Accelerating Molecular Dynamic Simulation on Graphics Processing Units

MARK S. FRIEDRICHS,[1] PETER EASTMAN,[1] VISHAL VAIDYANATHAN,[2] MIKE HOUSTON,[3]
SCOTT LEGRAND,[4] ADAM L. BEBERG,[5] DANIEL L. ENSIGN,[2] CHRISTOPHER M. BRUNS,[1] VIJAY S. PANDE[2,5]
[1]*Department of Bioengineering, Stanford University, Stanford, California 94305*
[2]*Department of Chemistry, Stanford University, Stanford, California 94305*
[3]*Advanced Micro Devices, Sunnyvale, California*
[4]*NVIDIA Corporation, Santa Clara, California*
[5]*Department of Computer Science, Stanford University, Stanford, California 94305*

Received 19 September 2008; Revised 12 December 2008; Accepted 13 December 2008
DOI 10.1002/jcc.21209
Published online 3 February 2009 in Wiley InterScience (www.interscience.wiley.com).

Abstract: We describe a complete implementation of all-atom protein molecular dynamics running entirely on a graphics processing unit (GPU), including all standard force field terms, integration, constraints, and implicit solvent. We discuss the design of our algorithms and important optimizations needed to fully take advantage of a GPU. We evaluate its performance, and show that it can be more than 700 times faster than a conventional implementation running on a single CPU core.

© 2009 Wiley Periodicals, Inc. J Comput Chem 30: 864–872, 2009

Key words: GPU, molecular dynamics, implicit solvent

Introduction

Graphics processing units (GPUs) originated as specialized hardware useful only for accelerating graphical operations, but they have grown into exceptionally powerful, general purpose computing engines. Modern GPUs far exceed CPUs in terms of raw computing power.[1,2] As a result, the use of GPUs for general purpose computing has become an important and rapidly growing field of research. Many important algorithms have been implemented on GPUs, often leading to a performance gain of one to two orders of magnitude over the best CPU implementations.[3]

Molecular dynamics simulations of macromolecules are extremely computationally demanding, which makes them a natural candidate for implementation on GPUs. With currently available MD codes, for example, it is impossible to simulate the folding of any but the smallest, fastest folding proteins.[4,5] MD uses a combination of several algorithms. A few previous studies have investigated GPU implementations of specific algorithms used for MD. For example, Elsen et al., implemented a simple implicit solvent model (distance dependent dielectric).[1] Stone et al., have examined a GPU implementation for electrostatics.[2] Anderson et al., have implemented several algorithms, including integrators, neighbor lists, Lennard-Jones, and bond forces (but not torsions or constraints).[6]

Here, we present the GPU implementation of a complete, modern implicit solvent model for all-atom protein simulation in traditional force fields, with a very high performance compared to a single CPU core. We discuss some aspects of GPU architecture that impact the design of the code, and describe important optimizations needed to obtain good performance. We then evaluate it, and show that it can sometimes provide over 700 times the speed of highly optimized CPU based implementations.

Challenges of Porting to a GPU

While GPUs offer tremendous computing power, this comes at the cost of reduced flexibility. GPUs are different from CPUs in several fundamental ways that impact how they can be programmed. It is important to understand these differences to obtain good performance.

Scaling

CPUs typically provide a small number of very fast processing units, whereas GPUs have a large number of slower processing units. For example, the current high end Intel Xeon CPU has four processor cores,[7] while the current high end ATI GPU (Radeon 4870) has 800 math units.[8] That number is likely to

Correspondence to: P. Eastman; e-mail: peastman@stanford.edu

Contract/grant sponsor: Simbios via the NIH Roadmap for Medical Research; contract/grant numbers: U54 GM072970, NIH R01-GM062868, NSF EF-0623664.

continue increasing with future generations. Algorithms used for MD are traditionally evaluated based on how they scale with the number of atoms being simulated, but scaling considerations are only meaningful when the number of atoms is large compared to the number of math units. GPUs have already reached a point where, for small or medium sized proteins, the number of math units may be comparable to the number of atoms. On such a processor, the total amount of computation to be done may be much less important than how fully the available processing resources can be utilized.

Memory Access

One feature that CPUs and GPUs have in common is that memory access is much slower than computation, although GPUs have much faster memory systems, some having greater than 100 GB/second. Loading a value from memory can stall the processor for tens or even hundreds of clock cycles. CPUs deal with this by including a large amount of very fast cache memory. This allows programs to access memory in random order, and as long as the amount of data in use at one time is not too large, execution is fast.

In contrast, GPUs have only a very small amount of special purpose cache memory and hide latency with massive multithreading. Programs cannot rely on caches to hide latencies from random memory access. Instead, it is absolutely essential to group related data together and access it in contiguous blocks. In many cases, it is more efficient to repeat a calculation than to store the result in memory and reload it later.

Communication Between CPU and GPU

Data access from the math units to GPU local memory is slow compared to computation, but transferring data between the GPU and CPU across the PCIe bus is much slower still. For this reason, communication between the GPU and CPU should be kept to an absolute minimum. Ideally, the simulation should be executed entirely on the GPU, and results should be sent back to the CPU only infrequently for analysis and reporting. This requires that all parts of the computation be implemented on the GPU, including force evaluations, constraints, integration, etc. Otherwise, it will be necessary to transfer coordinates and/or forces between the CPU and GPU as part of each time step, which will reduce performance.

