GPU-Optimized Hybrid Neighbor/Cell List Algorithm for Coarse-Grained MD Simulations of Protein and RNA Folding and Assembly

Andrew J. Proctor
Wake Forest University
Department of Computer Science
1834 Wake Forest Road
Winston-Salem, NC 27109
procta06@gmail.com

Cody A. Stevens
Wake Forest University
Department of Computer Science
1834 Wake Forest Road
Winston-Salem, NC 27109
stevca9@gmail.com

Samuel S. Cho
Wake Forest University
Departments of Physics and Computer Science
1834 Wake Forest Road
Winston-Salem, NC 27109
choss@wfu.edu

ABSTRACT

Molecular dynamics (MD) simulations provide a molecular-resolution view of biomolecular folding and assembly processes, but the computational demands of the underlying algorithms limit the lenth- and time-scales of the simulations one can perform. Recently, graphics processing units (GPUs), specialized devices that were originally designed for rendering images, have been repurposed for high performance computing, and there have been significant increases in the performances of parallel algorithms such as the ones in MD simulations. Previously, we implemented a GPU-optimized parallel neighbor list algorithm for our coarse-grained MD simulations, and we observed an N-dependent speed-up (or speed-down) compared to a CPU-optimized algorithm, where N is the number of interacting beads representing amino acids or nucleotides for proteins or RNAs, respectively. We had demonstrated that for MD simulations of the 70s ribosome (N=10,219), our GPU-optimized code was about 30x as fast as a CPU-optimized version. In our present study, we implement a hybrid neighbor/cell list algorithm that borrows components from the well-known neighbor list and the cell-list algorithms. We observe about 10% speedup as compared to our previous implementation of a GPU-optimized parallel neighbor list algorithm.

Categories and Subject Descriptors

J.2 [**Computer Applications**]: Physical Sciences and Engineering—*Chemistry, Physics*

General Terms

Algorithms, Design, Performance

ACM-BCB'13 September 22-25, 2013, Washington, DC, USA

Keywords

Verlet neighbor list, cell-list, performance analyses

1. INTRODUCTION

Molecular dynamics (MD) simulations are well established computational tools [1] that can characterize how biomolecules such as proteins and RNA assemble into well-defined configurations to perform its biological function [12, 6]. They can also be indispensible tools for studying when the assembly process results in misfolded states that correspond to protein aggregation diseases [10]. Although MD simulations have a long history of providing accurate, molecular-resolution descriptions of complex systems, the underlying algorithms are limited by the demanding computations required. It remains a challenge to develop and implement scalable algorithms that decrease the inherent computational demands of MD simulations.

In general, there are two sets of limitations of MD simulations, and they are problems of length-scale and time-scale such that there are limits on the size of the system and the amount of time one can perform these simulations. Ideally, one would use a detailed description of the biomolecule, and this can be accomplished using quantum mechanical calculations. However, one must include a description of all of the electrons in the biomolecule, which may actually be irrelevant to the problem at hand. Others use empirical force field MD simulations where classical mechanics is used instead by approximating biomolecules at an atomistic resolution [16, 4]. Still others use coarse-grained MD simulations where groups of atoms are approximated as beads [3, 15, 17]. By using a less detailed description of a biomolecule, we instead gain the ability to perform MD simulations of larger systems and on longer timescales.

Once the energy potential function for representing the biomolecule(s) of interest is determined, one must determine the rules for moving the biomolecule over time. Briefly, the MD simulation algorithm is as follows: given a set of initial positions and velocities, the gradient of the energy is used to compute the forces acting on each bead, which is then used to compute a new set of positions and velocities by solving Newton's equations of motion ($\vec{F} = m\vec{a}$) after a time interval Δt. The process is repeated until a series of sets of positions and velocities (snapshots) results in a movie trajectory [1].

The major bottleneck in the MD simulation algorithm is

the calculation of the forces that correspond to the pairwise long-range interactions between interacting beads. A common approach for minimizing the computations while retaining accuracy is to introduce a truncation method. Since the interactions between distant beads results in negligible interactions, one can partition the set of possible interactions and only compute those that are proximal to one another. In principle, distant "zero" forces could all be calculated to maintain perfect accuracy, but they could also be disregarded without significantly impacting the accuracy of the MD simulation. Among the different approaches include the well-known Verlet neighbor list and cell list algorithms [18, 1].

Since these interactions can be independently computed, MD simulation algorithms are particularly well-suited for implementation on graphics processor units (GPUs). As such, hybrid high performance computing platforms that include GPU accelerators, in addition to conventional multicore CPUs, have become important tools in scientific computing. For example, several studies recently demonstrated that GPU-optimized MD simulations, including the empirical force field MD simulation software NAMD [16] and AMBER [4] and the general purpose particle dynamics simulation software suites HOOMD [2] and LAMMPS [13], can significantly increase performance.

