GPU-Optimized Coarse-Grained MD Simulations of Protein and RNA Folding and Assembly

Andrew J. Proctor, Tyson J. Lipscomb, Anqi Zou
Department of Computer Science
Wake Forest University
Winston-Salem, NC 27109
Email: {proca06, tysonlipscomb, anqizou}@gmail.com

Joshua A. Anderson
Department of Chemical Engineering
University of Michigan
Ann Arbor, Michigan 48109
Email: joaander@umich.edu

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

ABSTRACT

Molecular dynamics (MD) simulations provide a molecular-resolution physical description of the folding and assembly processes, but the size and the timescales of simulations are limited because the underlying algorithm is computationally demanding. We recently introduced a parallel neighbor list algorithm that was specifically optimized for MD simulations on GPUs. In our present study, we analyze the performance of the algorithm in our MD simulation software, and we observe that the major of the overall execution time is spent performing the force calculations and the evaluation of the neighbor list and pair lists. The overall speedup of the GPU-optimized MD simulations as compared to the CPU-optimized version is N-dependent and ∼30x for the full 70s ribosome (10,219 beads). The pair and neighbor list evaluations have performance speedups of ∼25x and ∼55x, respectively. We then make direct comparisons with the performance of our MD simulation code with that of the SOP model implemented in the simulation code of HOOMD, a leading general particle dynamics simulation package that is specifically optimized for GPUs.

I INTRODUCTION

All cellular processes occur because biomolecules such as proteins and RNA fold and assemble into well-defined structures that allow them to perform specific functions [1] including enzyme catalysis, structural support, transport, and regulation. Understanding how biomolecules carry out their functions have direct medical applications because some functions are deleterious, such as the telomerase enzyme that promotes cancer, and misfolding events are widely accepted to lead to diseases such as Alzheimer's and Parkinson's.

A powerful computational tool for studying bimolecular folding and assembly mechanisms is molecular dynamics (MD) simulations, which allows the study of atomic and molecular systems, usually represented as spherical beads, and how they interact and behave in the physical world. The physical description of biomolecules in MD simulations is determined by an energy potential, which can include short-range bonding interactions and long-range van der Waals (often modeled as a Lennard-Jones potential) or electrostatic interactions. The gradient of the energies is used to compute the set of independent forces acting on each bead, which are in turn used to calculate the a new set of positions and velocities that determines how it moves in time. Although the MD simulation protocol is relatively straightforward, it is computationally demanding because the long-range interactions scales as $O(N^2)$ because each pair of independent interactions between beads i and j is considered. To reduce the computational demands of the long-range interactions, neighbor list and cell list algorithms consider only interactions that are close in distance, and these approaches scales as $O(Nr_c^3)$, where r_c is a cutoff distance.

An approach that is particularly well-suited for performing MD simulations with a demonstrated track record of success is the use of graphics processor units (GPUs) (Fig. 1). Recently, several studies developed and demonstrated that GPU-optimized MD simulations, including the empirical force field MD simulation software NAMD [2] and AMBER [3] and the general purpose particle dynamics simulation soft-