To test the effect of data transfers between the CPU and GPU, we modified the molecular dynamics code described in the next section to transfer all of the atomic coordinates from GPU to CPU and back again at each time step. No extra computation was done with these values; they were simply transferred and then discarded. Simulating the 576-atom villin headpiece with implicit solvent on an ATI Radeon 4870 GPU, this change caused the overall performance to decrease by 20%.

This applies not just to transferring data, but also to issuing commands. With any device controlled by a CPU but not physically part of it, there is a time delay between issuing a command and the beginning of its execution. In the case of a GPU, this latency can be greater than the time required to actually perform the calculation. For example, on an ATI Radeon 4870 GPU running under Windows XP, the latency of executing a kernel is $\sim$25 μs with the AMD Stream SDK 1.0.[9] Given this processor's maximum theoretical performance of over one teraflop,[8] 25 million floating point instructions could have been executed in that time. To combat this, it is necessary to do as much computation as possible in each function, combining what would logically be several function calls into one whenever possible.

Flow Control

Another important feature of GPUs is that the processors are not independent of each other. Threads are arranged in groups (typically between 16 and 64 threads on current GPUs), and all the threads in a group must execute exactly the same instruction at the same time (i.e. SIMD execution). This requires branching and other types of flow control to be used with care, since there is a large performance penalty if multiple branches need to be followed by the threads in a single group. When branch divergence occurs within a thread group, all threads will pay the cost of executing both sides of the branch. Whenever possible, it is best to avoid branching altogether. When it is absolutely necessary, it should be done in a spatially coherent way so that adjacent threads will usually follow the same branch.

Development Tools

Aside from the technical differences between CPUs and GPUs, the development tools available for GPU programming are much less mature than those for CPU programming. Our initial efforts on AMD boards were marked by considerable problems with buggy compilers, errors introduced by the requirement that array indices be floating point, limitations on the number of registers available in a kernel and the iteration count available in do-loops, to name several of the early ongoing issues. AMD and NVIDIA have expended considerable effort in fixing these types of problems, and as a result porting code to the GPUs is now much less painful. A major improvement on the AMD side was the introduction of the CAL[9] framework which allowed us to discontinue the use of the Microsoft DirectX 9 compiler which was the source of many problems. Nonetheless, problems caused by immature development tools are still frequently encountered.

In summary, realizing the full potential of the GPU still requires considerable effort in reworking the data structures and code to take advantage of the particular GPU architecture, and not all algorithms are amenable to these types of architecture. The AMD and NVIDIA implementations described here are different in many ways, reflecting the different strategies required to optimize the performance for the different boards.

Implementation

ATI Implementation Details

The ATI implementation is based on the Brook stream programming language.[10] Brook extends the C programming language to allow general-purpose programming on a GPU. The primary Brook constructs are streams and kernels. Streams are collections of relatively homogenous data that can be operated on independently in parallel. Kernels are the subroutines running on the GPU which carry out the parallel operations on the streams. In the current context, an example stream would be the atom coordinates which are input to a kernel that computes the bonded forces on each atom.

For the generations of ATI boards that were available while the software was under development, scatter capability (i.e. indirect writes such as $a[i] = x$) was not available. To circumvent this limitation, the different force and SHAKE calculations are carried out using two kernels:

1. A computational kernel to calculate the force or modified coordinates in the case of SHAKE and output the results to the frame buffer.
2. For each computational kernel a corresponding helper kernel to gather and sum the results stored in the frame buffer and update the final atom-indexed force array with the sum or updated coordinates in the case of SHAKE.

Bonded and Nonbonded 1–4 Interactions

The four bonded interactions (harmonic, angle, proper, and Ryckaert-Bellemans dihedrals) and the 1–4 nonbonded Lennard-Jones and Coulomb interactions are calculated by a single kernel. The inputs to the kernel are streams containing the indices of four covalently bonded atoms and the bond parameters (force constants, ideal bond lengths and angles, ...) needed to calculate all possible bond and nonbonded 1–4 interactions among the four atoms. The kernel output is the sum of these six forces for each of the four atoms. For example, if the atom indices are (i, j, k, l), then the harmonic bond forces between atoms (i, j), (j, k), (k, l), angle bond force (i, j, k) and (j, k, l), both proper and Ryckaert-Bellemans dihedral forces, and the 1–4 nonbonded force between (i, l) are computed. Many of the atom sub groups will occur in more than one input group. For example, the input atom indices (i, j, k, l) and (i, j, m, n) both implicitly include the harmonic bond (i, j) interaction. To prevent the harmonic bond interaction between atoms (i, j) from being included more than once, the input force constant for the duplicate entry is set to zero. A similar approach is followed for the other force subtypes to insure that each valid interaction is included exactly once and that invalid interactions do not contribute to the output force.

The strategy of computing forces that do not ultimately contribute to the final output force superficially seems to increase the computational time. However, the advantages of this approach are two-fold:

1. The number of helper gather kernels required is reduced from 15 to 4,
2. The number of memory accesses is significantly reduced.

The computational effort used in the helper kernels is minimal. However the overhead associated with the launch of each kernel is significant relative to the overall computational time for the relatively small systems considered here and in aggregate the overhead required is a significant fraction of the total time. For larger systems or for boards with scatter capability, this advantage may diminish in importance depending on the performance of scatter.

