Accelerating Molecular Dynamics Simulations with GPUs[*]

John Paul Walters, Vidyananth Balu, Vipin Chaudhary
Computer Science and Engg.
University at Buffalo
Buffalo, NY 14260
{waltersj, vbalu2, vipin}@buffalo.edu

David Kofke, Andrew Schultz
Chemical and Biological Engg.
University at Buffalo
Buffalo, NY 14260
{kofke,ajs42}@buffalo.edu

Abstract

Molecular dynamics simulations are known to run for many days or weeks before completion. In this paper we explore the use of GPUs to accelerate a Lennard-Jones-based molecular dynamics simulation of up to 27000 atoms. We demonstrate speedups that exceed **100x** on commodity Nvidia GPUs and discuss the strategies that allow for such exceptional speedups. We show that traditional molecular dynamics simulations can be greatly improved from a runtime of over 1 day to 18 minutes.

1 Introduction

Molecular dynamics simulations are used to track the evolution of a system of particles based on the interactions between them. It is used in physics, biology, material sciences, applied mathematics and chemistry where systems of up several million atoms are simulated for weeks or months prior to completion. Because of the exceptionally long compute time of MD simulations, it is a popular target for acceleration both using traditional high performance computing techniques, as well as novel architectures [10].

The programmable graphics processor has shown considerable promise for its use in compute-intensive simulations. Their many-core SIMD design is well-suited to numerical and probabilistic simulations such as Monte Carlo and molecular dynamics [16]. With their computational power far outpacing that of the typical CPU, and with the recent addition of double precision floating point hardware to GPUs, it is expected that they will become a standard tool for high performance computing and application acceleration.

Despite the computational power offered by modern graphics processors, they have traditionally been limited to the graphics domain in large part due to their lack of programmability. Until recently, GPUs could only be programmed through the graphics API, such as DirectX or OpenGL. Solutions such as BrookGPU [5] provided a C-like compiler and runtime, but have not proven popular among programmers. Nvidia's CUDA platform, however, has become widely used both inside and outside of academia [11]. Like BrookGPU, CUDA, provides a C-like compiler and runtime. However in the case of CUDA, the compiler is nearly identical to standard C making the use of GPUs an easier choice for application developers.

In this paper we implement a Lennard-Jones-based molecular dynamics simulation on the Nvidia 8800 GTX Ultra GPU using the CUDA platform. We demonstrate that speedups exceeding 100x can be achieved on commodity graphics hardware. As such, we make the following contributions in this paper:

- Implement a highly accelerated MD simulation using Nvidia GPUs.
- Demonstrate high performance and scalability of over 100x on standard graphics processors.

The remainder of this paper is organized as follows: in section 2 we describe the related work and its relation to our implementation. In section 3 we provide a brief overview of molecular dynamics simulations. In section 3.1 we provide the necessary background on the Nvidia 8800 GTX GPU as well as the CUDA architecture. In section 4 we describe our MD simulation implementation and its performance results. Finally, in sections 5 and 6 we provide a brief discussion and our conclusions.

2 Related Work

The use of graphics processors for molecular simulation calculations is starting to receive increasing attention. Stone et al. ported some of the core calculations to an Nvidia GeForce 8800 GTX processor, and achieved a performance of 10x-100x speedup over conventional processors [15]. Andersen et al. have developed a general-purpose molecular dynamics code that runs on the 8800GTX. They found that it runs at a speed equivalent to the performance of a standard general-purpose parallel code (viz., LAMMPS) running on 30 processors [2]. Liu

[*]This research was supported in part by NSF IGERT grant 9987598, MEDC/Michigan Life Science Corridor, and NYSTAR.

et al. demonstrate an accelerated CUDA-based MD simulator that achieved a 15x performance improvement over a conventional CPU [9]. Other architectures, such as the IBM Cell have shown promise for MD simulations. Olivier et al. implemented a GROMACS core on the Cell processor, achieving reasonable speedup [12]. Parallel molecular dynamics have also been shown to scale well on commodity clusters as well as the IBM Bluegene supercomputers [7, 8, 13].

