Parallel Verlet Neighbor List Algorithm for GPU-Optimized MD Simulations

Tyson J. Lipscomb
Wake Forest University
Department of Computer Science
1834 Wake Forest Road
Winston-Salem, NC 27109
tysonlipscomb@gmail.com

Anqi Zou
Wake Forest University
Department Computer Science
1834 Wake Forest Road
Winston-Salem, NC 27109
anqizou@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

Understanding protein and RNA biomolecular folding and assembly processes have important applications because misfolding is associated with diseases like Alzheimer's and Parkinson's. However, simulating biologically relevant biomolecules on timescales that correspond to biological functions is an extraordinary challenge due to bottlenecks that are mainly involved in force calculations. We briefly review the molecular dynamics (MD) algorithm and highlight the main bottlenecks, which involve the calculation of the forces that interact between its substituent particles. We then present new GPU-specific performance optimization techniques for MD simulations, including 1) a parallel Verlet Neighbor List algorithm that is readily implemented using the CUDPP library and 2) a bitwise shift type compression algorithm that decreases data transfer with GPUs. We also evaluate the single vs. double precision implementation of our MD simulation code using well-established biophysical metrics, and we observe negligible differences. The GPU performance optimizations are applied to coarse-grained MD simulations of the ribosome, a protein-RNA molecular machine for protein synthesis composed of 10,219 residues and nucleotides. We observe a size-dependent speedup of 30x of the GPU-optimized MD simulation code on a single GPU over the single core CPU-optimized approach for the full 70s ribosome when all optimizations are taken into account.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design, Performance

ACM-BCB'12 October 7-10, 2012, Orlando, FL, USA

Keywords

coarse-grained MD simulations, CUDPP, CURAND, floating point analysis, energy drift

1. INTRODUCTION

Biomolecules interact and assemble to form molecular machines that carry out biological functions in the cell. Molecular dynamic (MD) simulations of these processes provide molecular-resolution clues to the solutions to the biomolecular folding problems that have important applications because misfolding results in aberrant aggregation events that are widely associated with devastating conditions such as Alzheimer's and Parkinson's diseases and prion disorders [4]. As such, MD simulations are now indispensible tools in biophysics because long time-scale trajectories of biologically relevant biomolecular system can directly describe how these biomolecules perform their function. Considerable effort has been made in attaining a global understanding of the self-organization principles that determine protein and RNA folding mechanisms, resulting in breakthroughs in experiments [7], theory [21], and computations [20]. However, one of the major reasons why the challenge still remains is because of the computational demands of performing simulations of biologically relevant proteins and RNA molecules on timescales that can be directly compared with experiments.

In MD simulations, the potential energy function, which determines how the biomolecule is represented, is evaluated as a function of the coordinates of its individual substituent particles (e.g., atoms, residues/nucleotides, etc.) [1]. These particles are connected through short-range bonding interactions, and they are analogous to vertices (particles) and edges (bonds) from graph theory. The derivatives of the potential energies is numerically calculated to obtain the forces, which are in turn used to solve Newton's equations of motion, thereby moving the biomolecules in consecutive time steps to result in a "movie" trajectory of how the biomolecules moves in time.

However, simulating the folding of biologically relevant protein and RNA on timescales that can be directly compared with experiments remains an extraordinary challenge. Protein and RNA biomolecular experiments show that their folding time-scales are on the order of 10-1,000 μsecs [7], which is rapid given the inherent complexity of biomolecular folding processes [21]. The major bottleneck for simulating these processes is the force evaluation involved in

these complex systems, specifically the long-range interactions that must be computed for each pair of interacting particles, and they constitute over 90% of the MD simulation computations. However, there exist several methods for reducing the computational requirement without significantly sacrificing accuracy, such as the Verlet Neighbor List algorithm [19, 1].

To increase the timescales, some researchers use coarse-grained simulations that still accurately capture folding and binding mechanisms of biologically relevant protein and RNA biomolecules [6, 10, 5] . For these classes of simulations, groups of atoms are represented as a bead or a group of beads such that the degrees of freedom that are considered to be negligible in the overall folding mechanism are excluded. The description of the biomolecule is effectively simplified while still retaining accuracy so that these simulations become more feasible to compute.

Another significant direction that has already achieved successes is the use of graphics processing unit (GPU) hardware, which readily lends itself to a high degree of parallelization and very fast floating point calculations [18]. The CUDA programming language allows general-purpose applications, enabling a wide range of scientific computing groups to implement their code using GPUs and significantly improving performance over the traditional CPU-optimized approach. For MD simulations, the forces between each pair of independent interactions are computed in parallel.

In our present study, we introduce a GPU-optimized algorithm for calculating neighbor lists using only parallel operations, thereby eliminating the need to transfer data between the CPU and GPU to update the neighbor list. This algorithm is especially well-suited for implementation on GPUs because each of these operations involved in the algorithm are already optimized and available in the open source CUDA Data Parallel Primitives (CUDPP) Library [9]. We also discuss non-trivial, but straightforward, GPU-specific performance optimization techniques for MD simulations.

