cell 0, array location 1000 holds the index of the second particle in cell 0, etc. Atomic operations are used when adding a particle to a cell. However, note that since the cells are small and the density is less than 1.0, there is typically at most one particle in a cell at a time, so an atomic operation by one thread does not block another thread, since they are accessing different array locations.

3.3.2 Assigning Microcells to Threads

As mentioned above, the particle coordinates are used to compute its cell. The thread ID has three components: threadIdx.x, threadIdx.y, and threadIdx.z. Each corresponds to a unique cell in the cube of microcells, with the specific cell determined as an offset in each dimension from the cell of the particle. For instance, when processing a particle in cell $(7, 6, 5)$ and using a $7 \times 7 \times 7$ cube of microcells, cell $(4, 3, 2)$ is assigned to thread $(0, 0, 0)$ and cell $(10, 9, 8)$ is assigned to thread $(6, 6, 6)$. These simulations use periodic boundary conditions, so the code wraps around the box if necessary. The maximum block size used is 512 ($8 \times 8 \times 8$) threads. For cutoffs greater than 3.75σ, there are more than 512 microcells, so the threads iterate over multiple microcells. This allows the code to support arbitrarily large cutoffs. There is some loss of efficiency, so a different implementation could be considered for problems that use a cutoff $\geq 4.0\sigma$, although this is uncommon.

3.3.3 Microcell List Implementation Details

The CPU determines which move to perform: displacement, insertion, or deletion. The CPU then invokes the appropriate GPU kernel. The GPU uses a single block to evaluate the move and either accept or reject the move. Results do not need to be exchanged between the CPU and GPU except for periodic checkpointing of the system status and at the end of the simulation, except for some general statistics so that properties such as the average energy and average pressure can be calculated.

When a move is accepted, a particle ID may need to be inserted into a microcell and a particle ID may need to be removed from a microcell. Inserting an ID is straightforward; the location where the ID is inserted into the particle array is determined by the number of particles already in the cell and then the number of particles in the cell is incremented. Deleting an ID is simple when there is only one particle in a microcell; just decrement the particle counter for that cell. Because the densities are generally less than 1.0, this is the typical case. If there are multiple particles in a cell, then we need to scan through the IDs in the list and replace the deleted particle with the ID of the last particle in the list. Of course, this will still be faster than with the traditional cell list, since there are fewer particles in a microcell.

3.3.4 Further Optimizations

As shown in section 4, this novel cell list implementation offers significant performance improvements over both the code without cell lists and the version that uses traditional cell lists. Now, we describe further modifications to the code to fully exploit the advantages of the microcell list properties. To provide a fair comparison, we retained both the optimized and unoptimized versions and report results for both versions. Note that the unoptimized version uses a slightly different design, where only one microcell is assigned to each thread, unlike what is described above. So, the unoptimized version is inherently more efficient for some larger cutoffs, but cannot process cutoffs larger than 4.25σ. Even so, the optimized version is faster than the unoptimized version for all cutoffs. The primary optimizations are:

- Since the block size is fixed, we create a single kernel call that iterates over multiple moves. This is based on an input parameter that specifies how often checkpointing of the system is done. For the reported results, this corresponds to 10,000 moves. So, we are replacing 10,000 kernel calls with one kernel call. Consequently, things like average energy and average pressure are now computed on the GPU.
- Since only a single block is used for each move, no global memory is used for storing the results of the energy calculations. Instead, these are returned using parameters.
- The __launch_bounds__() compiler directive is used to provide a hint to the compiler that we are running just one block of at most 512 threads. This enables more compiler optimization of the code.
- The cudaDeviceSetCacheConfig() function is used to allocate more L1 cache and less shared memory for all the functions except CalculateTotalEnergy(). Since these other functions have just one block, dedicating more on-chip memory to cache improves performance.

As shown in the next section, these optimizations, along with many minor tweaks to the code, have resulted in significant performance improvements. Almost all of these optimizations were further improvements that were enabled by the use of the microcell list implementation.

4 Performance Results

In this section, we present a two-part performance evaluation of the cell list implementations of the parallel Monte Carlo simulation for the grand canonical ensemble. The first part of the simulation study uses CUDA toolkit 4.2 [17] and evaluates the end-to-end application wall clock time without traditional cell lists against a single core CPU implementation. The serial and CUDA implementations have been executed on a PC with an Intel i5-2500k CPU that has 8 GB of RAM running Linux kernel build 2.6.32 and compiled to a 64-bit executable with the Intel 13.0.0 compiler. Parallel results are collected from running the code on the same machine using a NVIDIA GeForce GTX 480 graphics card. This card has 1.5 GB of global memory, 15 streaming multiprocessors with 32 cores each, and compute capability 2.0. All code has been compiled with the full optimization flag (-O3). The purpose of this comparison is *not* to compare the performance between a CPU and GPU, since we are using only one CPU core. Instead, it is intended to show that the CUDA code without cell lists is a reasonably efficient implementation upon which the cell list code improves.

