A Highly Extensible Framework for Molecule Dynamic Simulation on GPUs

Xiao Zhang[1], Wan Guo[1], Xiao Qin[2] and Xiaonan Zhao[1]
[1]School of Computer Science Northwestern Polytechnical University 127 Youyi xi Road, Xi'an Shaanxi China
[2]Department of Computer Science and Software Engineering Auburn University, AL 36849-5347
E-mail: zhangxiao@nwpu.edu.cn;xilouyouki@163.com;xqin@auburn.edu;zhaoxn@nwpu.edu.cn

Abstract— *Molecular dynamics (MD) was widely used in chemistry and bio molecules. Numerous attempts have been made to accelerate MD simulations. CUDA enabled NVIDIA Graphics processing units (GPUs) use as a general purpose parallel computer chips as CPU. But it is not easy to port a program to GPU. We present a highly extensible framework for molecular dynamics simulation. And we discuss how to accelerate the process of port to GPU. We introduce how to find the performance battle and how to port the time costly procedure to GPU. We discuss about how to decrease the memory usage in GPU and how to improve the maintenance of molecular dynamics simulation. At last, we present the performance of linear and parallel simulation with different number of molecules. Source codes can be found at https://github.com/orlandoacevedo/MCGPU.*

Keywords: Molecular dynamics; Maintainability; Reproducible

1. Introduction

Graphics processing units(GPU) originated as specialized hardware to accelerate graphical operations. GPUs typically handle computation for graphics, such as 3D rendering and ray tracing. General-purpose computing on graphics processing units (GPGPU) is to perform computation in applications traditionally handled by the central processing unit (CPU). OpenCL is currently the dominant open general-purpose GPU computing language. The dominant proprietary framework is Nvidia's CUDA.

The CUDA architecture is built around a scalable array of multi-threaded Streaming Multiprocessors (SMs). It generates multiple threads on multiple processors. It uses Single Instruction Multiple Thread (SIMT) architecture. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel codes for independent, scalar threads, as well as data-parallel codes for coordinated threads.[1]

Molecular dynamic(MD) is widely used in chemistry and biomolecules. It is a compute intensive application. This kind of application can be accelerated by GPUs. There are several different algorithms used in MD. Several previous studies have implemented special algorithms on GPUs. Stone *et al.* introduce GPU-accelerated applications of electrostatics, molecular dynamics, and quantum chemistry[2]. ACEMD is a commercially licensed biomolecular dynamics software package; it is designed for execution on a single workstation with multiple GPUs. It appears to be most effective for system sizes of 10K to 100K atoms[3]. Folding@home is an early project developed on GPUs for molecular dynamics. They worked with ATI since 2005, and they have ported and optimized with CUDA for NVIDIA hardware. These kernels are also deployed in OPENMM software library. [4]. NAMD is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. It scales to hundreds of processors on high-end parallel platforms[5]. HOOMD is a freely available software designed for GPU execution [6]. It speeds up of over a factor of 30 compared to LAMMPS[7]. Elsen *et al.*, implemented a simple implicit solvent model (distance dependent dielectric) [8]. Stone *et al.* have examined a GPU implementation for molecular modeling[9]. Anderson *et al.* have implemented several algorithms, including integrators, neighbor lists, Lennard-Jones, and bond forces[10].

The BOSS program is a general purpose molecular modeling system that performs molecular mechanics (MM) calculations, Metropolis Monte Carlo (MC) statistical mechanics simulations. The MC simulations can be carried out for pure liquids, solutions, clusters, or gas-phase systems; typical applications include computing properties of a pure liquid, free energies of solvation, effects of solvation on relative energies of conformers, changes in free energies of solvation along reaction paths, and structures and relative free energies of binding for host-guest complexes.[1].

2. Challenges of Porting to GPU

GPUs offer high performance parallel computing capacities. There are several difficulties in applying GPU on big scale MD simulations.[11]. Some of the challenges are still present after four years.