ISBN: 978-1-62561-001-0

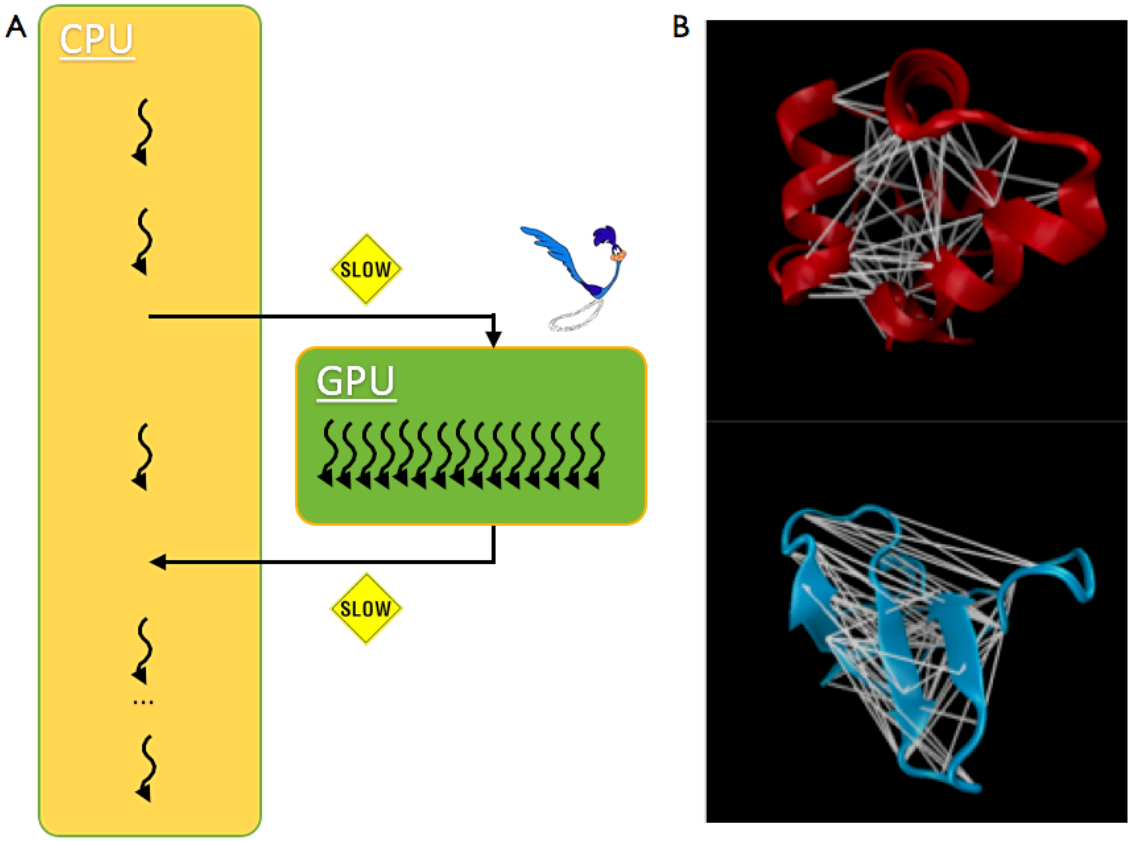

Figure 1: MD simulations of biomolecular systems on GPUs. (A) A schematic of the processing flow of CPUs and GPUs. In a single-core processor, a single thread sequentially executes commands while GPUs have many threads that executes commands in parallel. (B) A cartoon of two typical proteins with the native interactions between residues shown. Each interaction can be computed independently in MD simulations, which makes GPUs ideal architectures for implementation of MD simulation algorithms.

ware suites HOOMD [4] and LAMMPS [5], 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 a MD simulation.

In our present study, we implement a GPU-optimized parallel Verlet neighbor list algorithm, and we analyze its performance. We first compare the GPU-optimized MD simulation performance to a CPU-optimized implementation of the same MD simulation approach. We then evaluate the performances of the individual components of the MD simulation algorithm to isolate the major bottlenecks that still remain. We then make direct comparisons to the HOOMD MD simulation software by comparing the performances the full SOP model implemented in our in-house simulation code and in HOOMD.

1 MD SIMULATION ALGORITHM

MD simulations are now indispensible tools for studying biomolecular folding and assembly processes at a molecular resolution. MD simulate systems of atomic or molecular structures and the way they interact and behave in the physical world. The physical description of the biomolecule in MD simulations is determined by an energy potential. The most basic description of a biomolecule must be an energy potential 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, etc. One example is the Self-Organized Polymer (SOP) model energy potential [6, 7], which we will describe in detail in the next section. Regardless, the physical description of biomolecules essentially become spher-

ISBN: 978-1-62561-001-0

ical nodes that are connected by edges that correspond to interactions. The total structure (i.e., the collection of nodes and edges) within a molecular dynamics system can be thought of as an abstraction of a much smaller collection of physical entities (i.e., nodes) that acts as a single, indivisible unit within the system.

Once the energy potential is determined, one must determine the rules for moving the biomolecule over time. The molecular dynamics 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 trajectory [8].

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 $\vec{F} = m\vec{a} = -\zeta\vec{v} + \vec{F_c} + \vec{\Gamma}$ where $\vec{F_c} = -\frac{\partial \vec{v}}{\partial \vec{r}}$ is the conformational force that is the negative gradient of the potential energy $\vec{v}$ with respect to $\vec{r}$. ζ is the friction coefficient and $\vec{\Gamma}$ is the random force.

When the Langevin equation is numerically integrated using the velocity form of the Verlet algorithm, the next position of a bead after an integration step Δt is given by:

$$r(t + \Delta t) = \vec{r}(t) + \Delta t\vec{v}(t) + \frac{\Delta t^2}{2m}\vec{F}(t)$$

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_0, and a random set of velocities v_0. The above algorithm is repeated until a certain number of timesteps is completed, thus ending the simulation.

2 COARSE-GRAINED MD SIMULATIONS

When performing a molecular dynamics simulation there are a wide range of levels of precision that can be used depending on the desired degree of accuracy (Fig. 2). If a detailed model is desired, it is possible to model a biomolecule at the quantum level, simulating the behavior of each individual electron within the molecular structure. Unfortunately a quantum-mechanical description demands a great deal of computational resources to simulate and greatly limit the size of systems and the length of time that can be simulated. By current standards, a quantum-mechanical simulation is restricted to only about 100 atoms and the timescale is on the order of pico- or femtoseconds.

An obvious solution to the computational problems introduced by quantum-mechanical molecular dynamics simulations would be to simulate a system at the atomistic level, representing each atom with a bead. This greatly increases the size of molecules that can be simulated compared to quantum level simulations. Atomistic resolution folding simulations are, however, restricted to very small (about 50 amino acids long), fast-folding (less than a microsecond) proteins at atomistic detail (see [9], for example), even though biologically relevant biomolecules are much larger and can fold much more slowly. By comparison, a single protein chain is, on average, 400 amino acids long and takes considerably longer than a millisecond to fold. 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 [10, 11]. For these classes of simulations, groups of atoms are typically represented as a bead or group of beads such that the degrees of freedom that are considered to be negligible in the overall folding mechanism are excluded, thereby reducing the total number of simulated particles so that the simulations become more feasible to compute [7, 12].