2. BACKGROUND

2.1 Coarse-Grained MD Simulations

The basic idea of an MD simulation is to track the positions of N particles that interact with one another over a span of time. The particles or groups of particles are usually represented as spherical beads, and there could be many types of interactions, depending on the complexity of the system representation. In the simplest representation of biomolecules, each particle i is connected by springs to form "bonds" via harmonic interactions to particle $i-1$ and $i+1$ (except for the first and last particle), resulting in a linear chain that is characteristic of protein and RNA biomolecules. Each particle i also interacts with all other particles j through non-bonded interactions, and those interactions are typically represented as the classical Lennard-Jones interaction that consists of attractive interactions and repulsive interactions that determine the spherical interaction of the particles. These sets of interactions that define the biomolecule are used to solve Newton's equations of motions to predict the positions and velocities of each particle over a collection of snapshots over a period of time, resulting in a "movie" trajectory.

The time spent computing the interactions, particularly the Lennard-Jones interactions between all possible pairs, scales with $O(N^2)$. As such, it is necessary to coarse-grain the representation of biomolecules to perform MD simulations of biologically relevant-sized systems for long timescales. One such approach is the Self-Organized Polymer (SOP) model MD simulation approach, where each residue or nucleotide is represented by a single bead that is centered on the C_α or C2' position of proteins or RNA, respectively, and effectively reducing N, the number of simulated particles [11, 12]. Recent studies have demonstrated that MD simulations of biomolecules using the SOP model reproduces experimentally measured mechanical properties of biomolecules with remarkable accuracy [8, 22]

In the SOP model, the energy that describes the biomolecule, and hence dictates how it moves in time, is as follows:

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

The first term is the finite extensible nonlinear elastic (FENE) potential that connects each bead to its successive bead in a linear chain. The parameters are: $k = 20\text{kcal}/(\text{mol}\cdot$ $^2)$, $R_0 = 0.2\text{nm}$, $r_{i,i+1}^0$ 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 (SSA) potential, is applied to all pairs of beads i and $i+2$ to ensure that the chains do not cross.

The third term is the Lennard-Jones potential, 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 $r_{i,j}^0$ term is the actual distance between native pairs at a given time t.

The fourth term is a repulsive term between all pairs of beads that are non-native, and σ is chosen to be 3.8Å depending on whether the pair involves protein-protein, RNA-RNA, or protein-RNA interactions.

Since the equations of motion cannot be integrated analytically, many algorithms have been developed to numerically integrate the equations of motion by discretizing time Δt and applying a finite difference integration scheme. In our study, we use the well-known Langevin equation for a generalized MD simulation such that $\vec{F} = m\vec{a} = -\zeta\vec{v} + F_c + \Gamma$ where $F_c = -\frac{\partial V}{\partial r}$ is the conformational force that is the negative gradient of the potential energy V with respect to r. ζ is the friction coefficient and Γ is the random force. The role of the random force is to mimic the effect of solvent interacting with the biomolecule, and its computation is dependent on a suitable random number generator (see below).

When the Langevin equation is numerically integrated us-

ing the velocity form of the Verlet algorithm, the position of a bead $r(t) + v(t)\Delta t + \vec{F}(t)\frac{\Delta t^2}{2m}$ where m is the mass of a bead.

Similarly, the velocity after Δt is given by

$$V(t+\Delta t) = \left(1 - \frac{\Delta t\zeta}{2m}\right)\left(1 - \frac{\Delta t\zeta}{2m} + \left(\frac{\Delta t\zeta}{2m}\right)^2\right)v(t) + \frac{\Delta t}{2m}\left(1 - \frac{\Delta t\zeta}{2m} + \left(\frac{\Delta t\zeta}{2m}\right)^2\right)$$

The MD simulation program first starts with an initial set of coordinates (r) and a random set of velocities (v). The above algorithm is repeated until a certain number of timesteps is completed, the ending the simulation.

2.2 Verlet Neighbor List Algorithm

The evaluation of the Lennard-Jones forces can be further simplified by noting that according to Newton's Third Law, the interaction between particle i and j are identical to the interaction between j and i, reducing the problem to a still challenging N(N+1)/2 steps. More significantly, when the particles are far away from each other, the interaction is negligible and effectively zero. Therefore, a cutoff radius can be introduced into the computation of the interactions that determines whether it is computed or not.

In the Verlet Neighbor List algorithm (Fig. 1), instead of calculating the interactions between every possible pair of interactions, a subset neighbor list is constructed with particles within a "skin" layer radius, r_l. This list is updated every n timesteps, and only whose interactions between beads less the cutoff radius, r_c, are computed. These computed interactions are members of a further subset pair list. 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, where σ is the radius of the interacting particle. With the Verlet Neighbor List algorithm, the computations of the interactions become $O(Nr_c^3) \sim O(N)$, which becomes far more computationally tractable [19].