2.1 Integration with Original System

There are many MD simulation systems running on CPU. MD simulation systems are compute-intensive systems. GPUs have a huge advantage for these kinds of systems. Many simulation systems have migrated to run

[1]http://zarbi.chem.yale.edu/software.html#boss

on GPU. There are several methods to migrate current systems on GPUs (e.g, plugin, rewrite). Porting a current system allows a legacy code to take advantage of accelerators without rewriting the entire thing. In some cases, the effect of GPU performance improvements will be decreased by too many data transmission between CPU and GPUs. AMBER 11 begins to use NVIDIA GPU to massively accelerate PMEMD for both explicit solvent PME and implicit solvent GB simulations[2]. NAMD is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems based on Charm++ parallel objects[12]. GROMACS uses OpenMM acceleration library and plugins to run simulations on GPU[13]. LAMMPS use Geryon library to support GPU. It also allows portability to AMD accelerators, CPUs, and any future chips with OPENCL support[2].

2.2 Scale

Algorithms used for MD are traditionally evaluated based on how they scale with the number of atoms being simulated. GPUs have enough compute units to handle small or medium sized proteins.

One problem is how to increase the number of atoms that a GPU can simulate. It needs lots of memory to store atom states before computing. In order to compute in GPU, it needs another copy of data. These halve the maximum computable number of atoms in theory. In a big simulation process, the target can be divided into smaller areas and simulate by serial. But status of all atoms should be generated and stored at the beginning. Virtual memory can help avoid the shortage of memory; it stores big arrays in disk instead of physical memory. But it delays the transfer between memories to CPU.

Another problem is how to make several GPUs work together in one simulation process. GPU could not communicate with other GPUs directly before 2010. For example, If GPU in node A(GPU-A) need to communicate with GPU in node B(GPU-B). GPU-A needed to pass the data to a CPU in the same node(CPU-A). The CPU-A send the data to a CPU in another node(CPU-B). Then the CPU-B would transfer the data to GPU-B in node B. Most popular method divides the computing to several tasks run on different CPU. Communication between GPU and CPU should be kept to a minimum. NVIDIA provide GPUDirect; it adds support for peer-to-peer communication between GPUs through InfiniBand cards. Using GPUDirect, 3rd party network adapters, solid-state drives (SSDs), and other devices can directly read and write CUDA host and device memory, eliminating unnecessary system memory copies and CPU overhead.

MGPU is a C++ programming library targeted at single-node multi-GPU systems[14]. Such systems combine disproportionate floating point performance with high data locality and are thus well suited to implement real-time algorithms. They have a speed-up of about 1.7 using 2 GPUs and reach a final speed-up of 2.1 with 4 GPUs.

[2]http://ambermd.org/gpus/

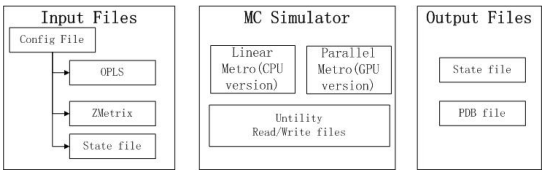


Fig. 1: Input/Output of MC Simulator

3. Design

The simulator is used to find a stable state of molecules in solvation box. The core theory of the simulation about solvation is that the energy will keep decreasing until a stable state during the movement of molecules. The process is shown below:

1) Initialization of simulator: This step places a given number of molecules in a box. The position and angle of molecules are generated at random.
2) Calculate energy of all molecules: This step calculates the energy of all molecules.
3) Random movement of molecule: This step chooses one molecule at random. Then it moves and rotates the selected molecule.
4) Calculate new energy after movement: This step calculates the energy of new state.
5) Judgment of movement: If the new energy is less than the old energy, then the movement is acceptable, use the new state to continue. If the new energy is larger than old energy, then it has a little possibility to accept it.
6) If the simulation step is less than given step number, go to step 2.