In the Self-Organized Polymer (SOP) Model, a coarse-grained approach that has been demonstrated to reproduce experiments remarkably well [6,7,13,14], the energy potential that describes the biomolecule, and hence dictates how it moves in time, is as follows:

ISBN: 978-1-62561-001-0

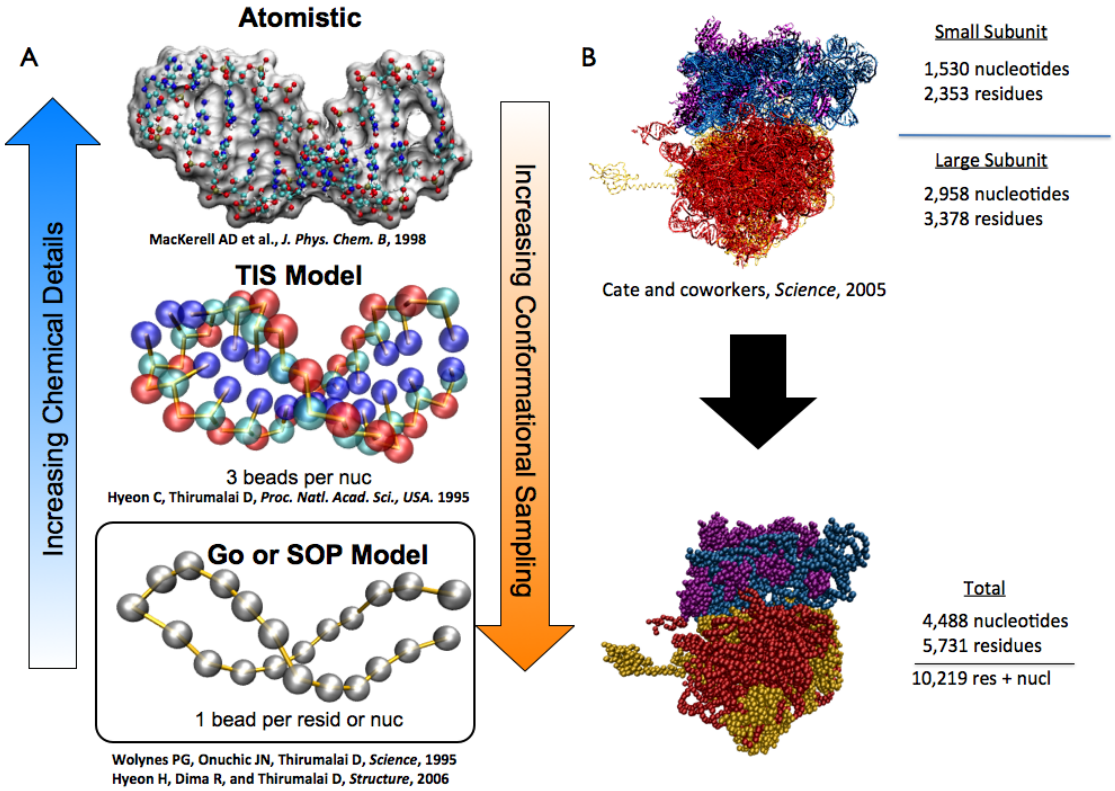

Figure 2: Resolution and sampling issues in MD simulations. (A) Different levels of MD simulation resolutions. In the coarse-grained approach, the negligible degrees of freedom are reduced by representing groups of atoms as beads. With fewer interactions, the timescales of the simulation can be increased because the computational demands are significantly reduced. (B) The coarse-graining of the ribosome structure. From the full atomistic structure, one can coarse-grained the ribosome to be represented beads for each residue and nucleotide.

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

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: $k = 20\text{kcal}/(\text{mol} \cdot \mathring{A}^2)$, $R_0 = 0.2\text{nm}$, $r_{i,i+1}^0$ 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 $i+2$ 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 protein-protein, protein-RNA, or RNA-RNA bead pair i and j, such that $|i-j| > 2$, a native pair is defined as having a distance less than 14Å 11Å or 8Å respectively, 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.

ISBN: 978-1-62561-001-0

The fourth term is a repulsive term between all pairs of beads that are non-native, and σ is chosen to be 7.0Å , 5.4Å , or 3.8Å depending on whether the pair involves protein-protein, protein-RNA, or RNA-RNA interactions, respectively.

The overall goal of the coarse-graining MD approach is to increase the timescales of the simulations so that conformation sampling is enhanced while still retaining the essential chemical details that capture the physics of the biomolecular folding and assembly processes (Fig. 2). It is important to note that the interactions in the van der Waals energies (and thus forces) scales as $O(N^2)$, which can be avoided using a truncation scheme such as a neighbor list algorithm, which we describe below.

II GPU-OPTIMIZED MD SIMULATIONS

At each timestep in a simulation, a variety of forces must be calculated for each bead. Since the forces on each bead can be calculated independently of each other at every timestep, individual threads can be assigned to each force that acts on each bead (Fig. 1). For example, if five distinct types of forces are being simulated on one hundred beads, a total of five hundred threads would be used. Each one of the threads can then perform the necessary calculations in parallel, greatly reducing the computation time of a simulation. Updating the positions and velocities are likewise highly parallel operations.

While calculating forces acting on each bead and updating their positions and velocities are by themselves parallel tasks, the entire operation is an ordered, serial process. At the beginning of each timestep the pair list is calculated, along with the neighbor list if sufficient time has passed since the last update. Positions are then updated based on the current distribution of forces. Once this has taken place, the forces acting on the beads based on their new positions must be calculated. Finally, the velocities of each bead are updated based on the forces present in the current timestep. Between each of these steps the entire process must be synchronized in order to perform accurate computations. An MD simulation can therefore be thought of as a set of highly parallel tasks that must be performed in a specific order.