The number of memory accesses is minimized since if the coordinates for atoms i, j, k are retrieved to calculate the angle bond force, then under the merged approach adopted here the coordinates can be 'reused' for the calculations of the (i, j) and (j, k) harmonic bond forces. In general memory accesses are expensive relative to computations in terms of clock cycles required. For the 576-atom villin headpiece, the total number of bonded interactions was 4197, the merged list had 1770 entries. Of the 1770 entries, 552 of the Ryckaert-Bellemans interactions, 1652 proper dihedral, 2485 angle, 5016 bonded, and 267 1–4 force calculations were not used (i.e., force contribution was set to zero). However, the speedup over an earlier unmerged collection of kernels for each of the bond and 1–4 nonbonded interactions was a factor of 6.2 for villin.

Nonbonded Interactions

The nonbonded interactions include the 6–12 Lennard-Jones and Coulomb potentials. The inputs to the computational kernel are the two Lennard-Jones parameters, the atomic charges, and an exclusion matrix of size $N^2/4$, where N is the number of atoms in the system. The output is the computed nonbonded forces. The entries in the exclusion matrix are encoded such that the (i, j) element of the matrix signals whether atoms with indices $4j$, $4j+1$, $4j+2$, $4j+3$ are to be excluded from interacting with atom i; this reduces the number of memory access and storage required. The choice of four j-entries per entry was made based on the degree of loop unrolling of the inner loop (see below). The encoding is accomplished by setting the initial value of each entry to $210 = 2 \times 3 \times 5 \times 7$. If atom $4j$ is to be excluded, the entry is divided by two, if atom $4j+1$ is to be excluded the entry is divided by 3,... The kernel then decodes the exclusion entry by testing whether the modulus of the exclusion matrix entry divided by (2, 3, 5, 7) for the four atoms $(4j-4j+3)$ respectively, is nonzero. For instance if atom $4j+2$ is to be excluded from interacting with atom i, but the other three atoms $(4j, 4j+1, 4j+3)$ are to be included, then the exclusion matrix entry at (i, j) is 42. Since 42 is divisible by 2, 3, 7, but not 5, the interaction of atom i and atom $4j+2$ is excluded, while the other three interactions are included. The exclusion matrix can be reduced to size $O(N)$ at the expense of making the kernel slightly more complicated. However, empirical tests showed this reduction does not lead to a significant lowering of the computational time required for the proteins of sizes being simulated here.

For the current ATI implementation, the computation time for the nonbonded terms scales as N^2 due to the absence of a scatter capability. In contrast, the Nvidia implementation makes use of scatter capability and the computation time scales as $N^2/2$. The nonbonded computational kernel (as opposed to helper kernel to gather the total force on each atom) unrolls both the inner and outer loops over the atoms by four. One objective of unrolling is to take advantage of streaming SIMD extension (SSE)-like capabilities available on some graphic boards. For example in calculating $1/r$, the code

$$\text{float4}\ r2, \text{inverse\_r};$$
$$\ldots$$
$$\text{inverse } r = \text{rsqrt}(r2);$$

is used, where $r2$ contains the four squared distances between atom i and atoms j, $j+1$, $j+2$, $j+3$. The function rsqrt( ) computes $1/\sqrt{r}$ for all four distances using SSE-like instructions to reduce the number of processor cycles needed. Unrolling also lowers the number of memory accesses since atom coordinates and parameters are reused: the coordinates and parameter associated with atom i only need to fetched from memory once to calculate the force between it and atoms j, $j+1$, $j+2$, $j+3$ and not four times as would be required for more naïve implementations.

Another optimization technique employed was to partition the inner loop over the atoms into roughly equal-sized blocks that are processed independently. Hence if the number of blocks is specified as p, then for a fixed outer loop index i (in practice i, $i+1$, $i+2$, $i+3$ since the outer loop is unrolled by four), p GPU threads are used to process the inner loop. The first thread would handle atoms with indices $j = 1, q$, the second thread would handle atoms with indices $j = q+1, q+2, \ldots 2q, \ldots$ and the pth thread would handle atoms with indices $j = q \times (p - 1) + 1, N$, where $q = \text{int}(N - 1/p) + 1$. The advantage of this approach is that it allows the thread scheduler on the GPU board to allocate resources more efficiently. While one block of work is waiting for coordinates and parameters to be retrieved from memory, another block whose data is available can be processed. The disadvantage to this approach is that more gathers are required to sum the total force on each atom: for p blocks, p gathers will be required to sum the contribution from each block. For the current generation of boards, a value of p equal to four was found to be optimal. For larger values of p, while the computation kernel was faster, the speed up was offset by the increased time to do the gathers, resulting in negative or little net gain.

OBC Implicit Solvent

The implicit solvent model used here is based on the OBC Type II model.[11] The electrostatic part of the solvation free energy is given by the equation

$$\Delta G_{\text{pol}} = -\tfrac{1}{2} \sum q_i q_j / f^{\text{GB}}(r_{ij}, R_{\text{I}}, R_{\text{j}})(1 - 1/\varepsilon_w)$$

Here q_i is the charge on atom i, R_i is the Born radius for atom i, r_{ij} is the distance between atoms i and j, and ε_w is the solvent dielectric. $f^{\text{GB}}(r_{ij}, R_I, R_j)$ is taken to have the functional form:

$$f^{\text{GB}}(r_{ij}, R_I, R_j) = [r_{ij}^2 + R_i R_j \exp(-r_{ij}^2/4R_I R_j)]^{1/2}$$

Since the Born radii are a function of the conformation of the biomolecule and hence the r_i, the force $d\Delta G_{\text{pol}}/dx_{ik}$ is given by the equation

$$d\Delta G_{\text{pol}}/dx_{ik} = (\partial \Delta G_{\text{pol}}/\partial r_{ij})(\partial r_{ij}/\partial x_{ik}) + \Sigma_m^N (\partial \Delta G_{\text{pol}}/\partial R_m) (\partial R_m/\partial r_{ij})(\partial r_{ij}/\partial x_{ik})$$

The calculation of the OBC force is implemented with three N^2 loops. The first loop calculates the Born radii based on the current biomolecule conformation. The next loop computes $(\partial \Delta G_{\text{pol}}/\partial r_{ij})$ and accumulates the $\partial \Delta G_{\text{pol}}/\partial R_m$ for each atom i. Using the term $\partial \Delta G_{\text{pol}}/\partial R_j$ calculated in the second loop, the third loop computes the second term of the force and adds it to the term $\partial \Delta G_{\text{pol}}/\partial r_{ij}$ to get the implicit solvent force on each atom.