The first step is to find which part is most time costly. GNU gprof is a profiler tool on Linux. It can find where the program spent its time and called times of each functions. It inserts code at the beginning and end of each function to collect timing information. After the program is complied with special option (-pg), the program will generate information needed for gprof. Simply run the program as usual; it will generate the performance data and write it into a file called 'gmon.out'. gprof can interpret the data and output a table listing the function name, call times and time used. From these, we can find that the calculation of energy is a compute intensive task.

3.1 Object Oriented

With compiler nvcc provided by NVIDIA, we can compile source codes including the host code and device code. For the host code, nvcc supports full features of object oriented designs. But for the source code that runs on device, nvcc

supports features of data aggregation class and derived class. It does not support run time type information (RTTI) and the C++ standard library. Seiller *et al.* present an object oriented framework for GPGPU-based image processing[15]. They created an interface for classes used in GPU and implemented different classes on CUDA and GLSL. MinGPU proposed a general purpose computation library based on object oriented framework[16]. But they do not support object oriented.

To compare simulation results and performance improvement, we implement serial and parallel methods to simulate the random movement of molecules. These two methods share most source codes runs on CPU, and the parallel simulation runs on GPU to calculate the energy of molecules. We want to apply the object oriented design during development. We implement GPU acceleration in C++ and CUDA. We have more experience in C++ than CUDA. So we implement a C++ version simulator without parallel. Then we add CUDA code to implement parallel compute on GPU. To make it easy to maintain and develop, we wanted to

1) minimize the amount of coding required
2) simplify the methods to port different algorithm to parallel
3) use same input files and get same results with two versions.

As shown in Figure 2, we use class BoxState to store all states of atoms including atom position, angles, and bonds. Class Calculator computes the free energy of a given state. Class generator moves molecules in the box and decides if the movement is acceptable. Simulator initiate the state of BoxState and calls Generator repeatedly by the given steps. All of these classes work together to simulate molecule solution in the box on CPU. We reuse class BoxState and Generator in the GPU version. Class used in GPU cannot be derived from class used in CPU. So class GPUBoxState is not the subclass of BoxState; it depends on BoxState. The class allocates memory in GPU and stores states in it. It creates a BoxState object to save states in CPU and synchronizes the data to GPU. The class GPUBoxState is just a wrapper of BoxState; all states of atoms are saved in BoxState. Many functions are reused just like the linear method, such as save/load atom state from disks and move molecules.

3.2 Memory model

There are $6.02*10^{23}$ molecules in 1 mol of water. The maximum system size that can be treated with the GPU is limited by the memory size. The CUDA programming model assumes that both the host and the device maintain their own separate memory spaces in DRAM. In particular, Langevin temperature regulation and the use of larger cotoffs for the effective Born radii calculations increase the memory requirements. By using AMBER, Tesla C2070 with 6.0 GB GPU memory can treat 54,000 atoms[17].

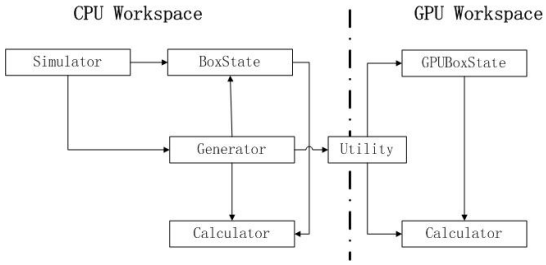


Fig. 2: Class structure of the Simulator

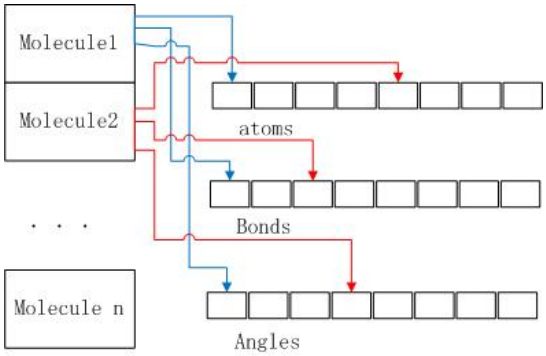
Name	Description	size
molecule	state of molecule, pointers to atoms and other structure	72
atoms	position and type of atoms in molecules	56
bonds	bond between 2 atoms	24
angles	angle of 2 adjacent atoms	24
dihedrals	the angle created by two planes	24
hops	atom pairs and their node distance(hops) away from each other	12