MD simulations lend themselves readily to GPUs because many independent processor cores can be used to calculate the independent set of forces acting between the beads in an MD simulation. For NVIDIA GPUs, software is written using the CUDA programming language, which allows programs to be written for execution on the GPU. Included are operations for data transfer so that floating point operations can be performed on the GPU. Once the data is there, a "kernel", the CUDA equivalent of a parallel function call, executes a set of parallel operations on the data. The kernel spawns a collection of threads that carry out the exact same instruction on its own subset of the data. These threads are extremely lightweight with very little creation overhead, and they are arranged into a hierarchy of thread blocks that are concurrently executed by SIMD multiprocessors [8]. The main bottleneck of a GPU program is the transfer of information between the CPU and GPU, and GPU-optimized algorithms must avoid this scenario by performing as much calculations on the GPU as possible before sending it back to the CPU. Although recasting the MD simulation software to the GPU can have significant performance gains, these efforts are often nontrivial because they require recasting traditional algorithms that were originally optimized for serial operations for parallel architectures or developing new ones. Therefore, there is a great need for new parallel, scalable MD simulation algorithms that can optimally take advantage of the GPU hardware.

In the present study, we begin with an overview of our coarse-grained MD simulation approach and describe a previously introduced a GPU-optimized algorithm for a parallel Verlet neighbor list algorithm that could be completely performed on the GPU using only parallel operations [9]. We had shown that our coarse-grained MD simulation code with the GPU-optimized parallel Verlet neighbor list algorithm performs comparably or better than a leading GPU-optimized general particle dynamics simulation software [14]. We then describe a parallel cell list algorithm, and we present a parallel hybrid neighbor/cell list algorithm that is specifically optimized for the GPU. Finally, we analyze its performance and scalability as compared with the parallel neighbor and cell list algorithms.

2. COARSE-GRAINED MD SIMULATIONS

The most basic physical description of a biomolecule is a potential energy function that includes the short-range connectivity of the individual components through a bond energy term and long-range interactions of the spherical components through attractive and repulsive energy terms. More sophisticated descriptions can include electrostatic charge interactions, solvation interactions, and other interactions. Regardless, the physical description of biomolecules essentially become spherical nodes that are connected by static edges that correspond to bonds and dynamic edges that correspond to long-range interactions [12, 6].

A very simple coarse-grained simulation model is the Self Organized Polymer (SOP) model, where each residue or nucleotide is represented by a single bead that is centered on the amino acid or nucleotide (at the C_α or C2' position) for proteins or RNA, respectively, thereby reducing the total number of simulated beads [7, 20, 9]. Recent studies have demonstrated that MD simulation of biomolecules using the SOP model reproduces experimentally measured mechanical properties of biomolecules with remarkable accuracy [20, 5].

In the SOP model representation, the potential energy function that describe biomolecules is as follows:

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

The first term is the finite extensible nonlinear elastic (FENE) potential that connects each bead to its successive bead in a linear chain where $k = 20\text{kal}/(\text{mol} \cdot \text{Å}^2)$, $R_0 = 0.2\text{nm}$, $r_{i,i+1}^0$ is the distance between neighboring beads in the folded structure, and $r_{i,i+1}$ is the actual distance between neighboring beads at a given time t.

The second term, a soft-sphere angle potential, is applied to all pairs of beads i and $i+2$ with separated by the distance $r_{i,i+2}^0$ in the folded structure to ensure that the chains do not cross.

The third term is the Lennard-Jones potential that describes van der Waals native interactions, which is used to stabilize the folded structure. For each bead pair i and j, such that $|i - j| > 2$, a native pair is defined as having a distance less than 8Å in the folded structure. If beads i and j are a native pair, $\Delta_{i,j} = 1$, otherwise $\Delta_{i,j} = 0$. The $r_{i,j}$ term is the actual distance between native pairs at a given time t, and $r_{i,j}^0$ is its distance in the folded structure.

Finally, the fourth term is a Lennard-Jones type repulsive term for van der Waals interactions between all pairs of