3. GPU PERFORMANCE OPTIMIZATIONS OF COARSE-GRAINED MD SIMULATIONS

Recently, several studies developed and demonstrated that GPU-optimized MD simulations, including the empirical force field MD simulation software NAMD [17] and the general purpose particle dynamics simulation software suites HOOMD [2] and LAMMPS [13], can significantly increase the performance. Briefly, modern GPU architectures consist of high-throughput many-core processors that are capable of fast, parallel floating point operations. These characteristics are ideal for MD simulations because the major bottleneck of their algorithms is computing the forces between independently interacting particles. The GPU cards are placed on the motherboard of a computer so that it can directly communicate with the CPU and thereby send and receive data via a PCI Express slot, a relatively slow process.

For NVIDIA GPU cards, one can develop software for GPUs using the CUDA programming language, which is essentially the C/C++ programming language with library calls to transfer information to the GPU and perform floating point calculations in parallel. In the CUDA programming model, an application consists of a sequential "host" program that executes commands on the CPU as it is traditionally done with programs. When a computationally expensive portion of the program that would benefit from the GPU is reached, the host transfers the relevant information to the GPU and executes "kernels" that perform the same identical instruction in parallel [15]. The major performance bottleneck in this paradigm is the transfer of information to the GPU, which may not be worthwhile if the speedup on the GPU is not significant. Also, most commercially available GPUs have a finite RAM memory (1.5-6 GB) (Table 1), limiting the size of a system one can simulate on a single GPU. For these reasons, trivial GPU optimization strategies include 1) not precompute values (as it typically done in CPU programs), 2) use the smallest data types possible, and 3) move all relevant data to the GPU at the beginning of the program and keep it there for as long as possible.

To perform coarse-grained MD simulations on GPUs, we developed a basic coarse-grained MD simulation code in the CUDA programming language. The simple SOP model is an ideal starting point for the development of more sophisticated simulation software. The main focus in the development of our GPU-optimized code was to 1) minimize computations without significantly affecting the accuracy of the simulations and 2) reduce the memory transfer between the CPU and GPU that would degrade the overall performance of the simulation. In the following sections, we discuss the non-trivial, yet still straightforward strategies for GPU-optimized coarse-grained MD simulations.

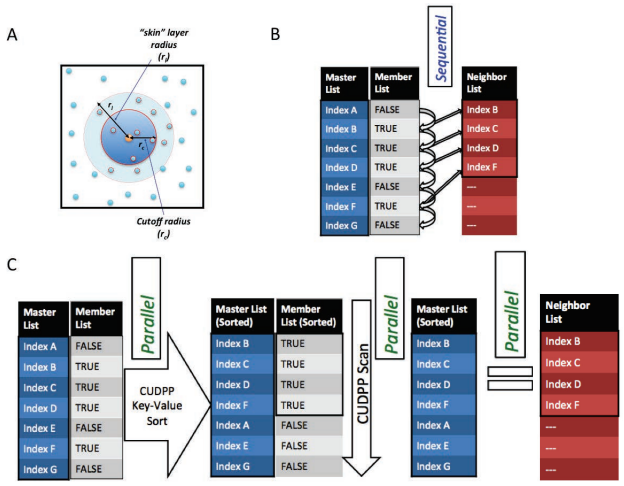

Figure 1: An overview of the differences in the traditional sequential Verlet Neighbor List algorithm and our GPU-optimized Parallel Neighbor List algorithm. (A) Schematic of the Verlet Neighbor List algorithm. Only the forces for pairs of interacting particles within the pair list are computed because all other interactions are considered to have negligible interactions. (B) In the CPU-optimized algorithm each entry in the Master List must be read serially and copied to an iteration-dependent location in the Neighbor List. (C) Using the GPU-optimized algorithm, the Master list can be sorted and scanned in parallel using CUDPP, producing an equivalent Neighbor List in a shorter amount of time.

Table 1: CPU and GPU Hardware Profiles

CPU/GPU Type	i7-950	480GTX	580GTX	C2070
# processors	4	480	512	448
clock speed	3.06 GHz	1.40 GHz	1.54 GHz	1.15 GHz
memory	16GB	1.5 GB	1.5 GB	6GB
memory bandwidth	25.6 GB/s	117.4 GB/s	192.4 GB/s	144 GB/s
FLOPS		1.35×10^{12}	1.58×10^{12}	1.03×10^{12}

3.1 Parallel Verlet Neighbor List Algorithm

Though many portions of the original CPU-optimized code are readily parallelizable, a seemingly notable exception is Verlet Neighbor List algorithm. Recall that the Verlet Neighbor List algorithm in its original form creates a subset neighbor list and a further subset pair list (Fig. 1A). Although computing whether each interaction is a member of a neighbor list or pair list can be performed in parallel, the creation of each subset list that is smaller than the original list is inherently a sequential operation (Fig. 1B). Further complicating the issue is that the neighbor list is updated once every ten thousand time steps, which is a relatively low frequency, but the pair list must be updated at every time step. Also, large amounts of data must be copied from the GPU to the host before the neighbor list or pair list is calculated, and the data representing the neighbor list or pair list must be copied from the host to the GPU once the serial calculation has taken place. The combined performance drawbacks of performing serial calculations on the CPU and transferring the large amounts of memory create a significant performance bottleneck that would benefit from parallelization.