Table 1: Data Structure size

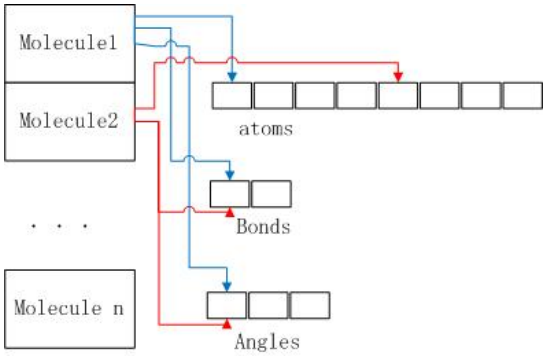
In our simulation, we need store molecule state, including atoms, bonds, angles, dihedral, and hops. We list the size of each structure in Table 1. As for a water molecule, it needs 5 atoms (2 are dummy), 4 bonds, 3 angels, 2 dihedrals, and 2 hops. The memory usage can be calculated by Formula 1.

$$\begin{aligned} Mem_{molecule} &= \sum(size_{items} * num_{items}) + size_{molecule} \\ &= 5 * 56 + 4 * 24 + 3 * 24 + 2 * 24 + 2 * 12 \\ &+ 72 \\ &= 592 \\ Mem_{simulator} &= Mem_{molecule} \times num_{molecules} \end{aligned} \tag{1}$$

To simulate 10,000 molecules (50,000 atoms), it needs 5,920,000 bytes. In our simulation, the atom is the most important element during computing. It stored the 3-dimension position, sigma and epsilon used in LJ calculations. The angles and hops between atoms may change in the real world, but in a simplified model, we can assume the molecules move as a whole, which means bond, angle, dihedral, and hop are equal within all molecules of the same kind. By merging all of these items into one block and having all molecules point to it, we can get a simplified memory usage formula 2. To simulate 10,000 molecules (50,000 atoms), it needs 3,520,000 bytes, about 3.36GB. It decreases 40% of the memory usage. This optimization does not affect the compute efficiency. Because the relative position of atoms in one molecule is same, we can also save position information in the molecule instead of atoms. To simulate 10,000 molecules (50,000 atoms), it needs 2,560,000 bytes, about 2.44GB. This optimization has bad influence on compute speed, because it needs to calculate the position of each atom before use them.



(a) Memory model of molecules



(b) Optimized Memory model

Fig. 3: Memory Model for Simulation

$$
\begin{aligned}
Mem_{molecule} &= \sum(size_{atom} * num_{atom}) + size_{molecule} \\
&= 5 * 56 + 72 = 352 \\
Mem_{simulator} &= Mem_{molecule} \times num_{molecules} + size_{items}
\end{aligned} \tag{2}
$$

The CPU and GPU memories are in different address spaces. This means data must be synchronized between different address spaces. CUDA provides APIs to copy memory between CPU and GPU, but it is a big performance penalty for these synchronizations. Applications should strive to minimize data transfer between the host and the device. To avoid performance penalty, we identified the changed data during the computation in GPU, and synchronized the changed part. In each procedure of the energy computation, only one atom changed, so what we need is to copy single atom information to GPU, and copy energy results back to CPU memory. Because of the overhead associated with each transfer, many small transmissions are combined into a single large transfer.