The optimization strategies employed for the nonbonded interactions discussed above were also applied to both of the implicit solvent loops with the same degree of unrolling and partitioning of the inner loop. One approach that was tried but was unsuccessful in reducing the computational time was to merge the second loop of the implicit solvent calculation with the nonbonded calculations. The obvious advantage of this approach is that only two sets of N^2 loops instead of three would be needed. This approach would also remove one set of gathers required to sum the forces. However, the required time actually increased. Further analysis showed that the number of cache misses increased significantly and the register requirements were much higher.

Another approach attempted was to merge the calculation of the Born radii with the third loop, thereby removing the need for the initial N^2 loop. The drawback to this strategy is that the Born radii used in the two remaining loops are then based on the conformation at the previous timestep. Given that the Born radii change slowly relative to the femtosecond timestep employed, the approximation appeared to be worth the reduction in computational effort. However energy conservation studies using a Verlet integrator showed the approximation led to unacceptable drift in the energy, and as a result, this approach was abandoned.

SHAKE Algorithm and Stochastic Dynamics

Stochastic dynamics was applied to the system as outlined by van Gunsteren and Berendsen[12] The bond lengths between hydrogen atoms and their heavy atom partners were constrained to their ideal value using the SHAKE algorithm.[13] The stochastic dynamics algorithm with constraints is implemented in four major steps (see Performance section of Ref. 12): an integration step, followed by an application of SHAKE, followed by another integration step and a final application of SHAKE. Each of the steps is implemented as a O(N) loop and no effort was made to optimize these steps since the contribution of this portion of the simulation to the total simulation time is small.

One small change to naïve implementations of stochastic dynamics and then SHAKE algorithm was to not compute the new coordinates until the last major stochastic dynamics step; instead the change between the new and old coordinates is passed between the method calls (Δx), as opposed to intermediate coordinate values (x). This reduces the number of additions and subtractions and hence minimizes the error introduced through rounding. We observed a significant improvement in the degree of energy conservation when this small change was included with runs using the velocity Verlet algorithm.

The Gaussian-distributed random numbers used in the stochastic dynamics algorithm are generated using the KISS[14] and Box and Mueller[15] algorithms. The KISS algorithm is a combination of three simple random number generators that yields a uniform distribution of values on the interval [0, 1]; it has been shown to pass a number of stringent tests.[16] The Box-Mueller algorithm takes the output values from KISS and transforms them into a Gaussian distribution. A large set of random values ($\sim 10^6$) is generated on the CPU using these algorithms and written to the GPU; the random values cannot be generated efficiently on the GPU due to a lack of integer arithmetic on the available ATI boards. For even relatively small proteins, the set of random values is quickly consumed. To generate a new set of random values on the CPU and copy them to the GPU each time a set is exhausted slows the program significantly. Hence the following strategy was adopted: once the set of random values is consumed, the values are recycled after being randomly permuted. After 100 such shuffles, a new set of random values is generated on the CPU and copied to the GPU, overwriting the previous set. In practice for a protein of 544 atoms, a new set was generated every $\sim$100,000 timesteps and for a 1254 atom protein every $\sim$40,000 timesteps.

NVIDIA-Specific Implementation Details

The NVIDIA implementation of the kernel was first meant to be a rough port of the existing ATI code into CUDA, a C-like language present on all NVIDIA GPUs from the 8xxx series onward. In general, the CUDA implementation followed the ATI implementation outlined above. However, it immediately became clear that exploiting architectural features of CUDA allowed for significantly more efficient execution, with differences from the ATI implementation as detailed below.

Bonded and Nonbonded 1–4 Interactions

Relatively little effort was put into the CUDA implementation of bonded and nonbonded 1–4 interactions because it only consumes about 1/6th of the total execution time and because CUDA allows one to spread the calculation over many independent threads. The only caveat here is that the interactions were padded such that no thread divergence based on the type of bonded interaction would occur within a warp since the execution of such divergence is cumulative. An attempt was made to use the texture unit to accelerated reading atomic information from GMEM. While this slightly accelerated G8x/G9x kernels, it slightly decelerated GT2xx kernels so it was discarded.

Nonbonded Interactions

Due to the existence of scatter, thread synchronization, and a 16 K of high-speed shared memory in each processor within CUDA-compatible GPUs, each nonbond kernel can exploit the symmetry of the force calculation matrix to calculate f_{ij}, then reverse its sign to generate f_{ji}. This reduces the magnitude of the overall calculation by a factor of $\sim$2 while incurring a small amount of overhead to coordinate this calculation. To do so, the kernel operates on the unique set of p x p tiles of the force matrix that are either above or along the diagonal, where p is the warp width (see Fig. 1). For each of these tiles, there is a corresponding swath of p atoms along the x and along the y axis.[17]

Each tile is then operated on by warps of p threads within a larger thread block, up to eight warps on G8x/G9x and up to 10 warps on GT2xx. For each of these tiles, there is a corresponding swath of p atoms along the x and y axis. To calculate the force data for such a tile, p threads then read one atom's worth of data from the x swath into their register space and then the corresponding set of atomic data for the tile's y swath into the shared memory. Furthermore, because all the threads in a warp are guaranteed to execute synchronously, each thread can interact with one atom's data in shared memory at a time for p iterations without any fear of overlap or any need for overt synchronization.