When optimizing a CPU-based program or algorithm for use on a GPU, there are many different factors that must be taken into account. Though the type of calculations that CPUs and GPUs perform are fundamentally the same, the ways that they go about performing these calculations can vary due to differences in hardware implementations. The parallel hardware of GPUs and their different memory hierarchies and access patterns place constraints on the programmer that are often not necessary to address when writing a CPU-based program.

1 TYPE COMPRESSION

A trivial approach to minimize memory transfer between the CPU and GPU is to use the smallest data types possible. For integer data types, the minimum size of each variable one can choose is determined by the maximum value that it is expected to hold. For example, if 8 bits were allocated to assign indices for each bead in an MD simulation, the maximum number of beads in the simulation would be $2^8 = 256$. While the minimizing data types limits the size of the system one can simulate, the benefit is the lower latency that results in faster performance.

One limit to this approach is that the available built-in sizes of data types in C/C++, CUDA, and most other programming languages are limited and fixed. Therefore, it is possible, and indeed likely, that the optimal sized data type for a simulation may be either too small to store the necessary information or larger than necessary, leading to wasted space in the form of bits that will never be used and slow down performance.

In our implementation of the SOP model MD simulation code, we used arrays of indices for the beads involved in a native or non-native Lennard-Jones interactions. The largest biomolecule we studied was the 70s ribosome, which is represented by 10,219 beads. The minimum number of bits required to represent the indices of the beads, n, is $\lceil \log_2(n) \rceil$, which for the 70s ribosome is $\lceil \log_2(10,219) \rceil = 14$. Using 14 bits allows a maximum of $2^{14} = 16,384$ beads to be represented in this way.

Unfortunately, there are no 14-bit data types in CUDA or C/C++. The next largest available data type, ushort (unsigned short integer), occupies 16 bits of memory, which is enough to represent $2^{16} = 65,536$ different values. Not only is this significantly more than necessary, two bits will remain unused (Fig. 3A).

The weight and minimum energy distance of interaction between the beads is determined by whether the bead represents an amino acid or nucleic acid of a protein or RNA, respectively. There are a total

ISBN: 978-1-62561-001-0

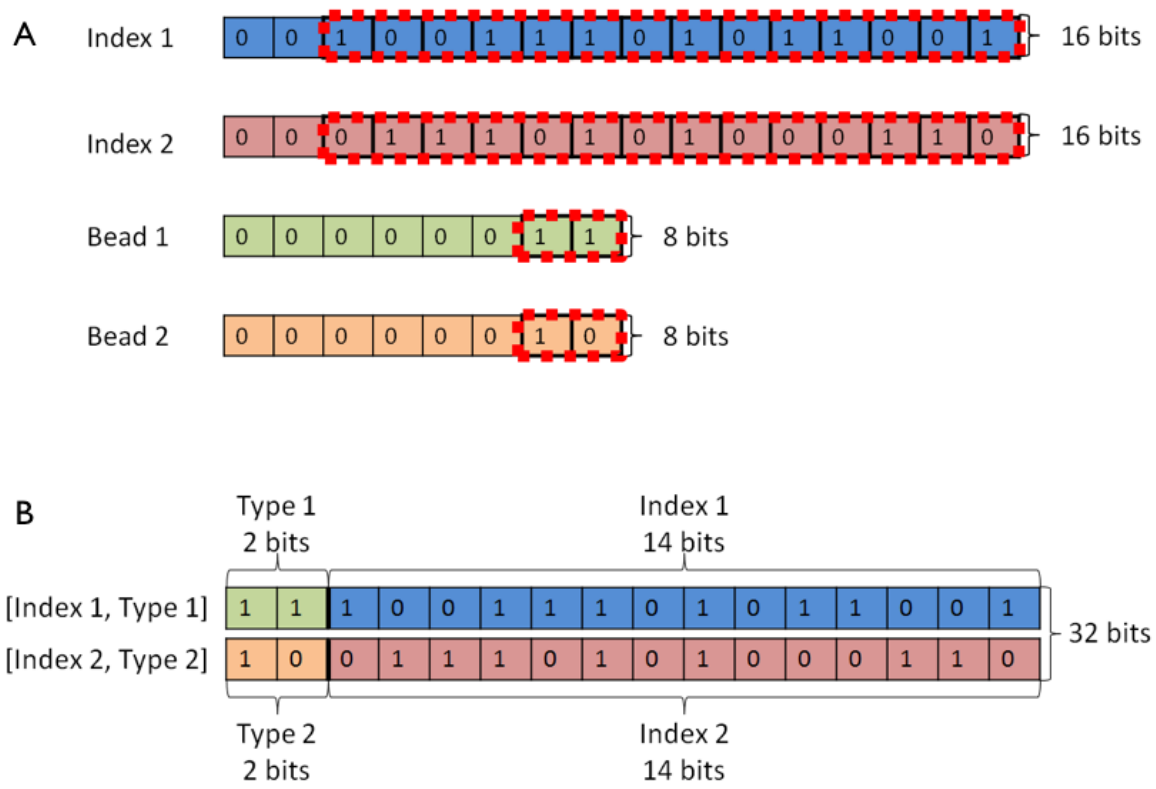

Figure 3: Schematic of the type compression optimization for representing the indices and types of the interacting beads. (A) A total of 48 bits are required when using two 16-bit ushorts and two 8-bit uchars to represent two interaction indices and two interaction types. A total of 16 bits will always be unused. The bits that will be used in the compressed type is highlighted in red. (B) Only 32 bits are required when using a single 32-bit ushort2 to represent two interaction indices and two interaction types. Note that there are no unused bits when simulating the 70s ribosome (10,219 beads).

of three different types of interactions, namely the protein-protein, protein-RNA, and RNA-RNA interactions, meaning that no more than 2 bits would be necessary to represent these three types. The smallest available integral data type in C/C++ and CUDA is the uchar (unsigned character), which is an 8-bit data type capable of representing $2^8 = 256$ different values. If we used uchar to represent the three different interaction types, 6 bits will always be unused, leading to a waste of storage space and increase in memory transfer time (Fig. 3A).