In our implementation of the original Verlet Neighbor List algorithm (Fig. 1B), the algorithm first searches through an array of all of the interactions between pairs of beads in the system, which we called the "Master List" and determines which interactions have beads that are within the layer cutoff distance, r_l. An array of equal length to the Master List called the Member List is updated as the Master List is iterated through. When the two beads in an interaction represented by an entry in the Master List are found to be within r_l, the corresponding entry in the corresponding "Member List" is set to "true". Alternatively, if the beads within an interaction in the Master List are not within r_l, the corresponding entry in the Member List is set to "false". The calculations performed for each entry of the Master List are done independently, so this portion of the algorithm is easily parallelizable.

The second portion of the neighbor list algorithm is more complex and does not lend itself to parallelization. The Member List is iterated through serially and when an entry at position i is found to be "true", the i-th entry of the Master List is copied to the Neighbor List. The index in the Neighbor List array that the Master List entry will be copied to is determined by how many previous entries have been added the the Neighbor List. For example, if j entries have already been added to the Neighbor List when the i-th entry of the Member List is found to be "true", the i-th entry of the Master List will then be copied to the j+1-st entry of the Neighbor List. The pseudocode for this process is listed below:

```
j = 0;
for(i = 0; i < num_NL; i++)
    if(Member_List[i] = TRUE)
        Neighbor_List[j] = Master_List[i];
        j++;
```

If the action taken by each iteration of the for-loop was independent of the others, each action could be assigned to its own independent, parallel process. However, since the value of the j variable could possibly vary at each iteration, it is unknown which location of the Neighbor List the true values of the Master List could be copied to without calculating each of the previous values. It is therefore clear that this particular implementation of the Verlet Neighbor List algorithm cannot be parallelized since the location of the copy that takes place in each iteration of the for-loop is dependent on the results of each of the previous iterations.

To parallelize the Verlet Neighbor List algorithm to run on a GPU, a new approach was taken by utilizing the key-value sort and parallel scan functionalities of the CUDA Data Parallel Primitives Library (CUDPP) (Fig. 1C). This approach calculates a neighbor list that is equivalent to the one calculated by the serial Verlet Neighbor List algorithm and accomplishes this entirely on the GPU, circumventing the need to transfer data between the CPU and GPU and perform serial calculations on the CPU.

The first step of this parallel method is to perform a key-value sort on the data, using the Member List as the keys and the Master List as the values. Since the Member List needs to only hold a value of "true" or "false", these values can be represented with a zero or one. Sorting these values in numerical order will simply arrange them in such a way that the "true" values are moved to the top of the Member List and the "false" values are moved to the bottom of the list. The associated values of the Master List will be moved in an identical fashion, meaning that the entries of the Master List that are part of the Neighbor List will occupy the first entries in the list and can be copied directly to the Neighbor List in parallel. Although the entries that need to be copied to the Neighbor List are guaranteed to be at the top of the Master List after the key-value sort takes place, the exact number of entries that need to be copied to the Neighbor List will still need to be determined. Fortunately, the parallel scan algorithm implemented by CUDPP can quickly sum up the values in an array in parallel. Since the values present in the Member List consist of only zeros and ones, a parallel scan will produce the total number of "true" values present in the array, indicating how many entries of the Master List will need to be copied to the Neighbor List.

Using the parallel key-value sort and scan functionalities of CUDPP therefore produces a Neighbor List that is equivalent to the one created by the serial CPU algorithm without requiring the GPU to pause while the data is transferred to and from the CPU and serial calculations are performed on the CPU. Our algorithm eliminates a very significant performance bottleneck and allows the program to perform much

more efficiently. Note that each operation is performed in parallel, and they can be implemented completely on the GPU. Also, the sorting and scan algorithms are performed in place in the array so that no new data structures have to be constructed, minimizing the memory footprint that would degrade the performance of the algorithm on a GPU.

3.2 Type Compression

The most straightforward approach to cut down on the amount of data that is transferred between the host and device, thereby minimizing latency, is to use the smallest data types possible. When using integral data types, the minimum size of each variable is dictated by the maximum value that it could be required to hold in a simulation. For example, if eight bits were allocated to assign a unique number to each bead in a simulation, there would be a maximum of 2^8-1 = 255 beads that could be used in a simulation. Using smaller data types therefore limits the size of the structure that can be simulated, but in practice this does not create any problems so long as reasonable sizes are chosen. Data types in C/C++, CUDA and most other programming languages come in only a limited number of sizes, however, so it is entirely possible that when trying to find an optimal size for a data type given a maximum structure size, the maximum values given by the built-in data types may be either too small to store the information necessary, or much larger than necessary, leading to wasted space in the form of bits that will never be used.