3 Background

Molecular simulations [1, 6] are used in many branches of science and engineering, both as a tool for predictive modeling and as a means to investigate behaviors at the nanoscale. Two basic approaches are in widespread use. The Monte Carlo (MC) method aims to sample an ensemble molecular configuration consistent with the equilibrium distribution, without regard to any mechanical processes that govern the true time evolution of the system. The molecular dynamics (MD) technique is (typically) governed by Newtons laws of motion, and preserves the temporal nature of the behavior. Monte Carlo methods are appealing because they permit a wide variety of non-physical sampling techniques to be applied to improve the generation of new configuration, and because they are easier to extend to new ensembles, such as for systems at a fixed temperature and pressure. Molecular dynamics is, of course, needed if the dynamics is of interest, and MD has several advantages of its own in terms of simplicity, sampling effectiveness, and error-checking.

Molecular simulations are conducted using a wide variety of system sizes, with numbers of atoms ranging from dozens to millions, depending on the problem of interest and the available resources. The core of the calculation is the computation of distances between pairs of atoms, and subsequently the forces and/or energies that they exert on each other. These energies are represented by a simple formula, for example the Lennard-Jones (LJ) potential:

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

Here u is the mutual energy experienced by two atoms separated by a distance r, and σ and ε are model parameters representing the size of the atoms and the strength of their interaction, respectively. Typically the LJ model is used to represent the non-bonded interactions, i.e., those interactions between atoms that are not directly bonded to each other (for which their interaction is represented by another simple model potential). The force exerted by the atoms on each other is given by the gradient of the potential: $F = -\nabla u$. The total energy and force are determined, in principle, by summing over all pairs. In practice, interactions are considered only with atoms in the local vicinity of a given atom, as determined by cell partitioning and/or by keeping lists of neighboring atoms. Also, in most cases periodic boundary conditions are applied such that the simulated system is surrounded by replicas of itself, and interactions are then taken between the nearest images of any pair of atoms. The Lennard-Jones potential can be calculated without taking square roots, which in some platforms can be an expensive operation. Commonly models will also incorporate partial electric charges to represent the electrostatic interactions. The corresponding energies are computed using Coulombs law, which necessitates the calculation of the square root. Additionally the interactions described by these models are very long ranged, with strong interactions felt for distances much larger than the size of the simulated system. Ewald sums and related techniques must be applied to treat these interactions appropriately [1, 6].

A common approach to the parallelization of molecular simulations involves a decomposition in which individual processors are assigned a given region of space, or a given set of atoms. This decomposition can be used for both MD and MC simulations. MC can be parallelized in a more complete way by conducting independent simulations on separate processors with results that can be averaged after all are complete. MD simulations cannot be fully parallelized in this sense, at least if the aim is to generate information about the long-time behavior of the system ("long-time" in MD simulations today corresponds to about 1 microsecond or more of physical dynamics, although strong efforts are now being made to reach the millisecond time scale [4, 14]). Apart from a few specialized instances, there is little recourse for parallelization of the dynamical behavior of an MD simulation—a microsecond can be reached only by running a nanosecond simulation 1000 times longer.

The molecular dynamics code we use in this paper is quite elementary. It simulates a system of atoms (monatomic molecules) governed by the Lennard-Jones potential described in Equation 1. At each time step, the force is calculated between each pair of atoms within a cutoff distance taken to be half the simulation box length. The net force on each atom is used in conjunction with Newton's equation of motion to propagate the system forward in time. The integration is performed using a velocity-Verlet algorithm. No cell or neighbor lists were used as part of the computation, so at each step distances are computed between each pair, and those beyond the cutoff do not contribute to the energy or forces.

3.1 CUDA and the NVIDIA 8800 Architecture

In this section, we briefly summarize the architectural details of NVIDIA GeForce 8800 GTX GPU which was used in our tests and the challenges in realizing the computing power of GPUs. The 8800 GTX is composed 16 multiprocessors with each multiprocessor composed of 8 stream processors resulting in a total of 128 stream processors. The 8800 GTX has 768 MB of DRAM memory along with read only constant and texture memory. Because the global memory space is not cached, proper memory access patterns must be followed in order to achieve maximum memory bandwidth. Reads from constant memory and texture memory are cached, however. Each multiprocessor has 16 KB of on-chip memory that is shared between all stream processors within a single multiprocessor.