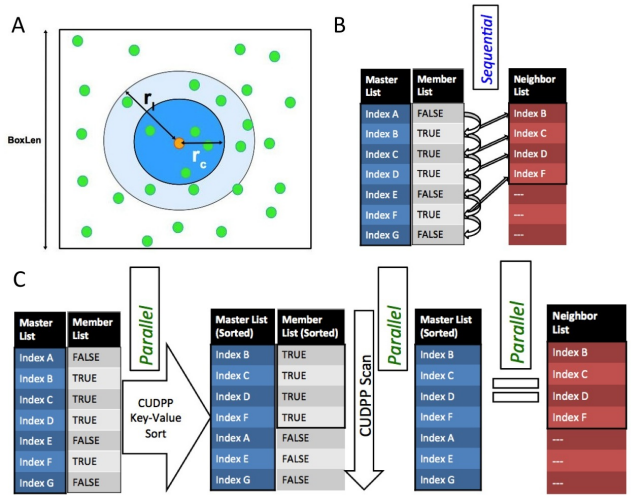

Figure 1: Illustration of the serial and parallel Verlet neighbor list algorithms. (A) In the original Verlet neighbor list algorithm, two lists are maintained: a neighbor list of interactions, with distance cutoff, r_l, is updated every N timesteps and a pair list that is a subset of the neighbor list where the distance cutoff is r_c. Only the interactions between beads in the pair list are computed. (B) Copying the entries of the Master List to to the Neighbor List is a sequential process usually performed by the CPU. (C) A parallel approach to copying the lists is presented. Step 1: Perform key-value sort on GPU using Member List as keys and Master List as values. This groups members of Neighbor List places them at the top of the Master and Member Lists. Step 2: Perform parallel scan to count the total number of TRUE values in Member List, determining how many entries are in Neighbor List. Step 3: Update Neighbor List by copying the appropriate entries from the Master list in parallel.

beads that are non-native, and $\sigma_{i,j}$, the radius of the interacting beads, is chosen to be 3.8Å, 5.4Å, or 7.0Å depending on whether the pair involves protein-protein, protein-RNA, or RNA-RNA interactions, respectively.

The most computationally demanding portion of the SOP model are the last two terms, which collectively include interactions between all long-range pairs of interactions. Trivially, we can halve the computations of the interactions by noting that each interaction between i and j is identical to the interaction between j and i, but the computational complexity remains the same. A common practice to minimize computations while retaining accuracy is to introduce a truncation method. Since the interactions between distant beads results in negligible interactions, one can partition the set of possible interactions and only compute those that are proximal to one another. In principle, these "zero" forces could all be calculated to maintain perfect accuracy, but they could also be disregarded without significantly impacting the simulation. It is important to note that the interactions in the Lennard-Jones potential scales as $O(N^2)$, which can be avoided using a truncation scheme such as a neighbor list or cell list algorithms as described below.

3. PARALLEL VERLET NEIGHBOR LIST ALGORITHM

In the original Verlet neighbor list algorithm, instead of calculating the force between every possible pair of long-range interactions, a subset "neighbor list" is constructed with beads within a "skin" layer radius of r_l. The skin layer is updated every n_{ts} timesteps, and the interactions within the cutoff radius r_c are computed at each timestep (Figure 1A). These interactions become members of the "pair list", which holds a subset of all interactions. The values of r_c and r_l are chosen as 2.5σ and 3.2σ, respectively as was done in Verlet's seminal paper [18]. With the neighbor list algorithm, the computations of the interactions become $O(Nr_c^3) \approx O(N)$, which becomes far more computationally tractable [18].

The original Verlet neighbor list algorithm is inherently serial because the generation of a neighbor list necessitates a serial operation (Figure 1B). From a list of all possible interactions between pairs of beads i and j, which we call the "Master List", a "Member List" is generated by determining whether a pair of beads are within a distance cutoff. If so, the subset of pairs of interactions that are within the distance cutoff are copied to a neighbor (or pair) list array. Since the first pair is placed at the top, then the second in the next place, and so forth, this is a sequential operation that would be optimal for a CPU but not on a GPU, so its implementation could suffer from a performance penalty if the data were transferred to the CPU to complete the operation.

To fully parallelize the algorithm to run on the GPU, Lipscomb et al. developed a novel GPU-optimized neighbor list algorithm that utilizes the key-value sort and parallel scan functionalities [9] (Figure 1C). The algorithm generates a neighbor (or pair) list using only parallel operations on the GPU that is identical to the one generated by the serial neighbor list and, in turn, avoids the memory transfer bottleneck of sending information to and from the CPU. In our MD simulation code, a Master List is used that indexes each pair of interacting beads i and j. We call a kernel to compute the distance between every pair of beads i and j that are each assigned an individual thread and label the interaction as "true" or "false" in the Member List depending on whether $r_{ij} < r_{cut}$. The interaction list also eliminates the need for a *for-loop* on the GPU, and a kernel with n threads is needed where n is the number of interactions in the list.

Once the Member List is generated, the first step of this parallel method is to perform a key-value sort on the data. A key-value sort involves two arrays of equal length: a keys array and a values array. Typically the keys are sorted in ascending or descending order with the corresponding entries in the values array (Master List) moving with the sorted entries in the keys array (Member List). When using the Member List as keys in the key-sort, the "true" values move to the top of the list and the "false" values move to the bottom. That is, the entries in the Master List that correspond to the indices of the interactions move along with their counterparts in the Member List that identifies the interaction as members of the neighbor (or pair) list. Next, the number of "true" entries are summed using a parallel scan operation to determine the number of entries to be copied from the Master List to the Neighbor List. Finally, that number is used to copy the first x entries of the Master List to the Neigh-

bor List, which can now be done in parallel. This algorithm clearly takes more steps to perform than the original Verlet neighbor list algorithm. However, since it can be performed entirely on the GPU through parallel operations, it avoids the performance bottleneck of transferring data between the CPU and GPU. Also, the Verlet neighbor list algorithm had to be recast into a form that is distinct and optimized for the GPU architecture.