Additionally, as mentioned above, there were 3 $O(n^2)$ nonbond kernels in the ATI implementation. However, unlike the ATI client, merging the nonbond kernel with the first loop of the implicit solvent kernel was a big win, improving performance by 20%. The difference here lies in the ability of the shared memory to hold a sufficient number of intermediate values to make this a net win.

SHAKE Algorithm and Stochastic Dynamics

Essentially the same update and SHAKE algorithm as implemented on the ATI client was implemented for the CUDA client. One noteworthy difference between the two codes was that the Gaussian-distributed random values used in the update algorithm are generated on the GPU since integer arithmetic is available on the Nvidia boards; this is in contrast to the ATI client where the values are generated on the CPU and transferred to the GPU. As a result, the random values were not recycled using permutations as is done on the ATI client. One observation particular to the Nvidia client is that optimal performance for this section of the code was achieved when the workload was spread evenly across all SMs even when those SMs ran very small thread blocks ($<$30 threads).

Performance

Speed

To assess the improvement in speed from using GPUs rather than CPUs, we ran a series of benchmark calculations on several different protein systems, utilizing the ATI and Nvidia GPU codes. For comparison, the same calculations were run on one core of a 2 $\times$ 2.66 GHz Dual-Core Intel Xeon (running Mac OS X) with the AMBER9 program[18] built with commercial Intel compilers. We ran simulations of the D14A variant of the lambda repressor monomer (1254 atoms)[19,20] in a fully extended conformation, the N68H mutant of the villin headpiece subdomain (582 atoms)[21] in a folded conformation, the Fip35 WW domain (544 atoms)[22,23] in both folded and extended conformations, and one chain of the α-spectrin subunits R15, R16, and R17 from chicken brain (5078 atoms).[24]

The solvent was an Onufriev-Bashford-Case (OBC) generalized Born model[11] for both minimization and molecular dynamics. The parm03 and parm96 force fields (as noted in Table 1) were used for the CPU calculations, but in the case of villin, where both force fields were used, the difference in CPU per-

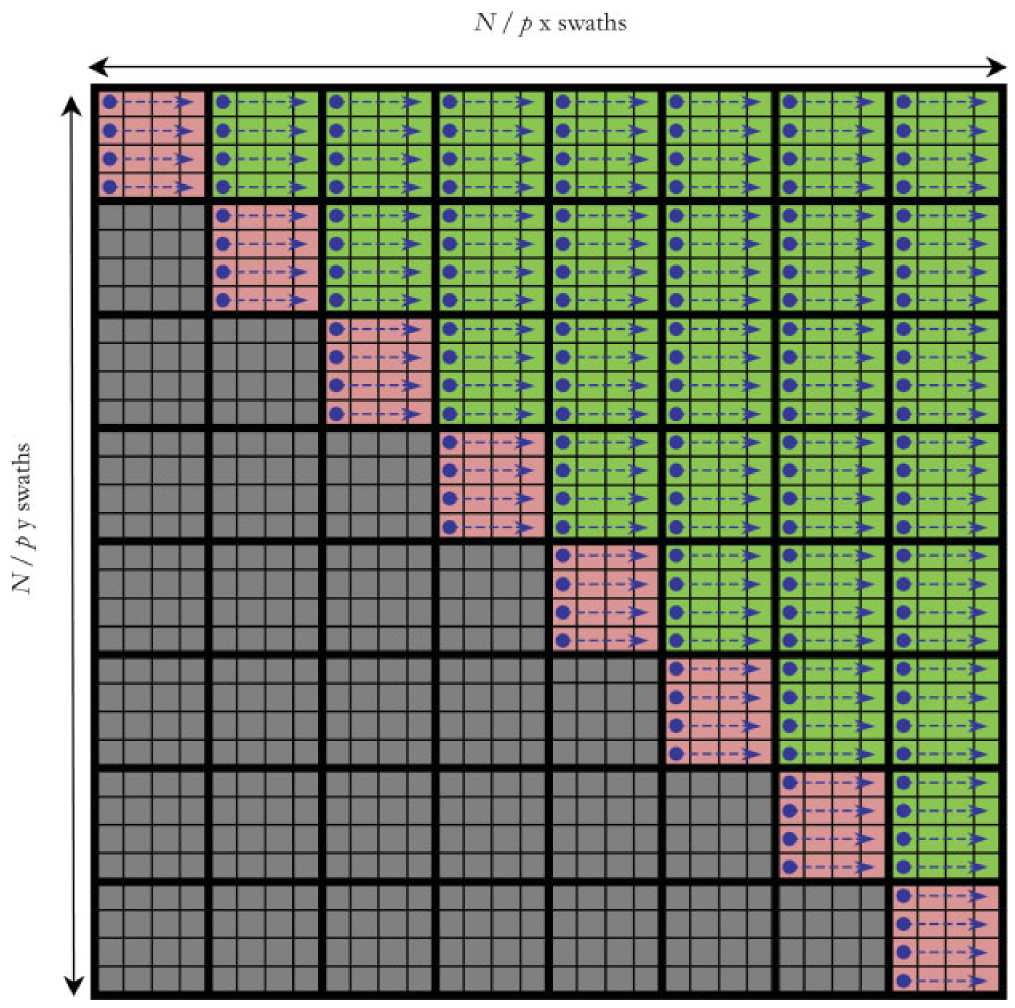

Figure 1. The set of *pxp* tiles required for force calculation-only the pink and green tiles need to be calculated. Force data for the grey tiles can be generated by negating the sign of the forces calculated for the corresponding green tile on the other side of the pink diagonal.