Since the indices for the beads in the 70s ribosome could be represented by using 14 bits and the type of interaction could be represented by using only 2 bits, we developed an approach to combine these two variables into a single 16-bit ushort value to completely eliminate the wasted space when using a ushort and a uchar to represent each of them individually [15]. The total memory requirement for each entry is reduced 33%, and the number of read and write operations is reduced by half because only one data type will be transferred instead of two.

In each interaction there are a total of two different beads that must be stored, so two ushort values will need to be stored. CUDA provides an implementation of a ushort2 data type, which is essentially two 16-bit ushort values combined into a 32-bit data type. The ushort2 data type can be thought of a two dimensional vector with an x and y component. Using a ushort2 instead of two ushort values takes up the same amount of memory, but can reduce the number of memory reads required to transfer data from main memory to thread-level memory, depending on access patterns. These components can be accessed in order to extract either of the two ushort values stored in the ushort2.

We combined the interaction index and interaction type into a single x or y component of a ushort2 by using the 2 highest-order bits of one of the 16-bit

ISBN: 978-1-62561-001-0

```
//The 2-bit type value and the 14-bit index value are
//combined into a single 16-bit value by left shifting
//the type value and then ORing the results with the
//14-bit index value.
#define COMBINE(type, idx) ((type << 14) | idx)

//The type of interaction stored in a type compressed
//value can be retrieved by right shifting the value
//by 14 bits
#define GET_TYPE(combined) ((combined) >> 14)

//The index of an interaction stored in a type compressed
//value can be retrieved by ANDing the value with 0x3FFF
//(equivalent to the binary value 0011 1111 1111 1111),
//which sets the two high-order bits to 0
//while leaving the 14 low-order bits unchanged.
#define GET_IDX(combined) ((combined) & 0x3FFF)
```

Figure 4: Type compression and decompression code in C.

ushort2 components to store the type of the interaction and the 14 low-order bits to store the identifying number of the bead (Fig. 3B). These values are combined by setting the component's 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 identifying bead number. Once these values are stored in a ushort2 component, they can be retrieved very easily. To find the type of the interaction that is stored in the component, 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 2 highest-order bits to zero, leaving the 14 lowest-order bits unchanged (Fig. 4).

2 PARALLEL NEIGHBOR LIST ALGORITHM

The calculation of the Lennard-Jones forces can be the most computationally intensive portion of a MD simulation because it is evaluated between pairs of all particles in the systems i and j, resulting in an algorithm that scales $O(N^2)$. However, when the particles are sufficiently far apart from each other, the interaction force between them is effectively zero. Noting this, the Verlet neighbor List Algorithm [16] first calculates the distance between each pair and constructs a subset neighbor list of particle pairs whose distances is within a "skin" layer radius, r_l. The neighbor list is updated every n time steps, and only the interactions between pairs of particles within a distance cutoff radius, r_c, are computed, resulting in a further subset pair list (Fig. 5). Verlet's original paper [16] chose 2.5σ and 3.2σ for r_c and r_l, respectively, where σ is the radius of the interacting particles. The resulting algorithm then scales $O(Nr_c^3) \sim O(N)$.

Though many portions of MD simulation algorithms lend themselves to parallelization, the algorithm used to calculate the neighbor and pair lists in the Verlet neighbor list algorithm cannot be modified to run in parallel, at least in the form originally introduced by Verlet [16]. In that algorithm, computing a subset list is inherently sequential (Fig. 5C). It can be computed on the GPU, but a single GPU core is very slow. Alternatively it could be computed on the CPU, but the resulting list must be transferred back to the GPU, a major bottleneck as we described above. We therefore developed an algorithm that involves only parallel operations so that it can be implemented entirely on the GPU [15] (Fig. 5D).

In the parallel neighbor list algorithm, we take advantage of the highly optimized GPU library functions in CUDA Data Parallel Primitives Library (CUDPP), namely the key-value sort and parallel scan operations. The first step is to perform a key-value sort on the data using the Member List as keys and the Master List as the values. Those interactions represented in the Master List are evaluated to be within the distance cutoff and the Member List holds values of "true" or "false" and represented with a zero or one. A sorting of these values in numerical order will place the "true" values at the top of the Member