4. PARALLEL CELL LIST ALGORITHM

Before we discuss our parallel cell list algorithm, we first review the original cell list algorithm [1]. Unlike the Verlet neighbor list algorithm, the cell list algorithm does not compute the distances between interacting pairs. Rather, the cell list algorithm works by dividing the the simulation box into many subdomains or "cells" as shown in Figure 2A. The beads are sorted into cells based on their x, y, and z coordinates. On a CPU, the cell list algorithm is typically implemented using linked lists to keep track of which the cell that each bead is located. Each cell has its own linked list and all beads in the cell are added as nodes in the list. Beads in the same cell and adjacent cells are considered to be interacting and are marked "true" so that their interactions are computed. A single cell has eight neighboring cells to compute in a 2D environment (above, below, left, right, and one in each diagonal direction), and the algorithm computes the interaction between pairs of beads in the same cell. In a 3D implementation, there are 26 neighboring cells to check. In its original form, the cell list algorithm evaluates each bead at every timestep.

The biggest advantage of the cell list algorithm is not having to compute the Euclidean distance between each bead in the biomolecule. Instead, the cell list algorithm computes its neighbors from adjacent cells through conditional "if" statements, which is less computationally expensive than computing the squares of the differences in the x, y, and z directions in a Euclidian distance calculation. A cell list algorithm can also account for periodic boundary conditions to wrap the interactions around the simulation box.

As we had seen in the previous section with the recasting of the Verlet neighbor list algorithm for GPUs, the implementation of even well-established MD simulation algorithms on GPUs can be very nontrivial and require rethinking of the problem in the context of the GPU architecture to take advantage of the parallel architecture while limiting the performance bottlenecks from information transfer to and from the GPU device. The original cell list algorithm is typically implemented by organizing beads belonging to a cell into a linked list structure. While this approach is optimal for a CPU based architecture, the memory overhead associated with the linked list data structure is unlikely to be optimal for GPUs because of the limited memory and the performance cost associated with memory transfers.

Even without implementing a linked list data structure, there are multiple approaches to implement the serial algorithm into an equivalent GPU-optimized parallel version that exploits the underlying hardware. In one possible approach on the GPU, each cell chould be associated with a CUDA thread block and a single thread should be used to compute the forces for each bead. This approach, similar to the linked-list approach, takes advantage of the parallel architecture of the GPUs by distributing the computational load among the different beads that are interacting indepen-