Table 1: Execution time in seconds for serial and CUDA programs.

Particles	**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

All these measurements have used one million simulation steps (move attempts) with a cutoff (r_{cut}) of 2.5σ. Although one million simulation steps is enough to show the relative performance of the various codes, it is important to note that for larger problems sizes, scientifically valid results require runs of hundreds of millions or billions of steps. Therefore, it would be reasonable to increase the difference in execution times by two or three orders of magnitude to get a better idea of the potential time savings. Note that valid results are produced by both the serial and the CUDA code, since they are statistically equivalent to published results [19]. The Mersenne twister algorithm has been used to generate the pseudo-random numbers used in these simulations [13].

Table 1 presents 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 to maintain a fixed initial density of 0.6 particles per σ^3. For problem sizes of less than 4096 particles, no speedup has been achieved due to low utilization of the GPU. However, when the simulation box has 4096 particles, the CUDA code begins to outperform the serial code, achieving about 2 times speedup as shown. As the system size increases, more mathematical operations are executed and the speedup of the GPU code continues to increase. For the largest problem size, we see a 15.8 fold speedup. The primary reason smaller systems do not show any speedup is because the kernel call overhead for smaller systems exceeds the gain of parallelism. Since MC simulations move only a single particle at a time, 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 expose enough parallelism to hide the overhead of kernel calls for such large systems.

In section 3.2, we described our implementation of the traditional cell list. Many different mappings of cells to blocks of the cell list algorithm have been evaluated. Table 2 explains the meaning of the corresponding notation.

From the results in table 3, we can see that when a kernel uses either 9 or 27 blocks, the best performance is achieved with a speedup of 8.32 and 8.31, respectively. In the case of 27 blocks, only one cell is handled by each block. The slightly extra overhead of synchronization and reduction for

Table 2: Legend of blocks, cells, and threads per kernel call.

Notation	Blocks/Kernel	Cells/Block	Threads/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

Table 3: Speedup of different cell list implementations over no cell list CUDA code.

Particles	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

additional blocks may make the 9×3 version faster than the 27×1 version, although the difference is not statistically significant in our results. For this reason, we use the 27×1 version, which assigns each cell to a different block, in the remainder of our performance comparisons. A more detailed performance evaluation of these algorithms can be found in [9].

The second part of the simulation study compares the performance of the various GPU implementations of the code. This illustrates the improvement offered by our novel microcell list implementation. These simulations were conducted on two different systems. The first system is the same PC as before, but now running CUDA 6.0 [17] on an NVIDIA Tesla K40c. The K40c has 12 GB of global memory, 15 streaming multiprocessors with 192 cores each, and compute capability 3.5. This is the highest-end NVIDIA GPU available when these simulations were run. The second system is a six-core AMD Phenom II X6 1045T CPU that has 12 GB of RAM running CUDA 6.0 [17] on an NVIDIA GeForce GTX 550 Ti. The GTX 550 Ti has 1.5 GB of global memory, 4 streaming multiprocessors with 48 cores each, and compute capability 2.1. Both PCs are running Windows 7 and the programs were compiled to 64-bit executables using MS Visual Studio 2010 with full optimization. These two systems will be referred to by their respective GPUs to distinguish between them when discussing the results.