As an illustration, we give an example in our development that optimally introduces type compression to minimize memory transfer latency. While optimizing the CUDA SOP MD simulation code, one variable that had its size reduced was the variable that identified which beads were being represented in an interaction that was stored in the neighbor list and pair list. In a 70s ribosome, a total of 10,219 bead are present, so the variable used to represent one of these beads must be able to represent at least 10,219 different values. Using a 13-bit variable would give a total of 2^{13}-1 = 8,191 total representable beads, which would not allow for enough unique values. Using a 14-bit variable, on the other hand, would allow 2^{14}-1 = 16,383 total representable beads, which is more than enough, but there are no 14-bit data types in CUDA or C/C++. The ushort (unsigned short integer) data type occupies 16 bits of memory, which is enough to represent 2^{16}-1 = 65,535 different values. This is more than would be necessary for the 70s ribosome, but leaves 2 bits that will always be unused. Still, using the ushort data type instead of the 32-bit int data type would reduces the amount of time taken while transferring information and the amount of space needed to store that information by half.

In addition to the variable identifying each bead in a neighbor list or pair list entry, a different variable is used to represent whether the interaction type is protein-protein, protein-RNA, RNA-protein, or RNA-RNA. Since there are a total of four different types of interactions, no more than two bits would be necessary to represent these values. The smallest data type in CUDA and C/C++ is the uchar (unsigned character) which is an 8-bit value capable of representing 2^8-1 = 255 different values. Though this is more than enough unique values, a total of six bits will always be unused, leading to a large waste of storage space and memory transfer time.

Since the identifying number for a bead in the 70s ribosome could be represented by using 14 bits and the type of interaction could be represented by using only 2 bits, these two variables could be combined into a single 16-bit ushort value to completely eliminate the wasted space when using a ushort and uchar to represent each of them individually. This reduces a total memory requirement for each entry from 24 bits to 16 bits, leading to a 33% reduction in memory requirements and latency.

The values were combined into a single data type by using the two highest-order bits in the 16-bit ushort to store the type of the interaction and the fourteen low-order bits to store the identifying number of the bead. These values are combined by setting the ushort value to be equal to the value of the interaction type left shifted by 14 bits and using a bitwise OR operation to set the low-order fourteen bits to the identifier. Once these values are stored in a ushort variable, they can be retrieved very easily. To find the type of the interaction that is stored in the variable, the value is simply right shifted by 14 bits. Alternatively, to find the identifying number stored in the variable, a bitwise AND operation is performed to set the two highest-order bits to zero, leaving the fourteen lowest-order bits unchanged.

3.3 GPU-Optimized MD Simulations Performance

We next benchmarked the performances of the CPU- and GPU-optimized codes as a function of system size by simulating systems ranging from the $tRNA^{phe}$ molecule (76 beads) to the full 70s ribosome (10,219 beads) (Fig. 2,3) for 1,000,000 timesteps. The ribosome is an ideal model system for benchmarking because even though it is relatively large, it can be readily discomposed into smaller subunits. We used the experimentally known structure of the ribosome [16] as a basis for constructing a SOP-Model representation for CPU- and GPU-optimized MD simulations.

To quantify the performance difference between the CPU- and GPU-optimized codes performed on a single CPU core/GPU card desktop, we measure the speedup factor, $S(p) = T_{CPU}/T_{GPU}$, where T is the execution time for the calculation of the CPU- or GPU-optimized code for each of the systems we studied. Of course, one can unfairly inflate the S(p) by simply running inferior CPU-optimized code. However, we stress that the CPU- and GPU-optimized codes are optimized for their respective architecture. For example, the CPU-optimized code takes advantage of precomputing values to hold in memory so that the values would not have to be recomputed. On the other hand, since the memory transfer is a major bottleneck, it is simply faster to keep recomputing values at the GPU rather than transferring them to the GPU.

To evaluate our CPU- and GPU-optimized MD simulations, we began by implementing them on different types of GPUs, namely the NVIDIA 480GTX, 580 GTX, and C2070 (Fig. 2A). All three GPUs types we tested are based on the Fermi architecture, and they mainly differ in the processor speed, memory size, and number of computational cores (i.e., processors). A particularly significant limitation of GPUs is the relatively small amount of memory available to it. On the 480GTX and 580GTX, the memory is limited to 1.5 GB and the C2070 has a capacity of 6GB, which are all considerably smaller than most modern CPU architectures (Table 1). However, through significant mem-

ory optimizations, even the full 70s ribosome was able to fit into a single GPU in our GPU-optimized MD simulation code. Therefore, its performance became a function of the processor speed and number of processors (Fig. 2A).