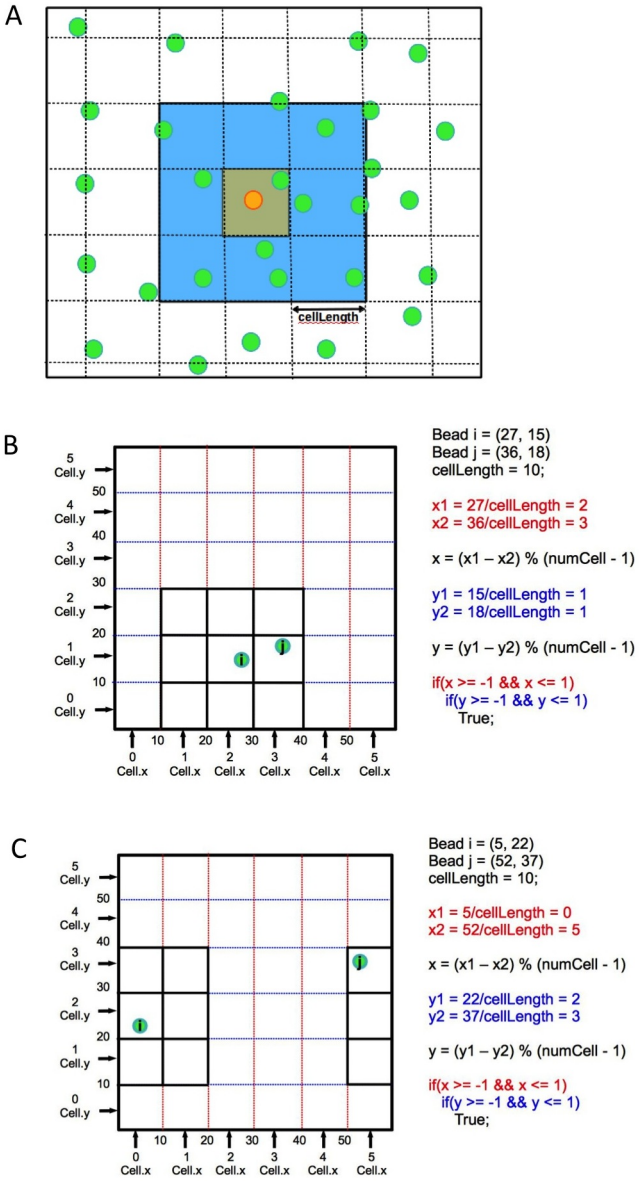

Figure 2: Illustration of cell list algorithm in 2D. (A) A 2D simulation box is shown with many beads. The simulation box is divided into cells with a length cellLength. The neighboring cells for the bead depicted in orange are shown in blue. Only the interactions between beads in the same or neighboring cells are computed. (B) An example is shown where two beads are in a simulation box that has been divided up into cells with each dimension length of 10. The two beads are deemed to have significant interactions if they are located in neighboring cells. The corresponding example and pseudocode illustrating how two beads in neighbor cells is determined is shown on the right. (C) A similar example is shown using periodic boundary conditions where the interactions are wrapped.

dently. However, this approach can leave a large number of threads unused. Unless the simulation system is saturated with beads, the vast majority of the cell will be empty. For each cell that is unoccupied, it leaves an entire thread block idle for the duration of that kernel call. Another potential approach would be to evaluate the forces by examining each bead individually. This would require each thread to locate the cell of the bead, determine the neighboring cells, and then loop over each bead in each adjacent cell. This method is also less than ideal because it requires each thread to perform several loops and it does not distribute the work to all the available processors.

Motivated to avoid these pitfalls and to optimally implement the algorithm on the GPU architecture, we instead present an alternative parallel cell list algorithm for the GPU. We evaluate whether pairs of beads are in neighboring cells by looking at each interaction individually. In our implementation of the cell list algorithm, we allocate n threads, one for each interaction. Each thread evaluates whether two beads i and j, are in neighboring cells. Instead of calculating a global cell id from the x, y, and z coordinates of the beads and searching for the beasd in neighboring cells, we compared the beads in each individual direction.

As a pedagogical illustration of the our cell list algorithm, we refer to Figure 2B, where two beads i and j are compared in the x and y directions. In this example, we chose a cell length of 10, but any value that evenly divides the simulation box is applicable. To determine the cell id in the x direction, the x coordinate of the bead is integer divided by the cell length. In Figure 2B, bead i has an x position of 27 and when divided by the cell length of 10, $x_i = 2$ so it exists in Cell.x = 2. We performed the same set of operations for bead j and determine that it lies in Cell.x = 3. Similarly, both beads i and j exist in Cell.y = 1. After taking the difference of Cell.x for both beads, an "if" statement checks if dx is within the range $-1 \geq dx \geq 1$. If the beads were in the same cell, then dx would equal 0. If dx equals 1 or -1, the beads are in neighboring cells to the left or right. Again, a similar scenario would occur for the y direction for the same reason.

To account for periodic boundary conditions, we want beads in the first cell and last cell to be recognized as neighbors. In our simulation code, this is done by computing dx modulo $numCells - 1$. In Figure 2C, Cell.x 5 and 0 are neighboring cells under the definition of periodic boundary conditions. If two beads are in these cells, dx will equal 5 or -5. Computing modulo $numCell - 1$ would assign dx a value of 0 thus allowing beads in the first and last cell to pass through the "if" statement. Beads i and j are compared in the x, y, and z directions in 3 nested "if" statements. If they are neighboring in all three directions, they are flagged as true and are added to the pair list to have their forces evaluated. This kernel is applied to each entry in the interaction list. A single thread is tasked with executing the kernel code for each pair of beads i and j and each thread does so independently and in parallel.

5. PARALLEL HYBRID NEIGHBOR/CELL LIST ALGORITHM

In the Verlet neighbor list algorithm, the computations required for calculating the interactions between beads in our simulations are significantly reduced while maintaining high accuracy by instead calculating the distance between them first to determine whether to compute the interaction using a distance cutoff. If the two beads are sufficiently far away, we know that the interaction is effectively zero so the interaction only negligibly contributes to the overall dynamics and can be ignored. To further improve performance, the Verlet neighbor list algorithm maintains two lists: 1) a neighbor list with a larger distance cutoff that is updated every N timesteps and 2) a pair list with a smaller distance cutoff with interactions that are a subset of the neighbor list. As we saw previously, the Verlet neighbor list algorithm was successfully recast for implementation on the GPU using only parallel operations.