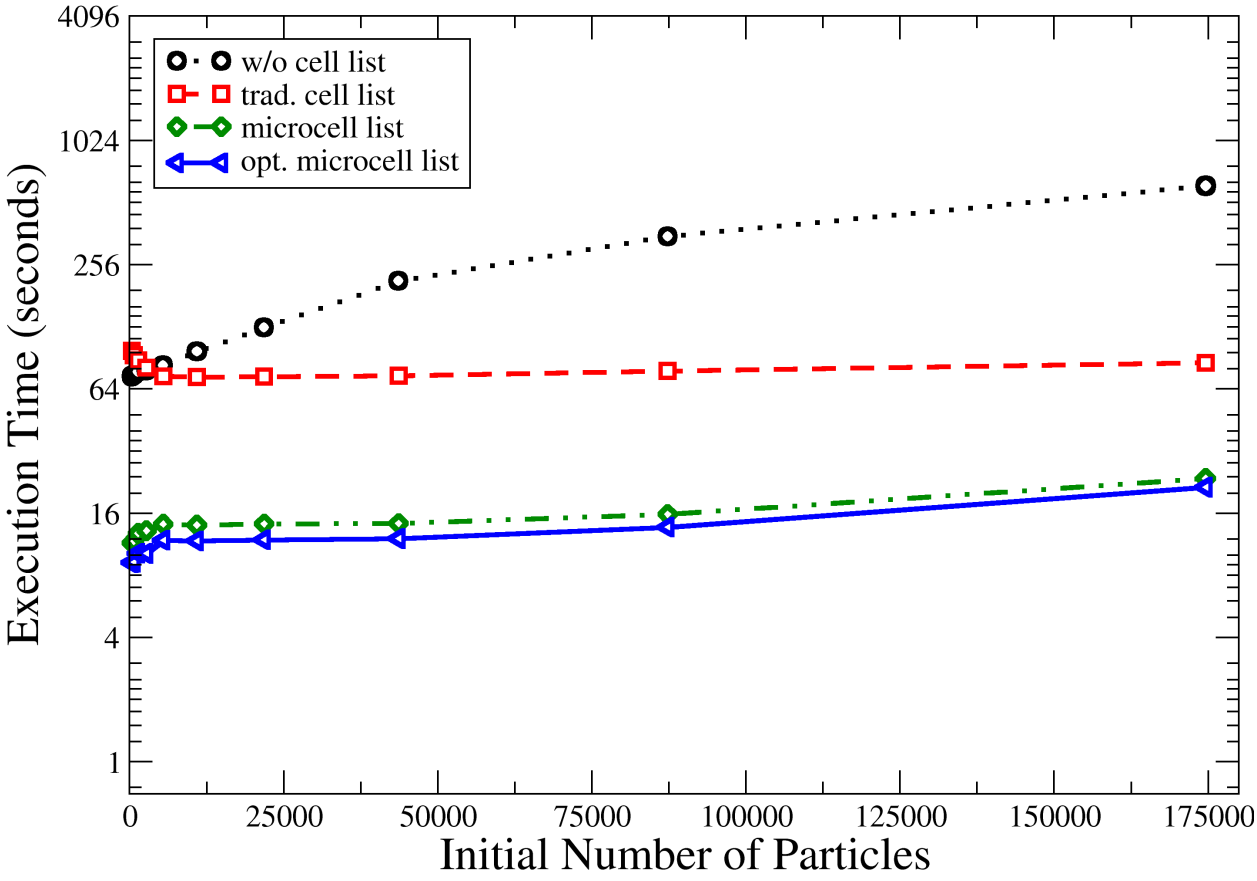


Figure 4: Performance of the GPU algorithms on the GTX 550 Ti.

Figures 4 and 5 are log-scale plots that show the results of four different GPU codes: the aforementioned code without cell lists, the aforementioned code with a traditional cell list, a version with the new microcell list, and an optimized version of the microcell list, running on the GTX 550 Ti and the K40c, respectively. All simulations were run for one million steps (move attempts) for different numbers of particles having a fixed initial density of about 0.67 particles per σ^3, with a cutoff (r_{cut}) of 2.5σ using a ratio of 30% displacement moves, 35% insertions, and 35% deletions. This initial density was chosen, since longer running experiments (not shown) for these configurations equilibrated at approximately this density. Hence, we able to obtain a better idea of the performance of these codes for long running simulations. Each point in the graphs is the average of five different runs with random seeds. The codes were validated against each other, since the code without cell lists had already been validated against the serial code and the literature. There was little variance in runtime among these five runs, always less than 2% from the average.

On both systems, the microcell list code shows significantly better performance for all problem sizes. The further optimizations to the microcell list code show additional performance improvement. As shown in figure 6, the optimized microcell list code running on the GTX 550 Ti is from 8 to 28 times faster than the code without cell lists, with more speedup for larger problem sizes. It is also more than four times faster than the code with a traditional cell list, and between 10% and 25% faster than the unoptimized microcell list code. As shown in figure 7, on the K40c, which