In the inner calculating of MD, it uses a molecule i and loops over all molecules j to calculate the minimum image separations. If molecules are separated by distances greater than the potential cutoff, the program skips to the end of the loop. One method is to create a neighbor list for each atom. The list is quite large, and it consists of dimensions

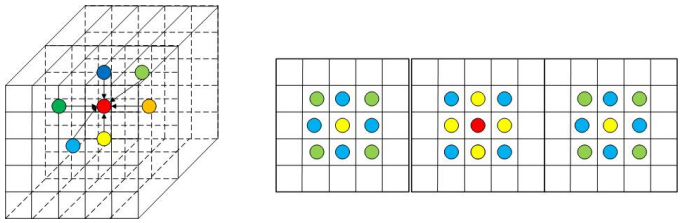


Fig. 4: Neighbor searching in Rubik's Cube structure

roughly $4\pi r\ 3\rho N/6$. Meanwhile it spends lots of time on computing the distance of each pair. We use a cube structure to store the neighbor relations between different atoms. We divided the box into small cubes according to the cutoff size. We can judge which cube hte atoms are in simply by their positions. As shown in Figure 4, if an atom in the central cube(red), we only need to check the atoms in the adjacent ones(yellow) and catercorner ones(blue and green). Assume the molecules distribute equably in the box, the number of atoms in each cube is roughly equal. We can use a list to store the atoms in each cube and map the 3-D cube into a linear data structure. The cube structure can be set up and used rapidly. The search operation allows the programmer to find neighboring atoms within 26 other cubes. The GPU is not used to speed up the search for an individual atom, but instead it is used to run multiple searches in parallel.

3.3 Improvement of Maintainability

The MD simulation tries to find the most probable distribution of molecules. This means an outcome will occur in a proportion of the time it occurs over the long run - this is the relative frequency with which that outcome occurs. Successful simulation using the same model will get similar results after long run. But it is difficult to verify the modification of the algorithm by a long run. It takes too much time to run a whole simulation. On the other hand, it's difficult to say which result is better especially since the difference is so small.

We try to find a way to make the simulation process repeatable. There are two steps using random process. During the initialization of the simulation environment, we use random numbers to place the molecules well-distributed among the volent box. Then we use random numbers to choose which molecule should move and how it will move (position and rotation). We must generate the same random sequence to get the same simulation result. For the initialization of molecule position, we can write the state of each molecule into a state file which is used to initialize the state of other simulation. This makes all the simulations begin with same states. A random seed is a long integer used to initialize a pseudo random number generator. Random seeds are often generated from the state of the computer system (such as the time). If we initialize a pseudo random number generator by a constant seed, the random generator will generate

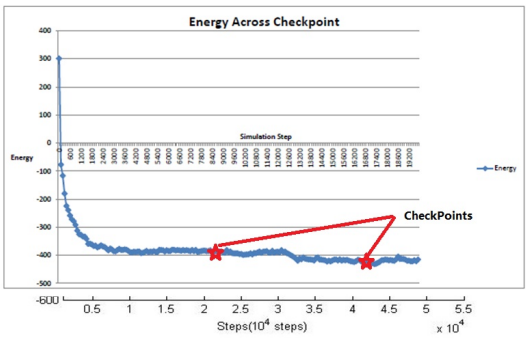


Fig. 5: Illustrate of Checkpoints

the same random sequence. For the random number used in movement, we can use the same random seed used in the previous simulation. And in linear version and parallel version simulators, we use the same generator class to make sure it uses the random sequence in the same method. We output random seeds on screen and the log file. If we want to repeat the simulation process, then we write the state file and random seed in the configuration file, which will get the exact same result as the previous simulation.