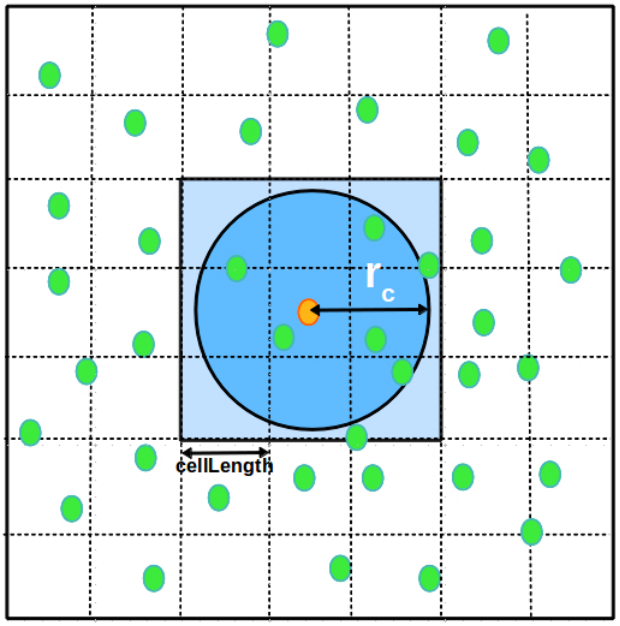

Figure 3: Illustration of hybrid neighbor/cell list algorithm. Combining the concepts of the neighbor and cell list algorithms, a cell list of interacting beads that are located in adjacent cells is updated every n_{up} timesteps, and a distance cutoff is r_c is used to compute a subset pair list of interactions that are computed.

In the cell list algorithm the relatively expensive distance calculations in the Verlet neighbor list algorithm, while still cheaper than calculating the actual interaction, is replaced by the less expensive conditional operations. By dividing up the simulation box into cells, the actual distances do not have to be calculated. As we described in the previous section, this algorithm can also be recast for GPUs using only parallel operations. In addition, by chosing appropriate cell lengths, the accuracy of our computations can be maintained while benefiting from the expected performance improvements.

It has been shown in previous studies [11, 19] that combining the Verlet neighbor and cell lists approaches into a hybrid method can improve performance on a CPU. We then asked whether it would be possible to take these two approaches and combine their benefits into a Hybrid Neighbor/Cell List algorithm for GPUs too. In our approach, we replace the neighbor list in the Verlet neighbor list algorithm with a

System	Neighbor	Cell	Hybrid
1ehz (76 beads)	27.22	25.5	27.35
16s (1,530 beads)	42.31	135.46	36.69
30s (3,883 beads)	80.64	735.92	76.40
50s (6,336 beads)	145.54	1787.23	132.18
70s (10,219 beads)	277.35	4766.35	253.74

Table 1: Execution times(s) of the GPU-optimized neighbor list, cell list, and hybrid neighbor/cell list algorithms simulated for 100,000 timesteps on the NVIDIA 480GTX.

cell list of interactions. That is, a cell list of interactions is generated that is updated every N timesteps, and a subset pair list of interactions within a distance cutoff is used to determine whether to compute the interaction. Just like the parallel Verlet neighbor list and cell list algorithms, the hybrid algorithm can be performed entirely on the GPU because it only involves parallel operations.

6. PERFORMANCE AND SCALING COMPARISONS OF TRUNCATION METHODS

Initially, our implementation of the hybrid neighbor/cell list algorithm had a slower execution time than the parallel Verlet neighbor list algorithm. However, we added an additional CUDA kernel call to store information as we will describe below, and we significantly reduced the execution time. When the hybrid list is comparing the location of two beads i and j, it must compute which cell they reside in for each dimension. Long range interactions exist between every bead in the system, therefore, each bead is involved in $N-1$ interactions where N is the number of beads. There are $N-1$ interactions because beads do not interact with themselves. This means each bead's cell location is computed $N-1$ times. As the system size increases, the cell location is needlessly computed more and more. To avoid this inefficiency, an additional CUDA kernel with N threads is used to compute and store the cell number for each bead. An array of size N stores the Cell.x, Cell.y, and Cell.z in the x, y, and z components of a float3 data type. When the interactions are being computed, the cell positons are accessed from memory rather than being calculated again.

With that modification, we made direct comparisons between the GPU-optimized parallel Verlet neighbor list, cell list, and hybrid neighbor/cell list algorithms implemented in our coarse-grained MD simulation code. All MD simulations were performed on a desktop computer with a Intel Core i7-960 3.20 GHz processor, 12 GB DDR3 RAM, and a NVIDIA 480GTX GPU. To benchmark the scaling of our code to different sized systems, we performed 100,000 timesteps of MD simulations to $tRNA^{Phe}$ (PDB code: 1ehz), and the 16s, 30s, 50s, and 70s subunits of the *E. coli* ribosome, which span about 3 orders of magnitude in the number of residues and nucleotides that are represented as beads in our simulations. To ensure a fair comparison, we counted the number of beads in the neighbor and cell list algorithms, and we chose the cell length parameter in the cell list algorithm to best match the number of beads in the neighbor list algorithm. To minimize the information transfer between the CPU and GPU, we also outputted the coordinates of the trajectory only at the end of the MD simulation. Of course, the MD simulation