Each multiprocessor can concurrently execute up to 768 threads; however, in practice this number will be limited by the number of registers and amount of shared memory used by a thread block. Threads are partitioned into thread blocks of up to 512 threads, with each thread uniquely identified by a thread ID. A thread block can also be specified as a 2 dimensional or 3 dimensional array of arbitrary size with each thread being identified by a 2 or 3 component index(`threadIdx.x`, `threadIdx.y`, and `threadIdx.z`). The `threadIdx` variable is automatically assigned by the CUDA runtime and need not be declared by the programmer.

Although an individual thread block is limited to 512 threads, multiple identical thread blocks executing identical kernels may be batched together into a grid of blocks. Thus, the total number of threads that can be launched in a single kernel invocation is much larger than the 512 thread limitation. Each block can be uniquely identified by its block ID. For blocks of 2 or 3 dimensions, they can be uniquely identified by a 3 component index (blockIdx.x, blockIdx.y, and blockIdx.z). Dimensions of the block grid, `gridDim` and dimensions of the block, `blockDim` are also available as three-component indexes and allow for unique thread identification over the entire grid of thread blocks.

In order to achieve full occupancy of the GPU a kernel must be restricted to use only 10 registers. Therefore a major challenge to developing software for CUDA-enabled GPUs is to minimize register usage. By minimizing the number of registers per-kernel, CUDA is able to maximize the number of simultaneously executing threads. A higher occupancy indicates a greater number of active threads per multiprocessor, and can help to mask latency during global memory loads. Threads in the same thread block can share data through the on chip shared memory and synchronize their execution. Synchronization and communication is not available between thread blocks.

4 Implementation and Performance Results

GPU tests were performed on a machine consisting of a 3.0 GHz AMD Athlon 642 processor with 8 GB RAM and 2 NVIDIA 8800 GTX Ultra GPUs. The sequential MD tests were taken on identical hardware. A single GPU was used for our GPU performance analysis. The output of the simulation is the final position of each atom in the system (tested up to 27,000 atoms). Correctness of each simulation was confirmed to within three decimal places.

Our GPU implementation is split into two kernels: a position/velocities kernel, and a force/velocities kernel. The split-kernel design is used in order to provide a simple barrier between position updates and force calculation, with the velocity calculations split between both kernels. The velocity computations are split between both kernels as it is based on the force of the previous and current iteration.

Because the Nvidia CUDA platform provides only limited synchronization and thread spawning costs are quite low, we elected to construct the MD algorithm in two parts for simplicity. Other synchronization techniques could be implemented that would allow a single kernel design to be used. However, as our speedups (to follow) show, it is not clear that the additional programming burden of such an approach would yield noticeable improvement.

We utilize 8192 active threads, the maximum number of threads supported given our register requirements (16 multiprocessors of 512 threads each). For any simulation size, each thread is responsible for updating the position, force, and velocity of a single atom. For larger simulations with $\mathtt{atoms} > 8192$ we over-subscribe the number of threads allowed on the architecture. In case of over-subscription, batches of 8192 threads are executed. Both kernels are called repeatedly within a loop operating on the host. The loop controls the numbers of simulation steps and is user-controlled.

In Algorithm 1 we provide pseudocode for our GPU implementation of the MD simulation. The GPU kernels are expressed in lines 6-8 and 10-13 of the algorithm. As can be seen, the force kernel (lines 10-13) is the more computationally expensive of the two kernels, requiring `atoms` additional iterations. We next describe the GPU-specific optimizations that were used to accelerate our MD simulation. In the figures to follow, we show results for 1000 steps in each simulation.

The first major optimization applied to the GPU kernels is the use of coalesced memory. Memory coalescing is a technique by which non-sequential and short global memory reads are combined into more efficient and larger sequential global memory reads. This allows the GPU hardware to more efficiently traverse the global memory. Reads by consecutive threads in a warp are combined by the GPU hardware into several, wider memory reads of up to 384 bits each. Consecutive 32-bit reads that are issued simultaneously are automatically merged into multiple 384-bit reads in order to efficiently saturate the memory bus. The neighboring x coordinates of the atoms lie close together and threads operate on this x coordinate in order, leading to coalesced reads for nearly every access to the global memory. The memory has been laid out in such a way that all the values for the x coordinates of the atoms lie close to