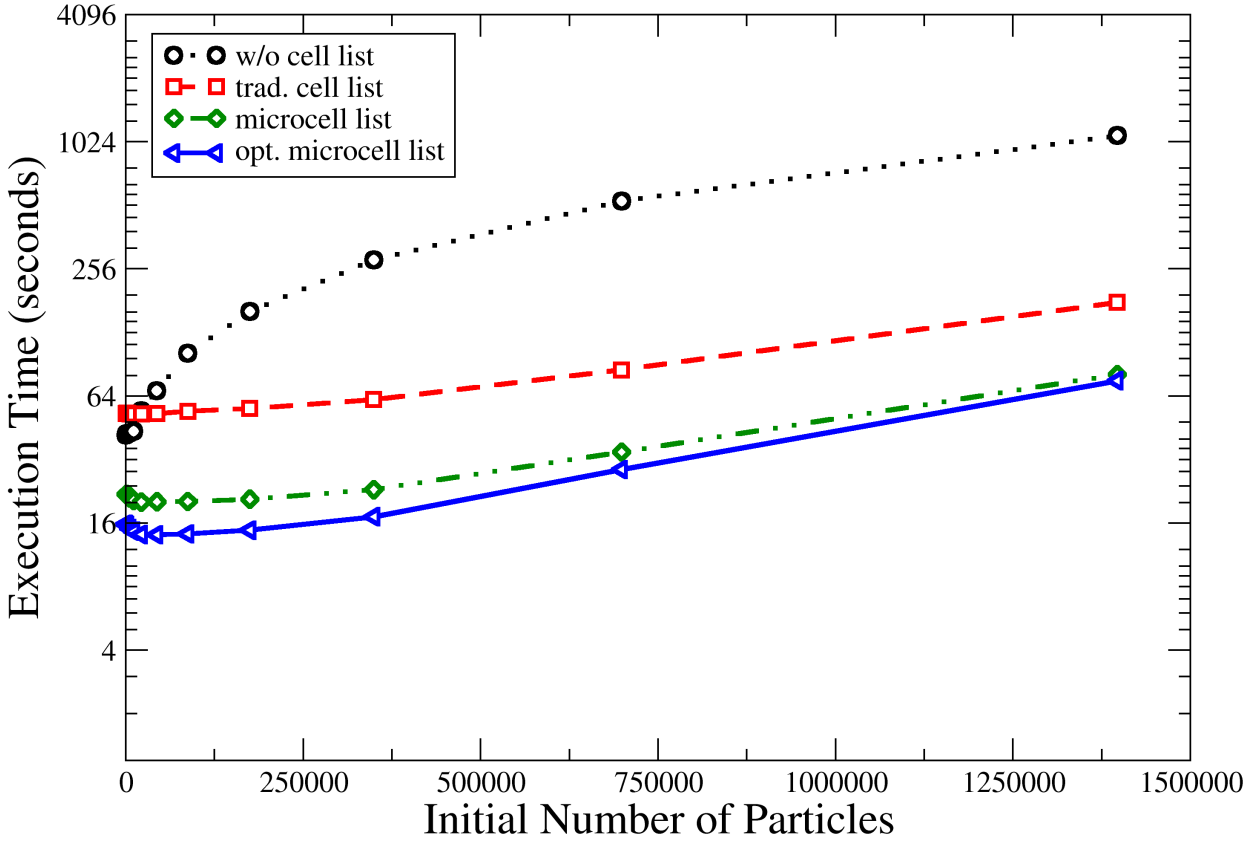


Figure 5: Performance of the GPU algorithms on the Kepler K40c.