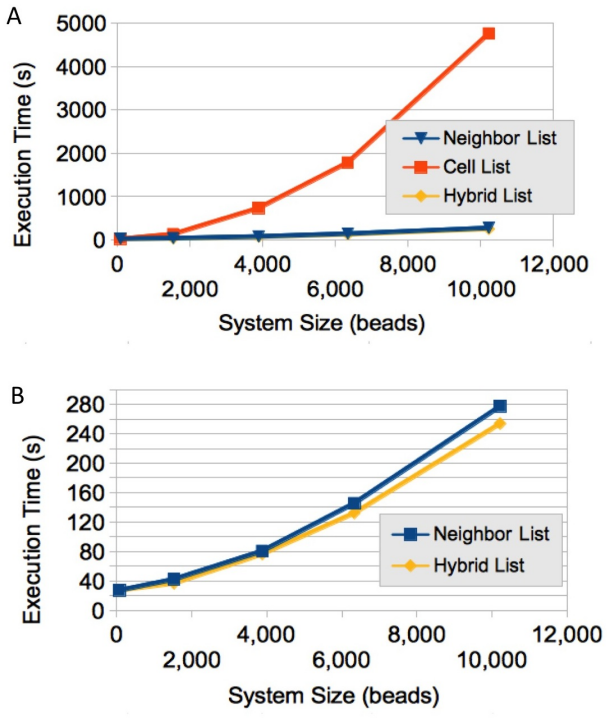

Figure 4: Execution times of GPU-optimized cutoff algorithms for systems of different size. (A) A comparison of the execution times of the neighbor, cell, and hybrid neighbor/cell list algorithms. (B) For clarity, we also present a comparison of the execution times for the neighbor and hybrid neighbor/cell list algorithms.

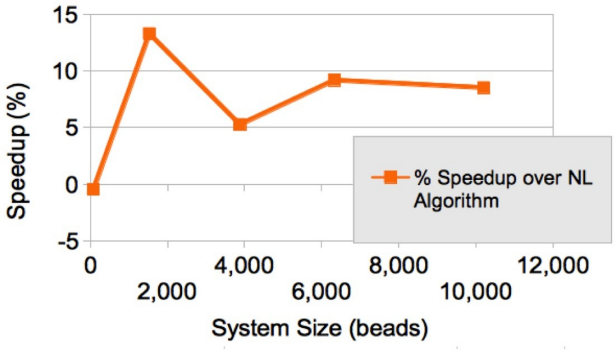

Figure 5: Percent speedup of GPU-optimized hybrid neighbor/cell list algorithm as compared to neighbor list algorithm.

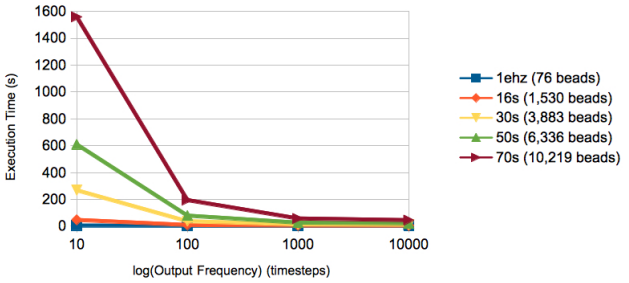

Figure 6: The output frequency dependence of the execution times scaling with system size.

coordinates are typically outputted with a regular frequency that is determined by the situation, and it requires the coordinates to be transferred from the GPU to the CPU. Note that we evaluate the effects of output frequency below.

Figure 4A shows the N-dependent execution times and the corresponding execution times are shown in Table 1. The parallel cell list algorithm is much slower than both the parallel Verlet neighbor list and hybrid list algorithms. The slower execution time is to be expected, however, because the cell list algorithm must determine whether two beads are in neighboring cells for every pair of interactions list at every timestep. The neighbor and hybrid lists only have to do this every N timesteps, and the pair list for each of those methods only has to evalute a small subset of the all possible interaction pairs. When we compare the parallel Verlet neighbor list and hybrid list algorithms by themselves, we observe similar execution times with differences that becomes more pronounced with larger systems Figure 4B.

The percent speedup of our parallel hybrid neighbor/cell list algorithm over the parallel Verlet neighbor list is, on average ∼10% faster, with the exception of our smallest system, which is represented by only 76 beads (Figure 5). The tRNA^{Phe} is so small that there are not enough beads to fully utilize the GPUs, and we actually observe a slower performance as compared to the parallel Verlet neighbor list algorithm. It is worth noting that the ∼10% speedup over the neighbor list algorithm is in addition to the ∼30x speedup of the neighbor list over the CPU version of the code for the 70s ribosome (10,219 beads).