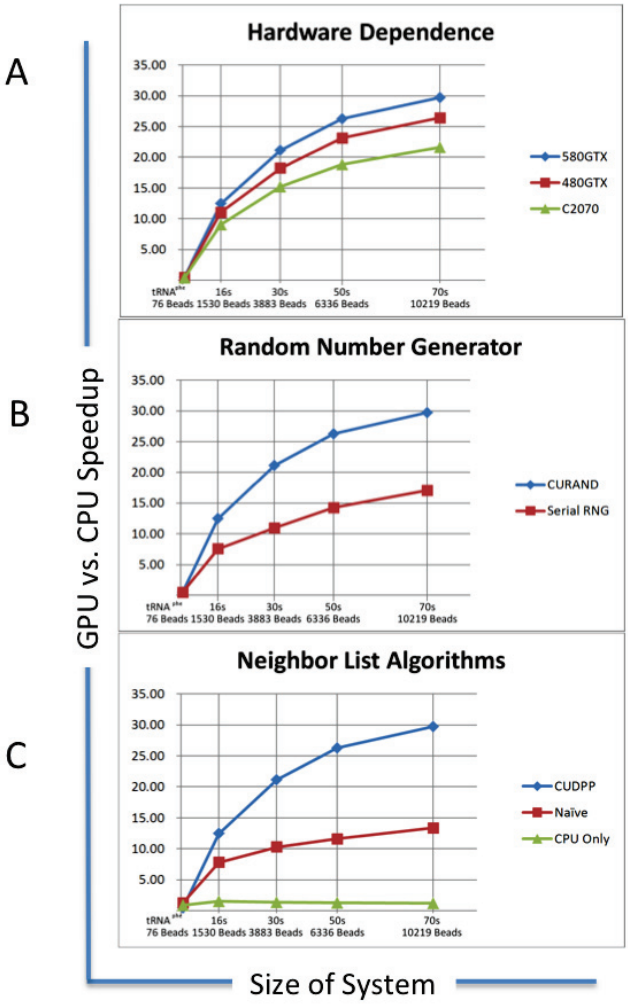

Figure 2: System size (N) dependent performance improvements of GPU-optimized SOP MD simulations benchmarked versus CPU-optimized implementation on a single CPU core/GPU card desktop. (A) GPU-optimized code with traditional neighbor list algorithm and various models of GPUs, (B) GPU-optimized code with additional CURAND random number generator, (C) GPU-optimized with GPU-optimized neighbor list algorithm.

Interestingly, for systems that are very small ($< \sim$100 beads), the GPU-optimized SOP model program is actually slower than the CPU-optimized one, presumably because the computational speedup benefit of the GPU is less than the cost of the the memory transfer, a major bottleneck for GPUs. Therefore, even if there is a speedup in the floating-point operations, the memory transfer latency becomes the rate-limiting step. However, for larger sized systems, there is a marked speedup using the GPU-optimized program over the CPU-optimized program. Indeed, the full 70s ribosome with 10,219 residues/nucleotides is $\sim$30x faster than the CPU-optimized code, and the exact speedup is clearly dependent on the number of particles in the system of study.

We next evaluated the difference between using a CPU-optimized random number generator [14] and the NVIDIA CURAND library for the computing of the random force term in our Langevin dynamics integrator. We observe a doubling of the speedup across all of the ribosome subunits we simulated for benchmarking (Fig. 2B).

Finally, we benchmarked our simulations using three different variants of the neighbor list algorithm on the GPU (Fig. 2C). When we computed the neighbor list on the CPU, we observed a modest speedup that leveled off. For simulations using the original neighbor list algorithm where the subset neighbor lists are generated on the GPU, there exists an N-dependent speedup with $\sim$14x speedup for the full 70s ribosome. However, for simulations using our GPU-optimized algorithm, the N-dependent speedup results in $\sim$30x speedup for the 70s ribosome.

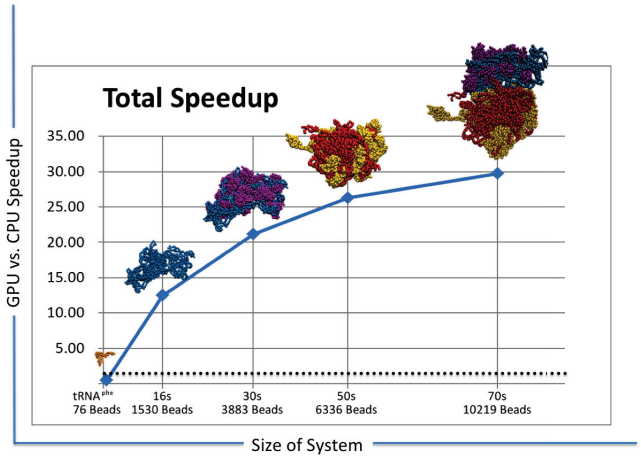

Figure 3: SOP Model MD simulation performance benchmarks with GPU-optimized neighbor list algorithm + CURAND on 580 GTX. The dotted line indicates that the GPU vs. CPU speedup is 1 (i.e., same execution time). The speedup is clearly N-dependent, which is $\sim$30x for the full 70s ribosome. Note that the tRNA^{phe} MD simulation execution time on the GPU-optimized code is slower than on the CPU-optimized code.