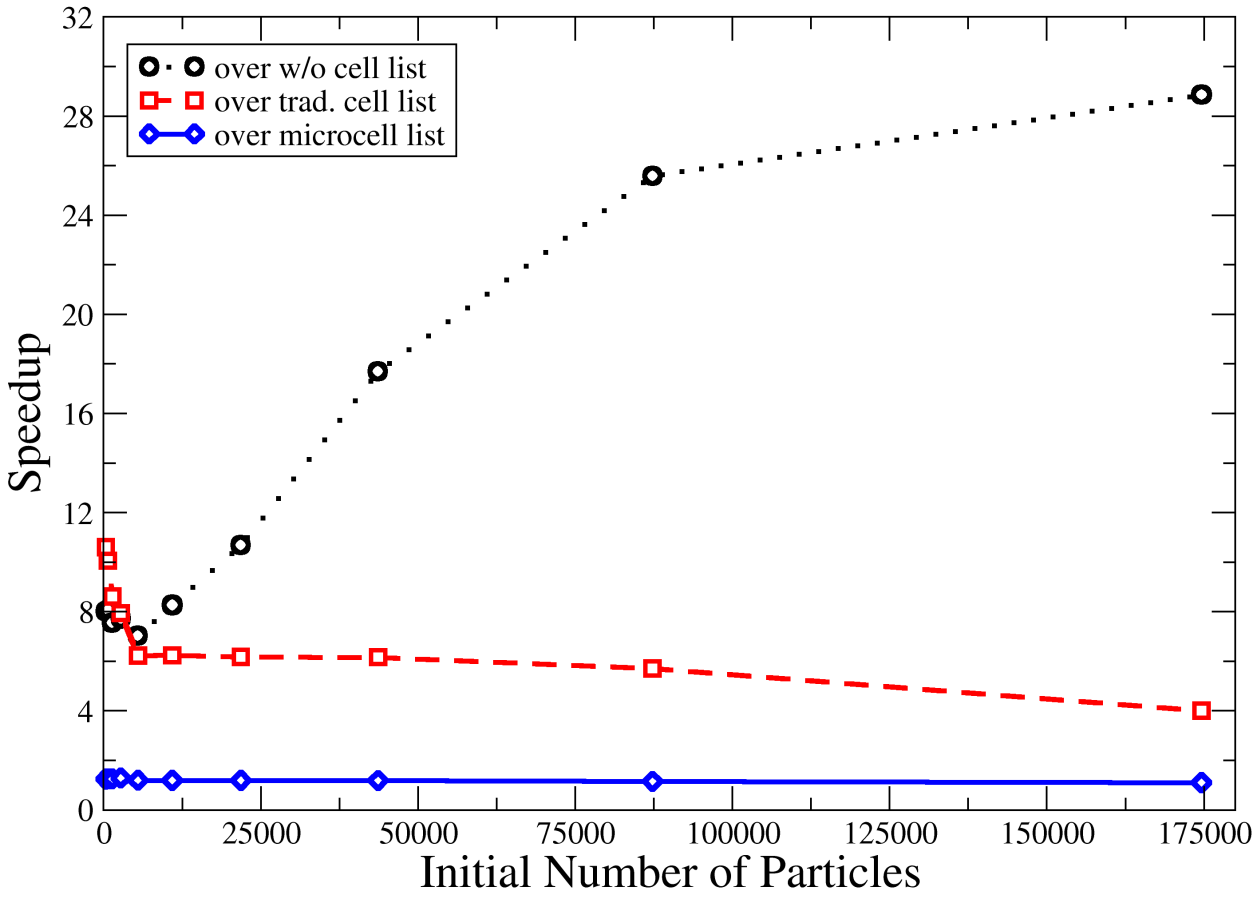


Figure 6: Relative speedup of the optimized microcell list on the GTX 550 Ti.

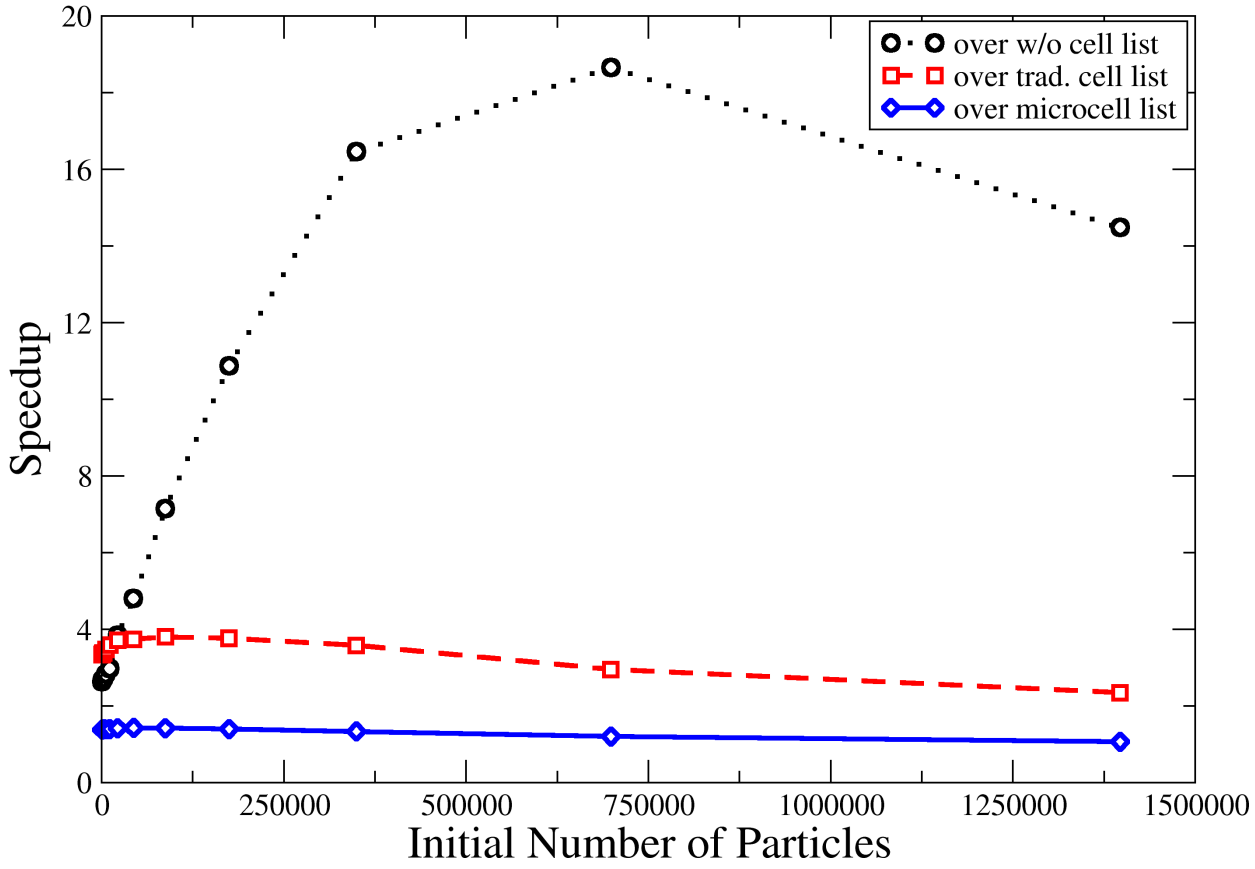


Figure 7: Relative speedup of the optimized microcell list on the Kepler K40c.

has much more computational resources, the optimized cell list code is up to almost 19 times faster than the code without cell lists, with again more speedup for larger problem sizes. It is also 2 - 4 times faster than the code with the traditional cell list code, and 20% to 43% faster than the unoptimized microcell list code, except for the case with over one million particles, when it is only 7% faster. In the comparisons of the cell list codes, the speedup is more for the smaller problem sizes. The reason is that even though the total system energy is calculated only once, the overhead for this $O(N^2)$ operation becomes an increasingly large portion of the execution time, especially when each move has been optimized to run so quickly.