formance was negligible. After a standard energy minimization step, 40,000 Langevin dynamics iterations were run for each of the benchmark systems using a nonbonded 'cutoff' of 999 Å and no periodic boundary conditions to ensure an O(N^2) calculation. The Langevin collision frequency was 91/ps and the bath temperature was 300 K. Bonds involving hydrogen were constrained with SHAKE with default settings.

As expected for an N^2 calculation, there was no important speed difference on the GPU or the CPU when comparing folded and extended conformations of the same system. To exhaustion, the variable determining the simulation performance was the system size. The times are given in Table 1. These benchmarks show that running on the ATI GPU can result in more than 60× speedup over the CPU for small systems of ~600 atoms. These smaller systems gain more than 100× speedup on the Nvidia GPU. Both GPUs are capable of more than two orders of magnitude speedup compared to the CPU when calculating an MD trajectory for the larger systems, such as lambda repressor (~1200 atoms).

Table 1. Benchmark Results.

Molecule	Atoms	Force field	Platform	Ns/day	Improvement
fip35	544	parm03	CPU/AMBER	4.5	–
fip35	544	parm03	ATI	279.2	62
fip35	544	parm03	Nvidia	576.2	128
Villin	582	parm03	CPU/AMBER	3.9	–
Villin	582	parm03	ATI	260.8	67
Villin	582	parm03	Nvidia	528.5	136
Lambda	1254	parm03	CPU/AMBER	0.79	–
Lambda	1254	parm03	ATI	141.7	179
Lambda	1254	parm03	Nvidia	201.6	255
α-spectrin	5078	parm99	CPU/AMBER	0.023	–
α-spectrin	5078	parm99	ATI	14.2	617
α-spectrin	5078	parm99	Nvidia	16.9	735

Improvement is the speedup obtained by running on the GPU versus running AMBER on the CPU.
ATI: Radeon HD 4870 GPU.
Nvidia: GeForce GTX 280 GPU.

Table 2. Computational Performance.

Molecule	Atoms	Platform	GFLOPS
fip35	544	ATI	88 (176)
fip35	544	Nvidia	83 (148)
Villin	582	ATI	95 (188)
Villin	582	Nvidia	87 (155)
Lambda	1254	ATI	239 (475)
Lambda	1254	Nvidia	154 (275)
α-spectrin	5078	ATI	392 (780)
α-spectrin	5078	Nvidia	212 (378)

See Appendix A for details of how the numbers were calculated. The value in parenthesis is based on the cost estimates that are more appropriate for CPUs than GPUs.
ATI: Radeon HD 4870 GPU.
Nvidia: GeForce GTX 280 GPU.

The scaling of performance with number of atoms is quite different for the GPU codes than for the CPU calculations. On the CPU, the scaling is very close to O(N^2), indicating that nonbonded interactions are dominating the calculation time. In contrast, when going from 544 atoms to 1254 (a factor of 2.3), the ATI code only slows down by a factor of 2.0. The scaling in this regime is actually sublinear. This is not surprising, given that the number of atoms in the smaller system (544) is less than the number of math units in the GPU (800), and it is difficult to generate enough parallel computation to use all math units and hide latencies. Even when going from 1254 atoms to 5078 (a factor of 4), both GPUs still scale subquadratically (a factor of 10.0 for ATI and 11.9 for NVidia). This suggests that the speedup of GPUs over CPUs could be even greater for still larger systems.

Another important performance metric is the overall processor utilization. By seeing how close we come to the peak theoretical performance of the GPU, we can see how efficiently our implementation makes use of the large computational resources available. The results are shown in Table 2. Note that we use two different methods for calculating GFLOPS which yield different numbers. See Appendix A for details. The performance increases with increasing system size, but even for α-spectrin we are only reaching a fraction of the processor's peak theoretical performance. This suggests that further optimization might significantly improve the performance, especially for small proteins, by better exploiting the resources of the GPU.

In practice, molecular dynamics simulations on a CPU are not run as described above. Rather, cutoffs are frequently applied for both long range interactions and Born radii calculations. These approximations reduce the complexity of the calculation from O(N^2) to O(N log N) which can be a significant performance gain even for small systems. However, Born radii calculations are formally O(N^2), and to our knowledge no GB formalism exists for which cutoffs are explicitly taken into account. Thus, the effects of applying a cutoff to GB are unknown so their use could be extremely dangerous. On the other hand, application of long-ranged cutoffs in nonperiodic systems is less risky, but still involves an approximation with an unknown effect on accuracy. As such, the benchmarks presented here represent the performance gain of running MD calculations on the GPU over running the most accurate CPU calculation possible for a given force field and GB model, rather than a performance gain over typical simulation conditions. However, it is clear from the performance gain that there is no longer a need to run CPU calculations which apply cutoffs to long-ranged interactions and to Born radii calculations. Instead, more accurate simulations can now be run using GPUs which do not involve approximations to the chosen model.