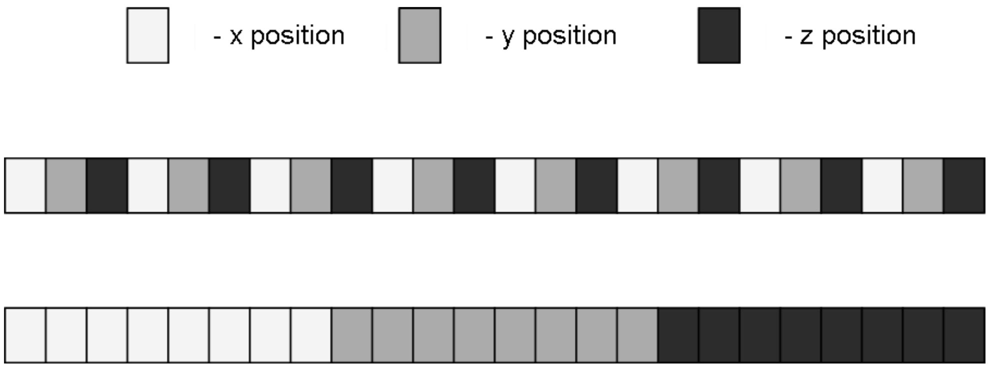

Figure 1: Illustration of non-coalesced memory (top) and coalesced memory (bottom).

Algorithm 1 Pseudo-code for GPU molecular dynamics simulation.

```
1:  Initialize the position, velocity, and force of atoms
2:  Allocate memory for the atoms on the GPU
3:  Transfer force, velocity, and position data to GPU
4:  for all steps do
5:      Launch position GPU kernel
6:      Update head atom positions
7:      Half-update head atom velocity based
        on previous force
8:      Zero head atom forces
9:      Launch force GPU kernel
10:     for all atoms do
11:         Compute force exerted by atom on
            head atom if within cutoff distance
12:     end for
13:     Complete update of head atom velocity
        based on current force
14: end for
15: Transfer data from GPU to CPU
```

each other, followed by the values of the y coordinates, followed by the values for the z coordinates (see Figure 1 for an illustration).

In Listings 1 and 2 we provide examples of non-coalesced vs. coalesced memory accesses. Our memory layout within the GPU implementation of the position updating kernel follows the pattern described in Figure 1 and Listing 2. The code has been optimized to use memory coalescing using the values `idx` and `natoms`. `idx` represents the thread ID produced by evaluating `blockIdx.x` $*$ `blockDim.x` $+$ `threadIdx.x` while `natoms` represents the number of atoms in the system.

Our second major optimization occurs in the force GPU kernel. In this case we were able to achieve significant speedup by placing the coordinates of the head atom (the atom whose velocity, force, and position, a thread is responsible for computing and are referenced in Algorithm 1) directly into registers rather than accessing the head atom from the main memory. As we show, each thread accesses its head atoms `atom` times at each force kernel invocation. At the end of the force kernel, we update main memory with the values stored in the registers. This enhancement increased the number of registers used by the force computation kernel from 10 to 16 registers while reducing the number of threads that can be run concurrently on a multiprocessor from 768 threads to 512 threads. The result is a reduction in the GPU occupancy from 100% to 67%. In Figure 2 we present the performance results due to our optimization of register usage. As can be seen, the use of registers to store the highly accessed head atoms results in significant performance improvements of up to 3.71x over the initial GPU implementation. While the best performance is often achieved with the highest possible GPU utilization, in this case we show that through a careful balance of register usage we can reduce the occupancy while drastically increasing performance.

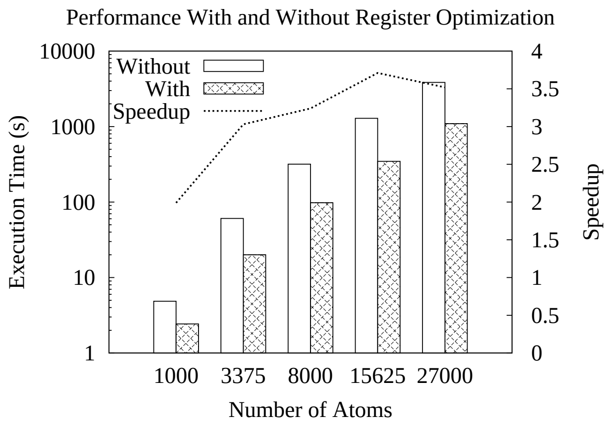

Figure 2: Improvements in speedup due to register optimization.