An interesting observation is that after the system is initialized and the initial system energy is calculated, each subsequent kernel is launched with a single block of 512 threads. Because the GPU does not partition blocks, only a single SM of the GPU is being used. This means that in the case of the K40c, the microcell list code is running all the moves using at most 1/15th the resources of both the version without cell lists and the version with the traditional cell list. Even so, the microcell list code is able to achieve significantly better performance. In essence, the microcell list algorithm allows us to achieve a more effective schedule [12] for computing the energetic decomposition on the GPU. This further demonstrates the potential benefits of using an algorithm tuned specifically to the GPU architecture.

As the cutoff increases, one would expect the relative performance of the cell list codes to decline. To test this hypothesis, we ran simulations with the same parameters as before, except that we fixed the initial configuration to have 87296 particles and a volume of 128K σ^3, and varied the cutoff from 2.5σ to 4.25σ. These results are shown as log-scale plots in figures 8 and 9. On both systems, we observe the expected slow decline in the speedup of the optimized microcell list code compared to the other three versions of the GPU code. However, for all cutoffs, the optimized microcell list code remains faster than the other three codes. So, even for systems with larger cutoffs, the microcell list code is a more efficient algorithm.

5 Conclusions and Future Work

Designing efficient algorithms for a parallel architecture can offer significant performance advantages over just porting an existing sequential algorithm. Our proposed cell list structure is an example of such an algorithm, which provides an efficient mapping of cells to the threads of a manycore architecture. The design of the microcell list allows simple calculations of in which cell a particle particle resides and which neighboring cells need to be processed. Our future work will examine alternative ways of choosing the neighboring cells and eliminating cells beyond the cutoff in order to further reduce the number of microcells that our functions consider.

We first compare an optimized serial code that runs on a single CPU core with our straightforward GPU implementation to show that the GPU implementation without cell list has reasonable performance. We then compare three different implementations of cell list on the GPU: an implementation using traditional cell lists, an implementation using our new microcell list, and an optimized version of the microcell list code.

The microcell list code shows significant performance improvements for all problem sizes even for large cutoffs. The further optimizations that this algorithm allows produce an even more efficient code. Besides achieving superior performance, the microcell list structure simplifies the computations and eliminates enough unnecessary comparisons to allow us to achieve this performance using just one of the streaming multiprocessors (SM) on the GPU. This offers the opportunity to