4. Performance

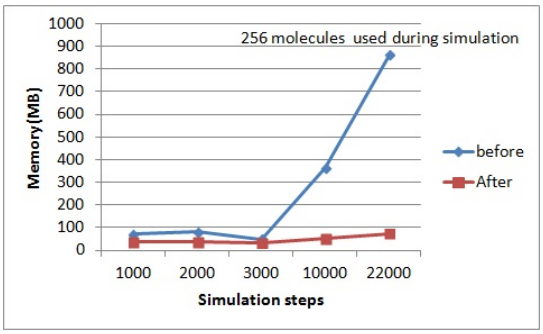
We use dense memory cluster (DMC) in Alabama supercomputer center[18]. The DMC has 1800 CPU cores and 10 terabytes of distributed memory. The DMC has sixteen GPU (Graphic Processing Unit) chips. These are a combination of two Tesla S1070 units (external GPUs connected in pairs to four DMC nodes) and four DMC nodes configured with a pair of Tesla M2070 cards each. These multi-core GPU chips are similar to those in video cards, but there are installed as math coprocessors. This can give significant performance advantages for software that has been adapted to use these processors. Thus the processing capacity of the DMC cluster is: single precision GPU capacity is 18.6 TFLOPS and double precision GPU capacity is 4.8TFLOPS. The job that runs on GPU can use a max of 120GB memory in large serial mode.

4.1 Memory usage

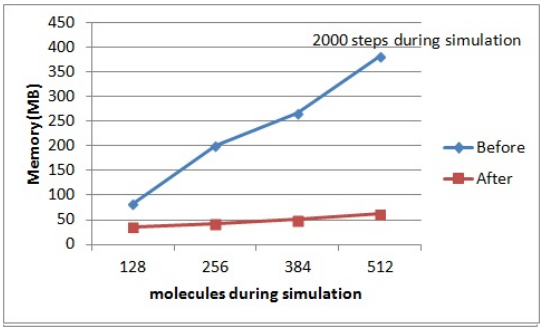
We moved all memory allocation and free memory into BoxState class and GPUBoxState to manage all memory used in GPU. Before the modification, the memory increases with atom number and simulation steps. After the modification, the memory only depends on the number of atoms as shown in Figure 6.

4.2 Development productivity

Unfortunately, it is hard to find a metric to measure the Eventual Efficiency of the design. One metric is how many codes one developer can produce per hour. But for this project, most source codes can be reused. Function



(a) constant molecule numbers



(b) constant steps

Fig. 6: Memory usage under different conditions

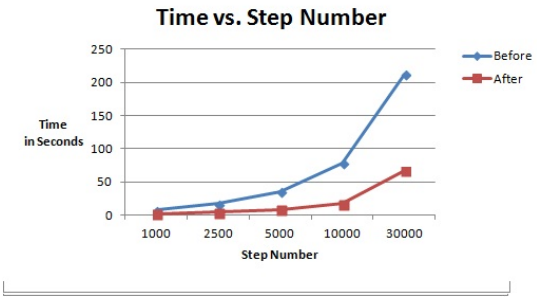
oriented designs may create more source codes than object-oriented. It saves succeeding developer work hours because the solution is easy to comprehend and reuse. With the improvement of maintainability, the correctness of extension and modification can be verified in a few minutes with previous input and output files. It saved many times in the test.

4.3 Performance Improvement

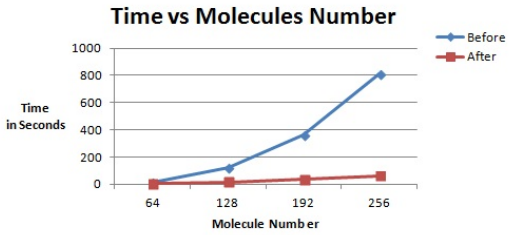
After modifying the search method for neighbor, the time complexity of search algorithm was optimized from exponential growth to linear growth. It is 90% faster after modification as shown in Figure 7.

5. Conclusion

In the paper, we present a framework to accelerate the developing process of porting MC simulation to graphical processing units. The implementation runs on single NVIDA GPU using CUDA program model. In this contribution, we described how to distinguish which part is time-costly. And we discussed how to port these codes to GPU with less program work. We also improved the memory model to make it capable of simulating large systems. Another contribution is that we found a way to make the simulation process repeatable. This modification makes it easy to verify future extension and enhancement.