Even taking the above into account, the speed improvements listed in Table 1 should not be taken as precise measures of the intrinsic speed advantage of GPUs over CPUs. The AMBER benchmarks were run on a single CPU core, but most desktop CPUs today have two or four cores. Also, it is possible that its performance could be improved by further tuning. AMBER is a mature and widely used package, so we expect that significant work has been done to optimize it, but it is always possible that further work could yield additional performance gains. Our goal in presenting CPU benchmarks is simply to give a point of reference against which the performance of the GPU code may be approximately measured.

Accuracy

Another important consideration in evaluating any dynamics code is the accuracy of the trajectories it produces. Both implementations described here used single precision floating point numbers throughout, since double precision has only very recently become available on GPUs and still carries a large performance penalty. Previous work has shown that single precision is sufficient to produce high quality results in molecular dynamics,[5] but only if care is taken to do calculations in a way that avoids unnecessary loss of accuracy. Also, some floating point operations on GPUs are not IEEE compliant. Any error resulting from this should be very small, but given the already limited precision being used, it is potentially a cause for concern.

To test the accuracy of our GPU codes, we incorporated the velocity Verlet algorithm into them, ran a series of simulations of lambda repressor, and measured how accurately energy was conserved. The results are shown in Table 3. Simulations were performed both with and without bond length constraints. When constraints were used, the accuracy of energy conservation was found to depend strongly on the convergence tolerance used for SHAKE, so results are shown for three different tolerance values.

The results compare favorably to those for other molecular dynamics codes, including ones which use double precision.[5]

Table 3. Energy Drift per Degree of Freedom (kT/ns/dof).

Constraints	Nvidia	ATI
None	0.0054	0.0178
H-bonds (SHAKE tolerance 10^{-4})	0.0611	0.1031
H-bonds (SHAKE tolerance 10^{-5})	0.0220	0.0541
H-bonds (SHAKE tolerance 10^{-6})	0.0060	0.0558

All simulations were 1 ns in length and used a time step of 1 fs.
ATI: Radeon HD 4870 GPU.
Nvidia: GeForce GTX 280 GPU.

This gives us confidence that lack of precision is not harming the quality of our simulations. We do note that the accuracy is somewhat lower for the ATI implementation than for the Nvidia implementation. We are still investigating to determine why this is true.

Future Work

A complete system for simulating molecular dynamics on GPUs has been presented. In the interest of creating such a complete system in a timely manner, some molecular dynamics scenarios have not been fully explored. It has not escaped the authors' notice that there remain several areas for which the molecular dynamics capabilities of this GPU implementation might be extended and enhanced. What follows are descriptions of a few such areas for potential enhancement.

The use of an implicit solvent model represents a tactical choice in the implementation of the current work. Two advantages of this choice include the avoidance of boundary condition issues, and minimization of the number of explicit atoms being simulated. On the other hand, explicit solvent simulations can, in principle, be performed on GPUs. The most important enhancement required to perform accurate simulations in explicit solvent is the implementation of periodic boundary conditions. Further, the calculation of long range forces would need to be enhanced to take such boundary conditions into account. Established techniques for such long-range force computations can also improve the asymptotic complexity of the algorithms, compared to the current implementation (see below).

The current work has been restricted to relatively small macromolecules (roughly 500–5000 atoms). Efficient simulation of significantly larger systems may require algorithms with lower time complexity than those used in this work. Our GPU implementations use some force computation algorithms that scale quadratically [$O(n^2)$] with respect to the number of atoms in the simulation. This approach is justified in the case of small proteins and smaller molecules, as the simplicity of the algorithms permits saturation of the parallel processing units. For larger molecule simulations, it may be more efficient to also provide linear time [$O(n)$] and/or log-linear [$O(n \log(n))$] algorithms for the force calculations. Such methods typically involve separating nonbonded interactions into short-range and long-range components. The long-range components are computed using Fourier or hierarchical methods,[25] while the short-range components are computed using cell linked list methods.[26] Such algorithms with lower time complexity will be especially important for simulations in explicit solvent, which contain many times the number of atoms found in implicit solvent simulations. The Fourier methods, in particular, are well suited to handling periodic boundary conditions, thus facilitating explicit solvent simulations as well.

Because our GPU implementations have been developed over a period of time, some of the latest advances in GPU hardware have not been fully exploited. For instance, recent ATI hardware permits "scatter" operations, which involve writing to different memory locations within a kernel. Our ATI implementation has avoided scatter operations because they were not available on earlier generation hardware. It might be possible to achieve greater computation efficiency by reengineering certain methods to take advantage of scatter operations. Another recent advance in GPU computing is support for double-precision floating point computations on ATI and NVIDIA GPUs. Double precision arithmetic still carries a significant performance penalty relative to single-precision arithmetic. It may be worth investigating situations in which the increased accuracy of higher precision arithmetic might be worth the additional computational cost.

Because we find ourselves in the somewhat special circumstance of creating many implementations of the same algorithms to support different hardware, we are especially sensitive to the importance of effective software testing environments. We would like to extend our test cases to be able to apply tests of correctness to essentially all molecular dynamics simulation programs.

More generally, there are undoubtedly many additional methods that could be implemented on GPUs to extend the range of available simulation scenarios. For example, more sophisticated force fields, such as polarizable force fields,[27] could benefit from GPU acceleration. The precise details of how best to implement these methods remain to be worked out. We hope that our current work will help form the basis for an ever increasing library of GPU accelerated molecular dynamics techniques.