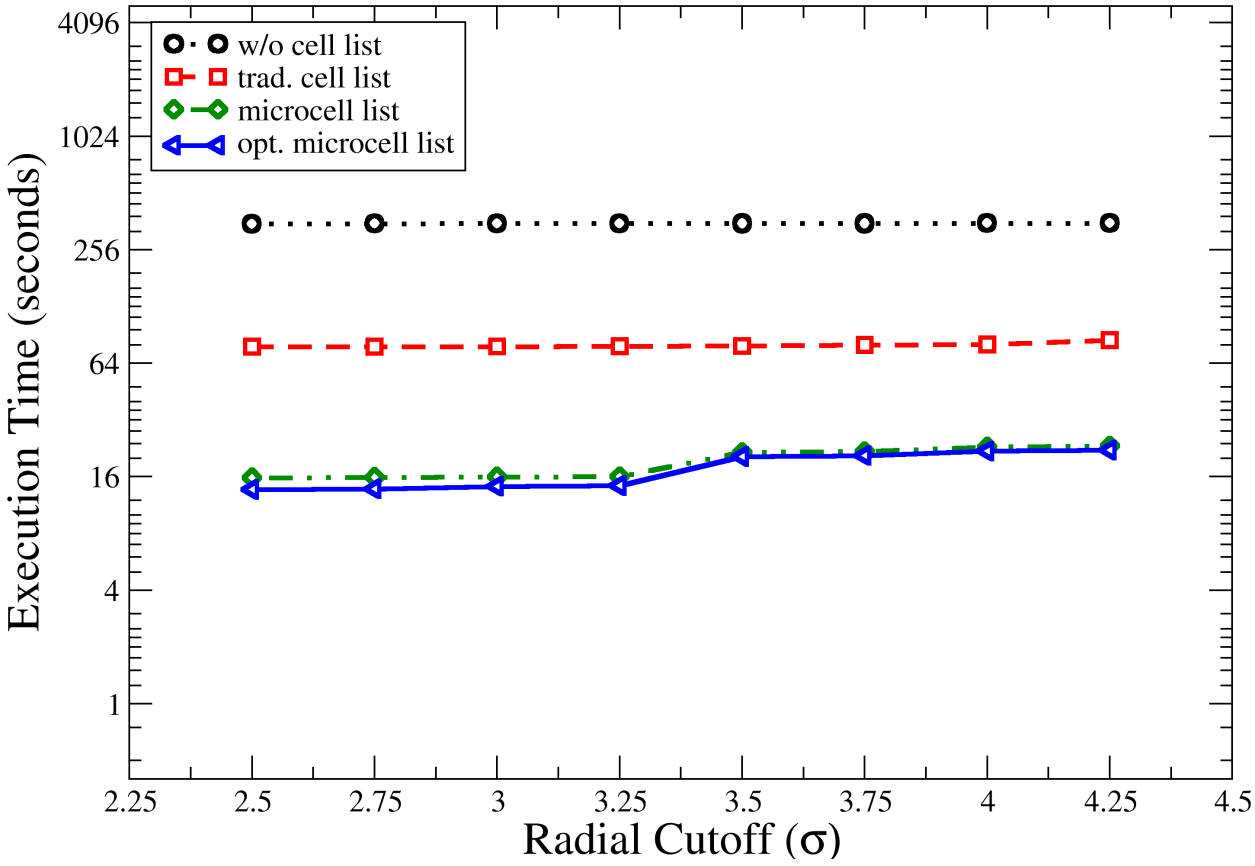


Figure 8: Performance impact of different cutoffs with about 85K particles on the GTX 550 Ti.

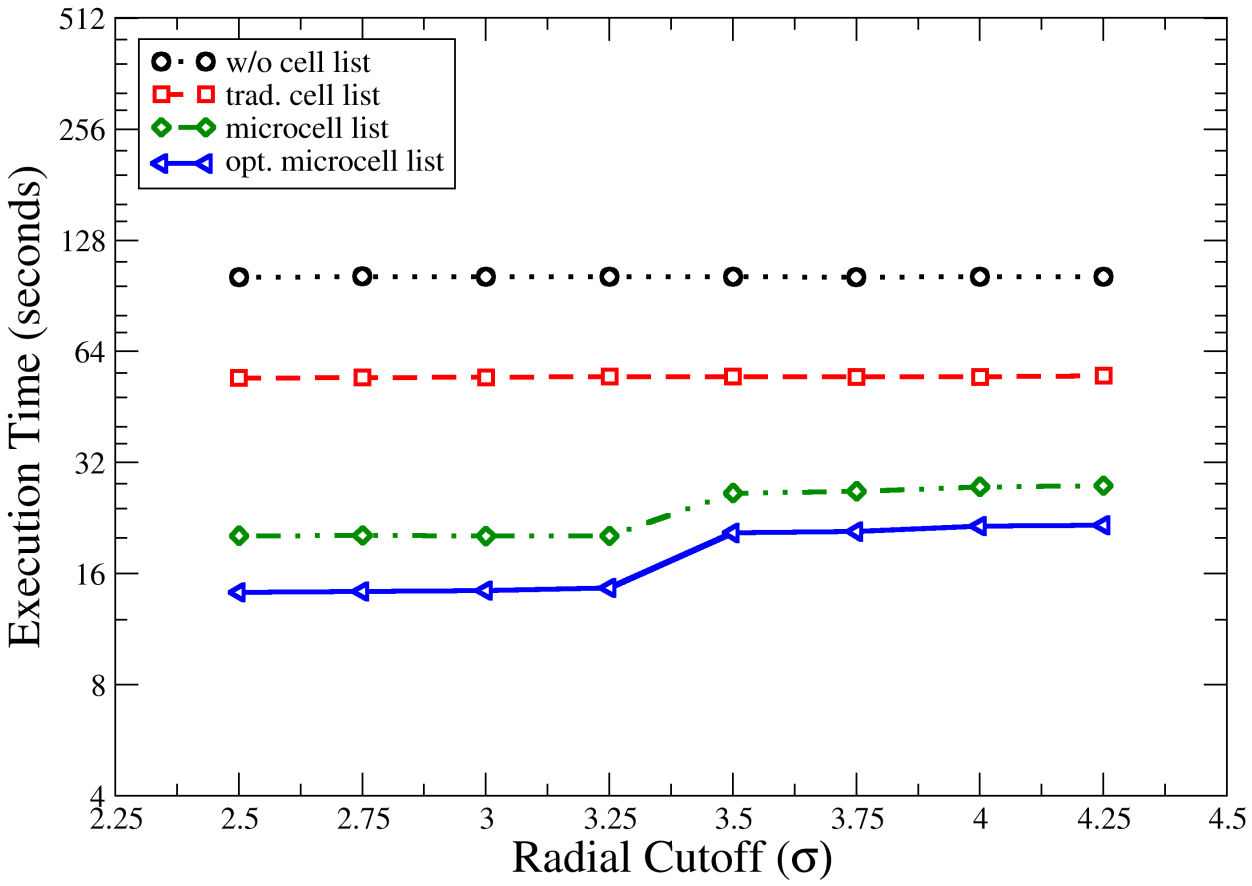


Figure 9: Performance impact of different cutoffs with about 85K particles on the Kepler K40c.

run multiple large-scale simulations on a single GPU simultaneously. Another option is to run multiple moves concurrently, each on a separate SM, and keep the first one that is accepted. We will investigate both of these options in our future work.