3.4 Floating Point Arithmetic and CUDA

When writing a program to run on a GPU, the mathematical operations that are performed will be done in an environment that is much different from that of the CPU, which requires programmers to ensure that operations produce accurate results. Though the current generation of NVIDIA GPUs are IEEE 754 compliant with both single and double precision floating point numbers, previous generations of hardware were not. The features that are supported by a GPU can be determined by its Compute Capability version, consisting of both a major and a minor revision number. GPUs that are Compute Capability 1.2 or lower do not support IEEE 754 double precision numbers and are not fully IEEE 754 compliant with some single precision floating point operations. However, Compute Capability 1.3 brought the

Table 2: Size-dependent execution times (in seconds) for 1 million timesteps of CPU/GPU floating point precision implementations of MD simulations

	$tRNA^{phe}$	16s	30s	50s	70s
Number of Beads	76	1,530	3,883	6,336	10,219
GPU Single	292.45	389.31	641.89	1,191.87	2,204.87
GPU Double	292.45	411.94	737.73	1,409.57	2,647.22
CPU Single	142.90	5,293.94	13,827.40	32,209.40	66,009.10
CPU Double	137.16	4,959.63	12,619.10	29,975.30	61,249.80

introduction of IEEE 754-compliant double precision support, but did not improve the single precision support of Compute Capability 1.2 and lower devices. Full IEEE 754 support with both single and double precision was introduced in Compute Capability 2.0 devices and has been implemented in all devices since these were introduced.

Even though the current generation of CUDA capable GPUs support IEEE 754 floating point arithmetic, a program written for a CUDA GPU will often produce results that differ from the ones generated by the CPU in an equivalent calculation. It is also likely that numerical results produced by one GPU program run will be different than those of the same program during a different run. In a single-threaded CPU program, the order in which the different operations will take place are well-defined, which result in identical output after every run. In any multi-threaded architecture such as GPUs, there exist multiple orders in which the operations can be performed. In GPUs, if different blocks perform calculations on overlapping portions of data, the order of those operations will therefore be undefined and can lead to slightly different results across multiple runs. This will often lead to differences between the results of a CPU-optimized calculation and a GPU-optimized one under certain circumstances.

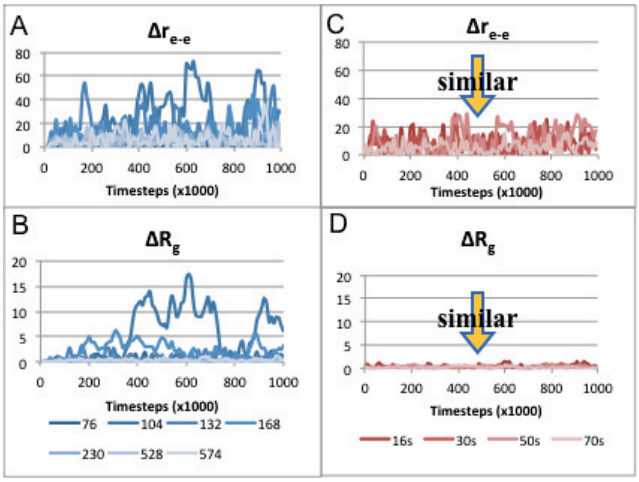

Figure 4: Difference between trajectories for (A,B) small and (C,D) large systems using the end-to-end distance, Δr_{e-e}, and radius of gyration, ΔR_g

3.5 Biophysical Metrics to Evaluate Single vs. Double Precision: Negligible Differences and No Appreciable Energy Drift

To evaluate the floating point precision calculations of our MD simulation code, we used multiple metrics that are commonly used in biomolecular folding experiments to compare single- and double-precision based MD simulation trajectories. The end-to-end distance, r_{e-e}, and radius of gyration R_g are readily measured for biomolecules in FRET and SAXS experiments, respectively. The differences of these metrics for each frame in a trajectory that corresponds to structures can evaluate the differences between the two trajectories. We observe larger differences for smaller sized systems, but the overall difference between the single- and double-precision based trajectories are minimal in our view, at least at the resolution of the biophysical metrics we used (Fig. 4).