In Figure 3 we present the overall performance results with varying cutoff distances. Because the simulation operates in reduced units to correspond to atoms of

```
for (n = 0; n<natoms; n++)
   for (i = 0; i<3; i++)
      position [n][i] += deltat *
      velocity [n][i] + deltat*deltat *
      force [n][i] * 0.5;
```

Listing 1: Original loop

```
for (n = 0; n<natoms; n = n + blockDim.x *
gridDim.x)
   for (i = 0; i<3; i++)
      position [n + natoms*i + idx] +=
      deltat * velocity [n + natoms*i + idx]
      + deltat * deltat *
      force [n + natoms*i + idx] * 0.5;
```

Listing 2: Coalesced loop

arbitrary size, we compute box sizes and cutoffs in terms of the atomic diameter of the simulation. We refer to the atomic diameter as σ. While the sequential execution time increases for increasing cutoff values, the execution time on the GPU remains nearly constant. This results in improved speedup for increasing box sizes, varying between a speedup of 74.77x for a cutoff distance of 10σ and 98.04x for a cutoff distance of 20σ.

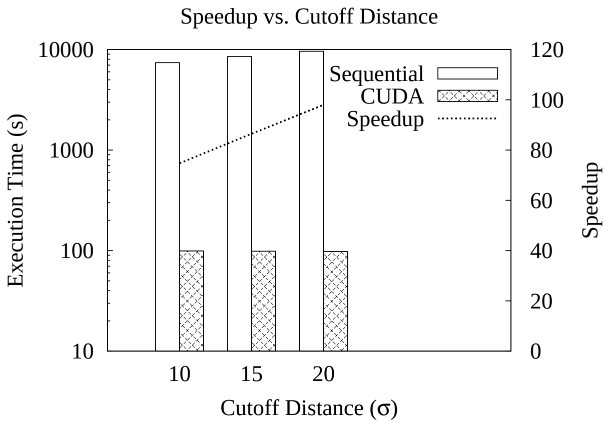

Figure 3: Performance for varying cutoff distance for a system of 8000 atoms with box size 40σ.

In Figure 4 we present the overall results of our MD simulation for varying simulation sizes (number of atoms). Our results include all performance enhancement and optimizations described in this paper. From Figure 4 we observe a maximum speedup of 104.07x for a system of 15625 atoms. As can be seen, we achieve increasing speedups through 15625 atoms before leveling off to 100.19x at $27,000$ atoms.

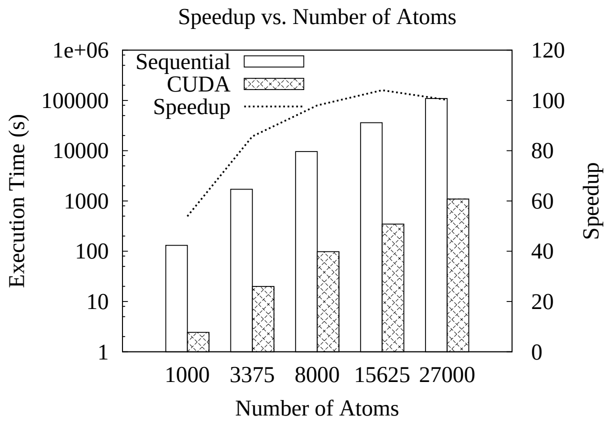

Figure 4: Speedup for varying numbers of atoms.

5 Discussion

As we have shown, numerical simulations such as molecular dynamics are well-suited to manycore parallel architectures such as the GPU. However, the exceptional performance of GPUs can only be realized through proper memory management. Basic strategies, such as memory coalescing, yield major performance improvements with minimal programming effort. More advanced techniques, such as mapping highly used variables to registers may also yield improvements as we showed in Figure 2. However, pinning highly accessed data to registers is only useful in cases where a limited number of data items are accessed many times. Further, significant programming effort is needed in order to balance the register usage per-kernel with the number of threads needed to achieve significant performance improvement. In our case we were able to vastly improve performance by reducing the overall GPU utilization from 100% to 67%. In our case we were able to significantly reduce main memory accesses at the cost of several registers per thread. In cases where the number of main memory accesses cannot be significantly reduced through an increase in register usage, performance may not substantially improve.