ISBN: 978-1-62561-001-0

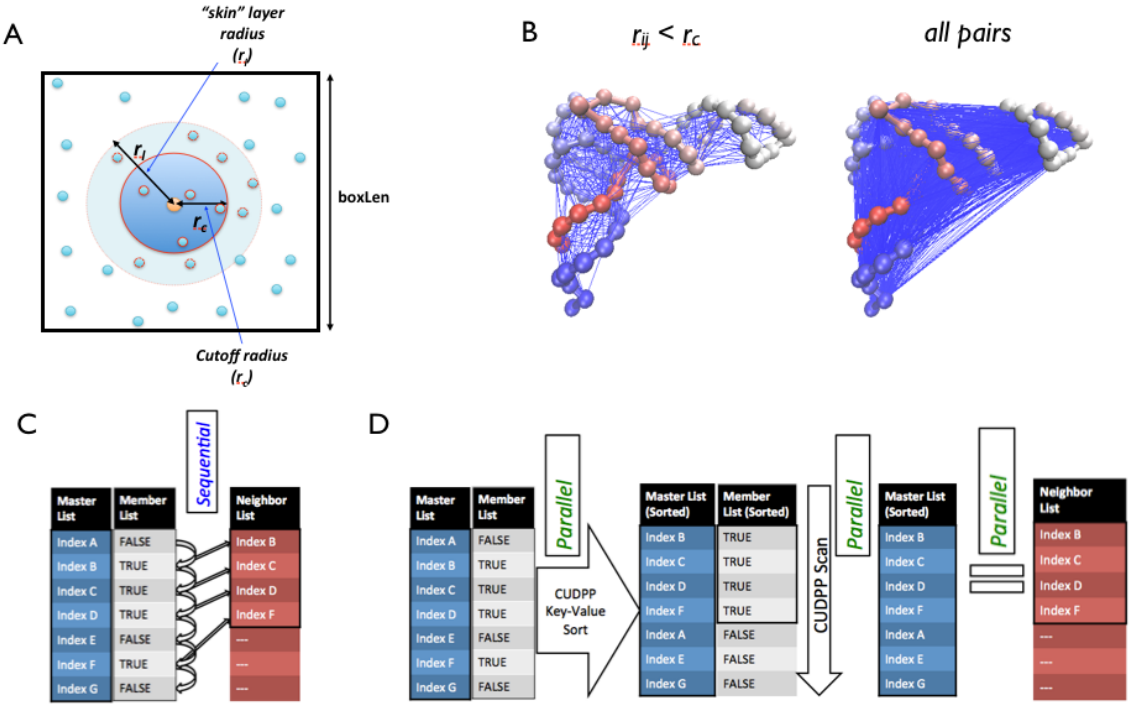

Figure 5: Verlet neighbor list algorithm (A) In the original Verlet neighbor list algorithm, out of a master list of all pairs of possible interactions, a neighbor list is constructed of interactions within a "skin" layer radius (r_l) distance cutoff. The neighbor list is updated every n timesteps from which a pair list is constructed of interactions within a cutoff radius (r_c) distance cutoff. At every timestep, the forces are evaluated only for the members of the pair list. (B) The structure of $tRNA^{phe}$ is shown with blue lines indicating members of the pair list (left) and all possible interaction pairs (right). (C) A schematic of the original Verlet neighbor list algorithm that involves an inferentially sequential implementation. (D) A parallel neighbor list algorithm that is specifically optimized for GPUs that involves only parallel operations.

List and the "false" values at the bottom. The next step would be to copy only those interactions that are within the distance cutoff to the Neighbor List in parallel, but one must first know how many interactions to copy. We therefore perform the parallel scan operation to produce the total number of "true" interactions and then copy only those interactions to the Neighbor List.

Overall, the parallel neighbor list algorithm results in a neighbor list that is equivalent to the one created by the original Verlet neighbor list algorithm, and it only consists of parallel operations that can be performed entirely on the GPU without transfer of information to and from the CPU that would degrade the performance.

3 PERFORMANCE SPEEDUP OF GPU-OPTIMIZED SOP MODEL MD SIMULATIONS IS N-DEPENDENT

We implemented the parallel neighbor list algorithm in a SOP Model MD simulation code that is optimized for the GPU and compared it to an equivalent SOP Model MD simulation code with the Verlet neighbor list algoirthm. We then performed 1 million timesteps of MD simulations for biomolecular systems of varying size. The biomolecular systems include a $tRNA^{phe}$ (76 beads), 16s ribosome (1530 beads), 30s ribosome (3,883 beads), 50s ribosome (6,336 beads), and 70s ribosome (10,219 beads). To evaluate the contributions of the individual components of the MD simulation execution, we calculated the execution times for the force evaluation, neighbor and pair list evaluations of the parallel neighbor list algorithm, the position and velocity updates, and general logging and I/O (Fig. 6A). The force and the neighbor and pair list evaluations of the parallel