Availability

The implementation reported in this manuscript will be made available at Simtk.org as part of the OpenMM API (http://simtk.org/home/openmm). OpenMM is designed for incorporation into molecular dynamics codes to enable execution on GPUs and other high performance architectures.

Appendix A

The performance numbers in Table 2 were determined by inspecting the code to count the exact number of floating point operations in each time step. There is no universally accepted way of counting operations, however. Addition, subtraction, multiplication, and division each count as a single operation, but we also must assign operation counts to transcendental functions such as logarithm and exponential. These operations are expensive to perform on a CPU, so they are traditionally assigned large operation counts. GPUs, in contrast, have specialized circuitry which allows them to be calculated very quickly.

This creates an ambiguity about how to calculate performance. One option is to assign operation counts that accurately reflect how quickly a GPU can perform each operation: if an exponential takes no more time to calculate than an addition, it should be counted as a single operation. Alternatively, one could assign operation counts that reflect how expensive an operation is on a CPU. If a typical CPU requires 20 clock cycles to calculate an exponential, the GPU should be given credit for doing 20 floating point operations, even though it does it much more quickly.

Table 4. Operation Counts for Transcendental Functions.

Function	GPU operation count	CPU operation count
Sqrt	1	15
Reciprocal Sqrt	2	16
Log	1	20
Exp	1	20

We therefore chose to use two different methods for calculating performance based on the two sets of operation counts shown in Table 4. The first set reflects how quickly a GPU can calculate various functions, and is most appropriate when comparing multiple simulations run on a GPU. The other set of values reflects how quickly a typical CPU can calculate those functions, and is most appropriate for comparing the performance of a GPU to a CPU.

References

1. Elsen, E.; Houston, M.; Vishal, V.; Darve, E.; Hanrahan, P.; Pande, V. S. SC06 Proceedings 2006.
2. Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. J Comput Chem 2007, 28, 2618.
3. Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; Purcell, T. J Comput Graphics Forum 2007, 26, 80.
4. Ensign, D. L.; Kasson, P. M.; Pande, V. S. J Mol Biol 2007, 374, 806.
5. Hess, B.; Kutzner, C.; van der Spoel, D.; Lindahl, E. J Chem Theory Comput 2008, 4, 435.
6. Anderson, J. A.; Lorenz, C. D.; Travesset, A. J Comput Phys 2008, 227, 5342.
7. Intel. Xeon Processor Specifications. Available at: http://www.intel.com/products/processor/xeon5000/specifications.htm?iid=products_xeon5000+tab_specs 2008.
8. AMD. ATI Radeon HD 4800 Series Home Page. Available at: http://ati.amd.com/products/Radeonhd4800/index.html 2008.
9. AMD. AMD Stream Computing. Available at: http://ati.amd.com/technology/streamcomputing/index.html 2008.
10. Buck, I.; Foley, T.; Horn, D.; Sugerman, J.; Fatahalian, K.; Houston, M.; Hanrahan, P. ACM Trans Graphics 2004, 23, 777.
11. Onufriev, A.; Bashford, D.; Case, D. A. Proteins 2004, 55, 383.
12. van Gunsteren, W. F.; Berendsen, H. J. C. Mol Simul 1988, 1, 173.
13. Ryckaert, J.-P.; Ciccotti, G.; Berendsen, H. J. C. J Comput Phys 1977, 23, 327.
14. Marsaglia, G. Random numbers for C: The END? 1999.
15. Box, G. E. P.; Muller, M. E. Ann Math Stat 1958, 29, 610.
16. L'Ecuyer, P.; Simard, R. ACM Trans Math Software 2007, 33, 22:1.
17. Nyland, L.; Harris, M.; Prins, J. In GPU Gems 3; Nguyen, H., Ed.; 2008, pp. 677–695.
18. Case, D. A.; Darden, T. A.; Cheatham, III, T. E.; Simmerling, C. L.; Wang, J.; Duke, R. E.; Luo, R.; Merz, K. M.; Pearlman, D. A.; Crowley, M.; Walker, R. C.; Zhang, W.; Wang, B.; Hayik, S.; Roitberg, A.; Seabra, G.; Wong, K. F.; Paesani, F.; Wu, X.; Brozell, S.; Tsui, V.; Gohlke, H.; Yang, L.; Tan, C.; Mongan, J.; Hornak, V.; Cui, G.; Beroza, P.; Mathews, D. H.; Schafmeister, C.; Ross, W. S.; Kollman, P. A. 2006. AMBER 9, University of California, San Francisco.
19. Yang, W. Y.; Gruebele, M. Biochem 2004, 43, 13018.
20. Yang, W. Y.; Gruebele, M. Biophys J 2004, 87, 596.
21. Kubelka, J.; Eaton, W. A.; Hofrichter, J. J Mol Biol 2003, 329, 625.
22. Freddolino, P. L.; Liu, F.; Gruebele, M.; Schulten, K. Biophys J 2008, 94, L75.
23. Liu, F.; Du, D. G.; Fuller, A. A.; Davoren, J. E.; Wipf, P.; Kelly, J. W.; Gruebele, M. Proc Natl Acad Sci USA 2008, 105, 2369.
24. Kusunoki, H.; Minasov, G.; Macdonald, R. I.; Mondragón, A. J Mol Biol 2004, 344, 495.
25. Sagui, C.; Darden, T. A. Ann Rev Biophys Biomol Struct 1999, 28, 155.
26. Allen, M. P.; Tildesley, D. J. Computer Simulation of Liquids; Clarendon Press: Oxford, 1987.
27. Ponder, J. W.; Case, D. A. Adv Protein Chem 2003, 66, 27.