The grand canonical ensemble has only three types of moves, and each of those moves changes at most one particle in the system. For this reason, the microcell list evolves slowly during the simulation. For some other ensembles, such as the Gibbs ensemble [11], volume transfer moves allow the volume of the simulation boxes to increase or decrease, which requires the pairwise energy of all the particles to be recalculated. In addition, if the volume changes, then the cell list needs to be rebuilt because all of the particles in the box move.

Since recalculation of the system energy and pressure are relatively frequent requirements with the Gibbs ensemble, we plan to develop an efficient microcell list version of the CalculateTotalEnergy() function. Our future work will also investigate how to rebuild the cell list efficiently. Although this adds overhead to the simulation, it is likely that the microcell list will still offer significant performance improvement. Volume moves are not common and only about half of volume move attempts are accepted. So, the volume changes, on average, about once every 200 moves. This will allow us to amortize the cost of rebuilding the microcell list over many moves.

6 Acknowledgments

The authors thank Micah Bojrab for detailed feedback that significantly improved the presentation of the paper. This material is based upon work supported by the National Science Foundation under Grant No. CBET-0730768 and OCI-1148168, and Wayne State University's Research Enhancement Program (REP). We also thank NVIDIA for donating most of the graphics cards used in this study.

References

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

[2] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, 1987.

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

[4] I. Beichl and F. Sullivan. The Metropolis Algorithm. *Computing in Science Engineering*, 2(1):65–69, Jan. 2000.

[5] F. Brugè. Systolic Calculation of Pair Interactions Using the Cell Linked-Lists Method on Multi-processor Systems. *Journal of Computational Physics*, 104(1):263–266, Jan. 1993.

[6] K. B. Daly, J. B. Benziger, P. G. Debenedetti, and A. Z. Panagiotopoulos. Massively parallel chemical potential calculation on graphics processing units. *Computer Physics Communications*, 183(10):2054–2062, Oct. 2012.

[7] D. Frenkel and B. Smit. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, San Diego, second edition, 2002.

[8] G. S. Grest, B. Dünweg, and K. Kremer. Vectorized link cell Fortran code for molecular dynamics simulations for a large number of particles. *Computer Physics Communications*, 55(3):269–285, 1989.

[9] E. Hailat. *Advanced Optimization Techniques for Monte Carlo Simulation on Graphics Processing Units*. PhD thesis, Wayne State University, Aug. 2013.

[10] J. Kim, J. M. Rodgers, M. Athènes, and B. 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.

[11] L. Loyens, B. Smit, and K. Esselink. Parallel Gibbs-ensemble simulations. *Molecular Physics*, 86(2):171–183, 1995.

[12] D. Lutz and D. N. Jayasimha. Do Fixed-Processor Communication-Time Tradeoffs Exist? *Parallel Processing Letters*, 5(2):311–320, June 1995.

[13] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, Jan. 1998.

[14] W. Mattson and B. M. Rice. Near-neighbor calculations using a modified cell-linked list method. *Computer Physics Communications*, 119(2-3):135–148, June 1999.

[15] 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(6):1087–1092, 1953.

[16] C. A. Navarro and N. Hitschfeld. Improving the GPU space of computation under triangular domain problems. *CoRR*, 2013.

[17] NVIDIA. *CUDA C Programming Guide*, 6 edition, Feb. 2014.

[18] 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.

[19] J. J. Potoff and A. 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.

[20] A. J. Proctor, C. A. Stevens, and S. S. Cho. GPU-Optimized Hybrid Neighbor/Cell List Algorithm for Coarse-Grained MD Simulations of Protein and RNA Folding and Assembly. In *Proceedings of the International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, BCB'13, pages 633–640, New York, NY, USA, 2013. ACM.

[21] 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.

[22] J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, and K. Schulten. Accelerating Molecular Modeling Applications with Graphics Processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

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

[24] L. Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Phys. Rev.*, 159(1):98–103, 1967.

[25] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU Architecture. *IEEE Micro*, 31(2):50–59, March-April 2011.

[26] Z. H. Yao, J.-S. Wang, G.-R. Liu, and M. Cheng. Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. *Computer Physics Communications*, 161(1-2):27–35, 2004.