6 Conclusion

In this paper we have shown that a typical Lennard-Jones-based molecular dynamics simulation will map naturally onto the manycore architecture of the Nvidia GPU using the CUDA platform. While modest performance improvement can be achieved through a naive port of the algorithm to the GPU, we have shown that a speedup of over **100x** can be achieved through detailed analysis of the algorithm and its memory access patterns. Our next goal is to apply similar techniques to molecular dynamics packages such as GROMACS [3].

References

[1] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, 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), 2008.

[3] H.J.C. Berendsen, D. van der Spoel, and R. van Drunen. GROMACS: A Message-Passing Parallel Molecular Dynamics Implementation. *Computer Physics Communications*, 91:43–56, September 1995.

[4] K.J. Bowers, E. Chow, H. Xu, R.O. Dror, M.P. Eastwood, B.A. Gregersen, J.L. Klepeis, I. Kolossvary, M.A. Moraes, F.D. Sacerdoti, J.K. Salmon, Y. Shan, and D.E. Shaw. Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. ACM, 2006.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *ACM SIGGRAPH*, pages 777–786. ACM, 2004.

[6] D. Frenkel and B. Smit. *Understanding Molecular Simulation: From Algorithms to Applications*. Academic Press, 2nd edition, 2002.

[7] F. Gygi, R.K. Yates, J. Lorenz, E.W. Draeger, F. Franchetti, C.W. Ueberhuber, B.R. de Supinski, S. Kral, J.A. Gunnels, and J.C. Sexton. Large-Scale First-Principles Molecular Dynamics simulations on the BlueGene/L Platform using the Qbox code. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 24, Washington, DC, USA, 2005. IEEE Computer Society.

[8] S. Kumar, H. Chao, G. Almasi, and L.V. Kale. Achieving Strong Scaling with NAMD on Blue Gene/L. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, April 2006.

[9] W. Liu, B. Schmidt, G. Voss, and W. Muller-Wittig. Molecular Dynamics Simulations on Commodity GPUs with CUDA. In *Proceedings of HiPC: The International Conference on High Performance Computing, LNCS*, volume 4873. Springer-Verlag, 2007.

[10] T. Narumi, Y. Ohno N. Okimoto, A. Suenaga, R. Yanai, and M. Taiji. A High-Speed Special-Purpose Computer for Molecular Dynamics Simulations: MDGRAPE-3. *NIC Workshop*, 34:29–36, 2006.

[11] NVIDIA. *Compute Unified Device Architecture (CUDA) Programming Guide*, version 1.0 edition.

[12] S. Olivier, J. Prins, J. Derby, and K. Vu. Porting the GROMACS Molecular Dynamics Code to the Cell Processor. In *IPDPS: The International Parallel and Distributed Processing Symposium*. IEEE Computer Society, March 2007.

[13] J.C. Phillips, G. Zheng, S. Kumar, and L.V. Kalé. NAMD: Biomolecular Simulation on Thousands of Processors. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18. IEEE Computer Society Press, 2002.

[14] D.E. Shaw, M.M. Deneroff, R.O. Dror, J.S. Kuskin, R.H. Larson, J.K. Salmon, C. Young, B. Batson, K.J. Bowers, J.C. Chao, M.P. Eastwood, J. Gagliardo, J.P. Grossman, C.R. Ho, D.J. Ierardi, I. Kolossváry, J.L. Klepeis, T. Layman, C. McLeavey, M.A. Moraes, R. Mueller, E.C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, and S.C. Wang. Anton, A Special-Purpose Machine for Molecular Dynamics Simulation. In *ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 1–12. ACM, 2007.

[15] 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), 2007.

[16] S. Tomov, M. McGuigan, R. Bennett, G. Smith, and J. Spiletic. Benchmarking and Implementation of Probability-Based Simulations on Programmable Graphics Cards. *Computers and Graphics*, 29(1):71–80, 2005.