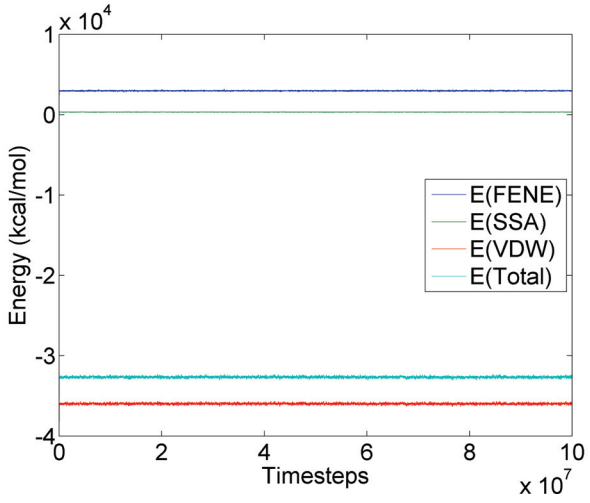

Figure 5: The energy over the course of a representative 100 million time steps of MD simulation trajectory. The total energy of the system, as well as the energies of the constituent terms of the SOP model, is demonstrated to have negligible changes over the course of the trajectory.

We also compared the energies associated with a single- and double- precision MD simulations. Some researchers using older generation GPUs observed an "energy drift" in their calculations using single precision numbers where the energy of the system slowly increased[3]. This is significant because the simulations were performed for an isolated system such that there should be no energy entering or escaping the system (i.e., the total energy should be nearly constant). This concept is known as "detailed balance," and it is a fundamental law in physics. In our simulations, we observe no appreciable energy drift over a long trajectory, even when using single-precision MD simulations (Fig. 5).

4. CONCLUSIONS

We developed a GPU-optimized coarse-grained MD simulation software, and we evaluated various performance strategies that are specific to the GPU implementation. We benchmark the performance of our code by evaluating the speedup versus a CPU-optimized implementation using various subunits of the ribosome over a range of different sizes. We observe an N-dependent speedup for the range of biologically relevant systems we studied. By maximizing the computations performed on the GPU, we noted the greatest speedup in our performance. In particular, a GPU-optimized Parallel Verlet Neighbor List algorithm we implemented by taking advantage of readily available CUDPP libraries resulted in a speedup of about 30x for the full 70s ribosome, a system with 10,219 residues/nucleotides. The algorithm is general for all MD simulations.

Furthermore, the speedup increases monotonically over the range of the systems we studied, but it should be noted that since there is finite memory in the GPUs, a physical RAM memory limit will be reached when the system is large enough, and a multi-GPU approach would be necessary to perform the simulation. However, such an implementation would not be expected to continuously increase the performance with system size because of the performance costs of communication between the GPUs.

5. ACKNOWLEDGMENTS

TJL acknowledges financial support from the Wake Forest University Computer Science Graduate Fellowship for Excellence. SSC is grateful for financial support from the Wake Forest University Science Research Fund and unrestricted GPU equipment donation from the NVIDIA Academic Partnership.

6. 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] B. A. Bauer, J. E. Davis, M. Taufer, and S. Patel. Molecular dynamics simulations of aqueous ions at the liquid-vapor interface accelerated using graphics processors. *Journal of Computational Chemistry*, 32(3):375–385, 2011.

[4] F. Chiti, N. Taddei, P. M. White, M. Bucciantini, F. Magherini, M. Stefani, and C. M. Dobson. Mutational analysis of acylphosphatase suggests the importance of topology and contact order in protein folding. *Nature Structural Biology*, 6(11):1005–9, 1999.

[5] S. S. Cho, D. L. Pincus, and D. Thirumalai. Assembly mechanisms of RNA pseudoknots are determined by the stabilities of constituent secondary structures. *Proceedings of the National Academy of Sciences*, 106(41):17349 –17354, Oct. 2009.

[6] 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.

[7] W. A. Eaton, V. Munoz, S. J. Hagen, G. S. Jas, L. J. Lapidus, E. R. Henry, and J. Hofrichter. Fast kinetics and mechanisms in protein folding. *Annual Review of Biophysics and Biomolecular Structure*, 29:327–59, 2000.

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

[9] M. Harris, S. Sengupta, and J. Owens. *Parallel Prefix Sum (Scan) with CUDA*. GPU Gems 3. Addison Welsey, 2007.

[10] R. D. Hills and C. L. Brooks. Insights from Coarse-Grained go models for protein folding and dynamics. *International Journal of Molecular Sciences*, 10(3):889–905, 2009.

[11] 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.

[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] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 3 edition, Sept. 2007.

[15] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1 edition, 2010.

[16] B. S. Schuwirth, M. A. Borovinskaya, C. W. Hau, W. Zhang, A. Vila-Sanjurjo, J. M. Holton, and J. H. D. Cate. Structures of the bacterial ribosome at 3.5 a resolution. *Science*, 310(5749):827 –834, 2005.

[17] 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.

[18] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–40, 2007.

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

[20] V. A. Voelz, G. R. Bowman, K. Beauchamp, and V. S. Pande. Molecular simulation of ab initio protein folding for a millisecond folder NTL9(1-39). *Journal of the American Chemical Society*, 132(5):1526–8, 2010.

[21] P. G. Wolynes, J. N. Onuchic, and D. Thirumalai. Navigating the folding routes. *Science*, 267(5204):1619–20, 1995.

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