ISBN: 978-1-62561-001-0

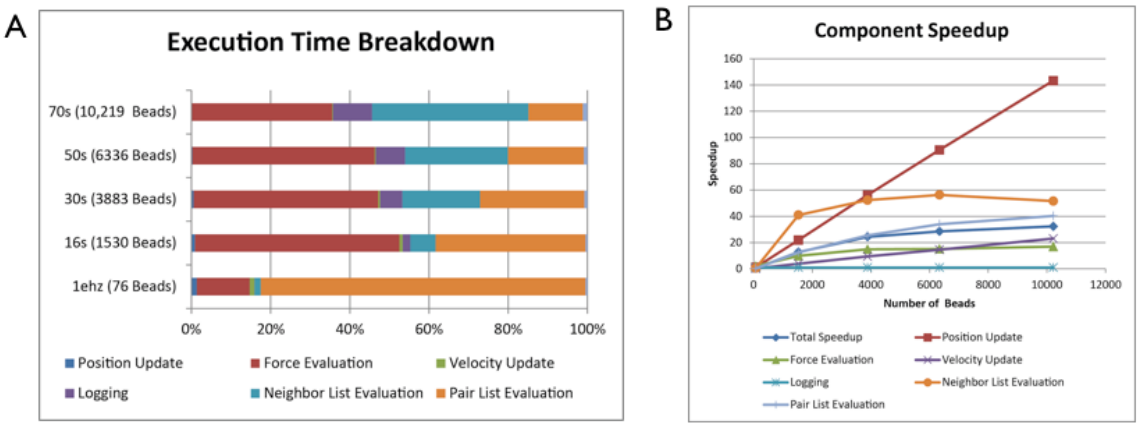

Figure 6: Relative execution times and speedup of different components of the SOP model MD simulations with the parallel neighbor list algorithm. (A) The percentages of the execution times are shown for different components of the GPU-optimized MD simulations biologically relevant systems. (B) The speedup of the different components of the GPU-optimized MD simulations are shown as compared to the equivalent single quad-core CPU-optimized MD simulations.

neighbor list algorithm accounts for almost the entire overall execution time of the MD simulations in all cases.

Then, we compared the N-dependent performance of the GPU-optimized versus the CPU-optimized MD simulations to calculate the overall difference between the two approaches. We observe an N-dependent speedup in which the smallest system we studied, the tRNA^{phe}, is actually slower on the GPU, presumably because any gain in the speedup of the calculations on the GPU does not overcome the time it takes to transfer of information to the GPU. In the full 70s ribosome, however, we observe a ∼30x speedup in the GPU-optimized MD simulations as compared to the CPU-optimized version. Furthermore, the Pair List and Neighbor List performances speedups were ∼25x and ∼55x, respectively (Fig. 6B). Interestingly, the speedup of the calculations of the new positions was ∼145x (Fig. 6B), but it contributes very little to the overall speedup because it represents a small portion of the overall execution time (Fig. 6A).

4 PERFORMANCE COMPARISON WITH HOOMD

To benchmark the performance of the parallel neighbor list algorithm, we compared the performance of our simulation code with that of HOOMD. HOOMD is a widely used general purpose particle dynamics simulation software suite that is implemented on GPUs. Its versatile and flexible code consists of many different types of potentials that include the harmonic bond and angle and Lennard-Jones potentials. As such, the SOP model can be largely implemented on HOOMD. The energy potential of the SOP model as we implemented in the HOOMD simulation code is as follows:

$$
\begin{aligned}
V(\vec{r}) =& V_{bond} + V_{angle} + V_{VDW}^{ATT} + V_{VDW}^{REP} \\
=& \sum_{i=1}^{N-1} \frac{k_r}{2} \left(r_{i,i+1} - r_{i,i+1}^0\right)^2 \\
&+ \sum_{i=1}^{N-2} \frac{k_\theta}{2} \left(\theta_{i,i+1,i+2} - \theta_{i,i+1,i+2}^0\right)^2 \\
&+ \sum_{i=1}^{N-3} \sum_{j=i+3}^{N} \epsilon_h \left[\left(\frac{r_{i,j}^0}{r_{ij}}\right)^{12} - 2\left(\frac{r_{i,j}^0}{r_{ij}}\right)^{6} \right] \Delta_{i,j} \\
&+ \sum_{i=1}^{N-3} \sum_{j=i+3}^{N} \epsilon_l \left(\frac{\sigma}{r_{i,j}}\right)^6 (1 - \Delta_{i,j})
\end{aligned}
$$

The first and second terms are the short-range harmonic bond and angle potentials The parameters are: $k = 20$kcal/mol and $r_{i,i+1}^0$ and $\theta_{i,i+1,i+2}^0$ is the distance and angle between neighboring beads in the folded structure, and $r_{i,i+1}$ and $\theta_{i,i+1,i+2}^0$ are the actual distance and angle between neighboring beads at a given time t, respectively. While these terms are explicitly different, they serve the same purpose as the original SOP model we implemented in our MD simulation code. Furthermore, we expect qualitatively

ISBN: 978-1-62561-001-0

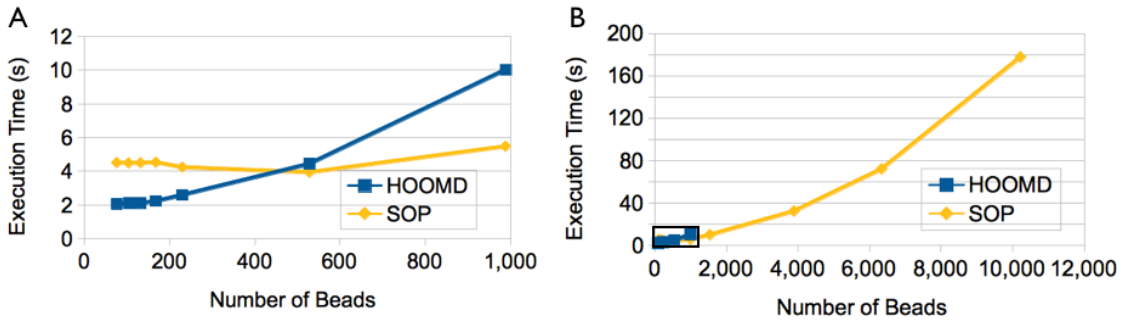

Figure 7: Execution times comparisons with HOOMD. (A) Comparison of execution times of the SOP Model implemented in our MD simulation code as compared to the SOP model implemented in HOOMD for system sizes up to ∼1,000 beads, which was the maximum hardware limit on the NVIDIA Tesla C2070. (B) The same comparison for systems sizes up to ∼10,000 beads, which is the maximum hard limit for our simulation code on the NVIDIA Tesla C2070.

identical simulation results.