To better gauge the performance of the hybrid neighbor/cell list algorithm under realistic conditions, we changed the output frequency of the coordinates in our MD simulation code. Since information transfer between the CPU and GPU is a major bottleneck in GPU algorithms, of course, we expect the output frequency to play an important role in the performance. We chose output frequncies ranging from every 10 timesteps to every 10,000 timesteps, and we observe that the difference becomes smaller when the coordinates are output after a larger number of timesteps (Figure 6).

7. CONCLUSIONS

We have presented a general parallel hybrid neighbor/cell list algorithm, which we implemented into a simple coarse-grained MD simulation code on the GPU. This algorithm, inspired by previous similar algorithms that were optimized for CPUs, is specifically optimized for the GPU, and it builds on a parallel Verlet neighbor list algorithm we previously developed and a parallel cell list algorithm we developed for this study. The implementation of a parallel cell list algorithm, which updates at every timestep, turned out to be significantly slower than the parallel Verlet Neighbor list algorithm. However, incorporating a parallel cell list algorithm, which uses relatively inexpensive conditional statements to search for proximal beads, into the parallel Verlet neighbor list algorithm resulted in ∼10% speedup.

8. ACKNOWLEDGMENTS

The National Science Foundation (CBET-1232724) financially supported this work. AJP and SSC acknowledge financial support from the Wake Forest University Center for Molecular and Cellular Communication and Signaling (CMCS). CAS was supported by a CMCS graduate research fellowship.

9. REFERENCES

[1] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, USA, 1989.

[2] 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, 2008.

[3] C. Clementi, H. Nymeyer, and J. N. Onuchic. Topological and energetic factors: What determines the structural details of the transition state ensemble and "en-route" intermediates for protein folding? an investigation for small globular proteins. *Journal of Molecular Biology*, 298(5):937–53, 2000.

[4] A. W. Gotz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, and R. C. Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. Generalized Born. *Journal of Chemical Theory and Computation*, 8(5):1542–1555, 2012.

[5] M. Guthold and S. S. Cho. Fibrinogen unfolding mechanisms are not too much of a stretch. *Structure*, 19(11):1536–1538, Nov. 2011.

[6] R. D. Hills and C. L. Brooks. Insights from Coarse-Grained Go models for protein folding and dynamics. *International Journal of Molecular Sciences*, 10:889–905, Mar. 2009.

[7] C. Hyeon, R. I. Dima, and D. Thirumalai. Pathways and kinetic barriers in mechanical unfolding and refolding of RNA and proteins. *Structure*, 14(11):1633–1645, 2006.

[8] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1 edition, Feb. 2010.

[9] T. J. Lipscomb, A. Zou, and S. S. Cho. Parallel verlet neighbor list algorithm for GPU-optimized MD simulations. *ACM Conference on Bioinformatics, Computational Biology and Biomedicine*, pages 321–328, 2012.

[10] B. Ma and R. Nussinov. Simulations as analytical tools to understand protein aggregation and predict amyloid conformation. *Current Opinion in Chemical Biology*, 10(5):445–452, Oct. 2006. Cited by 0133.

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

[12] D. L. Pincus, S. S. Cho, C. Hyeon, and D. Thirumalai. Minimal models for proteins and RNA from folding to function. *Progress in Molecular Biology and Translational Science*, 84:203–50, 2008.

[13] S. Plimpton and B. Hendrickson. A new parallel method for molecular dynamics simulation of macromolecular systems. *Journal of Computational Chemistry*, 17(3):326–337, 1996.

[14] A. J. Proctor, T. J. Lipscomb, A. Zou, J. A. Anderson, and S. S. Cho. GPU-optimized coarse-grained MD simulations of protein and rna folding and assembly. *ASE Science Journal*, 1:1–11, 2012.

[15] J. E. Shea and C. L. Brooks. From folding theories to folding proteins: a review and assessment of simulation studies of protein folding and unfolding. *Annual Review of Physical Chemistry*, 52:499–535, 2001.

[16] J. E. Stone, D. J. Hardy, I. S. Ufimtsev, and K. Schulten. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modeling*, 29(2):116–25, 2010.

[17] S. Takada. Coarse-grained molecular simulations of large biomolecules. *Current Opinion in Structural Biology*, 22(2):130–137, Apr. 2012.

[18] L. Verlet. Computer "Experiments" on classical fluids. i. thermodynamical properties of Lennard-Jones molecules. *Physical Review*, 159(1):98, 1967.

[19] D. B. Wang, F. B. Hsiao, C. H. Chuang, and L. Y. C. Algorithm optimization in molecular dynamics simulation. *Computer Physics Communications*, 177:551–559, 2007.

[20] A. Zhmurov, A. Brown, R. Litvinov, R. Dima, J. W. Weisel, and V. Barsegov. Mechanism of fibrin(ogen) forced unfolding. *Structure*, 19(11):1615–1624, Nov. 2011.