(a) constant molecule numbers



(b) constant steps

Fig. 7: Performance improvement under different conditions

Limited by the memory size, it can only simulate 54,000 atoms in one GPU. By using MPI, it can simulate large systems by running on different GPUs. There are some limits in our simulation; we did not change the related distance and angle of atoms in one molecule.

Acknowledgments

This work was made possible thanks to the NPU Fundamental Research Foundation under Grant No. JC20110227, the National Science and Technology Ministry No.2011BAH04B05, National High-tech R&D Program of China (863) under Grant No.2013AA01A215, and the National Natural Science Foundation of China under Grant No.61033007. And this work was supported by China Scholarship Council. Xiao Qin's research was supported by the U.S. National Science Foundation under Grants CCF-0845257 (CAREER), CNS-0917137 (CSR), CNS-0757778 (CSR), CCF-0742187 (CPA), CNS-0831502 (CyberTrust), CNS-0855251 (CRI), OCI-0753305 (CI-TEAM), DUE-0837341(CCLI), and DUE-0830831 (SFS).

References

[1] N. Corporation, "Nvidia cuda c programming guide," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2012, [Online; accessed 19-Sep-2012].

[2] J. Stone, D. Hardy, I. Ufimtsev, and K. Schulten, "Gpu-accelerated molecular modeling coming of age," *Journal of molecular graphics & modelling*, vol. 29, no. 2, p. 116, 2010.

[3] M. Harvey, G. Giupponi, and G. Fabritiis, "Acemd: accelerating biomolecular dynamics in the microsecond time scale," *Journal of Chemical Theory and Computation*, vol. 5, no. 6, pp. 1632–1639, 2009.

[4] F. Stanford, "Introduce to folding@home," http://folding.stanford.edu, 2012, [Online; accessed 10-Dec-2012].

[5] J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with namd," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[6] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *Micro, IEEE*, vol. 28, no. 4, pp. 13–27, 2008.

[7] C. Phillips, C. Iacovella, and S. Glotzer, "Stability of the double gyroid phase to nanoparticle polydispersity in polymer-tethered nanosphere systems," *Soft Matter*, vol. 6, no. 8, pp. 1693–1703, 2010.

[8] E. Elsen, M. Houston, V. Vishal, E. Darve, P. Hanrahan, and V. Pande, "N-body simulation on gpus," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 188.

[9] J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *Journal of computational chemistry*, vol. 28, no. 16, pp. 2618–2640, 2007.

[10] J. Anderson, C. 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.

[11] M. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. Beberg, D. Ensign, C. Bruns, and V. Pande, "Accelerating molecular dynamic simulation on graphics processing units," *Journal of computational chemistry*, vol. 30, no. 6, pp. 864–872, 2009.

[12] U. of Illinois, "Namd," http://www.ks.uiuc.edu/Research/namd/, 2013.

[13] GROMACS, "Gromacs openmm," http://www.gromacs.org/Documentation/Installation_Instructions_4.5/GROMACS-OpenMM, 2013.

[14] S. Schaetz and M. Uecker, "A multi-gpu programming library for real-time applications," *Algorithms and Architectures for Parallel Processing*, pp. 114–128, 2012.

[15] N. Seiller, N. Singhal, and I. Park, "Object oriented framework for real-time image processing on gpu," in *Image Processing (ICIP), 2010 17th IEEE International Conference on*. IEEE, 2010, pp. 4477–4480.

[16] P. Babenko and M. Shah, "Mingpu: a minimum gpu library for computer vision," *Journal of Real-Time Image Processing*, vol. 3, no. 4, pp. 255–268, 2008.

[17] A. Gotz, M. Williamson, D. Xu, D. Poole, S. Le Grand, and R. Walker, "Routine microsecond molecular dynamics simulations with amber on gpus. 1. generalized born," *Journal of Chemical Theory and Computation*, vol. 8, no. 5, p. 1542, 2012.

[18] ASC, "Alabama supercomputer center," http://www.asc.edu/, 2012.