The third and fourth terms are the Lennard-Jones interactions that are identically implemented. As such, the performance scales $O(N)$. The HOOMD code is designed to accommodate the Lennard-Jones interactions of a homogeneous and heterogeneous particle systems of different types of particles with different $r^0_{i,j}$. We note, however, that the implementation of the Lennard-Jones interactions requires a different type for each interaction pair because the $r^0_{i,j}$ is different for each native interaction pair, and the memory storage scales as $O(N^2)$. The HOOMD code was not designed to handle this many different "types" of interactions, and the current release version of the code limits the number of types so that the parameters fit in shared memory. We therefore explicitly removed the limit so that the GPU hardware memory will instead determine the limit on the number of types.

We implemented the SOP model into HOOMD and compared our simulation code to that of HOOMD. For the neighbor list algorithm, we ensured that the number of particles were approximately the same by comparing the average number of particles in the neighbor list for each particle in both our MD simulation code and in HOOMD. We chose this measure for comparison because the HOOMD code already performed this calculation, however, we do not anticipate significant differences in our conclusions with other reasonable measures. Since the original HOOMD code has a limit on the number of different types of interactions, we trivially modified the code to remove this software limit. However, there also exists a hardware limit in that the memory available to the GPU is finite. In the NVIDIA C2070 we used for our simulations, that limit is 6 GB of memory.

For systems less than ∼400 beads, HOOMD has a lower execution time than our MD simulation code (Fig. 7A). However, for larger systems up to ∼1,000 beads, our simulation code execution time is markedly less. After that point, the HOOMD code can no longer execute the simulations due to memory limits. However, our simulation code is able to handle larger systems because of the type compression and other optimization techniques we used to reduce memory transfer bottlenecks, and we again observe N-dependent execution times up to ∼10,000 beads (Fig. 7B).

III CONCLUSIONS

We analyzed the performance of a SOP model MD simulation code with a parallel neighbor list algorithm on the GPU. We observe that the force evaluation and the neighbor list calculations comprises of the largest percentage of the overall execution times. When compared to an equivalent CPU-optimized SOP model MD simulations on a single core, we observe an N-dependent speedup in which the smallest systems are actually slower on the GPU but the largest system (10,219 beads) we studied is 30x faster. The pair and neighbor list calculations of the parallel neighbor list algorithm calculations were observed to have speedups of 25x and 55x, respectively.

We next implemented the SOP model algorithm into HOOMD, a widely used leading general particle dynamics simulation software suite, to benchmark the performance of our simulation code. Since HOOMD is was not originally intended to be optimized for the SOP model algorithm, our results would be expected to favor our MD simulation code. We observe

ISBN: 978-1-62561-001-0

an N-dependent execution times in which HOOMD performs the simulations faster for smaller systems (∼400 beads), but our simulation code is faster for larger systems up to ∼1,000 beads, which is the limit of what can be performed on a NVIDIA Tesla C2070 due to hardware memory constraints. However, our present simulation code can accommodate systems of ∼10,000 beads.

ACKNOWLEDGMENTS

TJL acknowledges financial support from the Wake Forest University Computer Science Graduate Fellowship for Excellence. JAA was supported by the DOD/AD(R&E) under Grant No. N00244-09-1-0062. SSC is grateful for financial support from the National Science Foundation (CBET-1232724) and the Wake Forest University Center for Molecular Communication and Signaling.

References

[1] C. B. Anfinsen, "Principles that govern the folding of protein chains", *Science*, vol. 181, no. 96, pp. 223–30, 1973.

[2] 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*, vol. 29, no. 2, pp. 116–25, 2010.

[3] 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*, vol. 8, no. 5, pp. 1542–1555, 2012.

[4] J. A. Anderson, C. D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units", *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342–5359, 2008.

[5] S. Plimpton and B. Hendrickson, "A new parallel method for molecular dynamics simulation of macromolecular systems", *Journal of Computational Chemistry*, vol. 17, no. 3, pp. 326–337, 1996.

[6] C. Hyeon, R. I. Dima, and D. Thirumalai, "Pathways and kinetic barriers in mechanical unfolding and refolding of RNA and proteins", *Structure*, vol. 14, no. 11, pp. 1633–1645, 2006.

[7] 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*, vol. 84, pp. 203–50, 2008.

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

[9] 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*, vol. 132, no. 5, pp. 1526–8, 2010.

[10] 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*, vol. 298, no. 5, pp. 937–53, 2000.

[11] 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*, vol. 52, pp. 499–535, 2001.

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

[13] A. Zhmurov, A. Brown, R. Litvinov, R. Dima, J. W. Weisel, and V. Barsegov, "Mechanism of fibrin(ogen) forced unfolding", *Structure*, vol. 19, no. 11, pp. 1615–1624, Nov. 2011.

[14] M. Guthold and S. S. Cho, "Fibrinogen unfolding mechanisms are not too much of a stretch", *Structure*, vol. 19, no. 11, pp. 1536–1538, Nov. 2011.

[15] 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*, pp. 321–328, 2012.

[16] L. Verlet, "Computer "Experiments" on classical fluids. i. thermodynamical properties of Lennard-Jones molecules", *Physical Review*, vol. 159, no. 1, pp. 98, 1967.

ISBN: 978-1-62561-001-0