JCTC Journal of Chemical Theory and Computation

Simulation of Coarse-Grained Protein−Protein Interactions with Graphics Processing Units

Ian Tunbridge,[†] Robert B. Best,[‡] James Gain,[†] and Michelle M. Kuttel*,[†]

Department of Computer Science, University of Cape Town, Cape Town, South Africa and Department of Chemistry, Cambridge University, Cambridge, United Kingdom

Received July 12, 2010

Abstract: We report a hybrid parallel central and graphics processing units (CPU−GPU) implementation of a coarse-grained model for replica exchange Monte Carlo (REMC) simulations of protein assemblies. We describe the design, optimization, validation, and benchmarking of our algorithms, particularly the parallelization strategy, which is specific to the requirements of GPU hardware. Performance evaluation of our hybrid implementation shows scaled speedup as compared to a single-core CPU; reference simulations of small 100 residue proteins have a modest speedup of 4, while large simulations with thousands of residues are up to 1400 times faster. Importantly, the combination of coarse-grained models with highly parallel GPU hardware vastly increases the length- and time-scales accessible for protein simulation, making it possible to simulate much larger systems of interacting proteins than have previously been attempted. As a first step toward the simulation of the assembly of an entire viral capsid, we have demonstrated that the chosen coarse-grained model, together with REMC sampling, is capable of identifying the correctly bound structure, for a pair of fragments from the human hepatitis B virus capsid. Our parallel solution can easily be generalized to other interaction functions and other types of macromolecules and has implications for the parallelization of similar N-body problems that require random access lookups.

1. Introduction

The formation of multiprotein complexes, either transient or permanent, is integral to many biological processes. Some examples are antibody−antigen and protease−inhibitor complexes, protein complexes involved in cellular signal transduction processes, structural proteins that maintain the shape of a biological cell, and the very large multiprotein complexes represented by the proteasome, the nuclear pore complex and viral capsids. Identification of the site and strength of interaction (binding) between proteins involved in common cellular functions is integral to a comprehension of how they work cooperatively. This can improve our grasp of disease mechanisms and provide the basis for new therapeutic approaches. Consequently, the prediction of protein binding sites has been identified as one of the 10 most sought-after solutions in protein bioinformatics.[1] This problem is closely related to the well-known NP-hard "protein folding problem" of predicting the three-dimensional structure of a protein from its primary sequence.

In the absence of sufficient experimental data on the atomic structure of protein complexes, molecular dynamics (MD) or Monte Carlo (MC) simulations of protein complex components can assist in determining both their mode of interaction and the location of the interaction site(s).[2] Molecular simulations generate an ensemble of configurations, from which both structural and thermodynamic data can be extracted. The configurations representing bound protein complexes enable identification of both the docking sites and the relative orientation of the proteins, while the binding affinity of a complex can be estimated from the proportion of bound samples occurring in the ensemble. However, all-atom simulations of multiprotein complexes are highly computationally expensive and are therefore limited

* Corresponding author. E-mail: mkuttel@cs.uct.ac.za. Telephone: +27 (0)21 6505107.

[†] University of Cape Town.

[‡] Cambridge University.

10.1021/ct1003884 © 2010 American Chemical Society
Published on Web 10/05/2010

in scale by the available computing resources. Simulations are typically restricted to simple biological systems (e.g., small binary protein complexes without solvent) and to nanosecond time scales. Accurate coarse-grained models have helped to extend molecular simulations to more biologically relevant length- and time-scales.[3−11] These reduced molecular models aggregate single atoms into large spherical beads to significantly decrease the computational cost of a simulation. There is considerable potential for further accelerating molecular simulations by combining coarse-grained models with the computational power of massively parallel graphics processing units (GPUs).

Modern GPUs have floating-point computational capabilities far in excess of current central processing units (CPUs). Essentially, GPUs fall into the category of single program multiple data (SPMD) compute devices; organizing data into homogeneous streams of elements and executing a function, or *kernel*, on all elements of a stream simultaneously. Current high-end GPUs also have high memory bandwidth compared to CPUs. For example, the nVIDIA GTX280 has 240 fragment or stream processors and a theoretical memory bandwidth of 141 GB/s. As a consequence, these compact devices are capable of rapid high-throughput numeric operations and can be employed effectively by nongraphical memory-bound algorithms of high arithmetic intensity, such as the N-body problem inherent in molecular simulations. For coarse-grained potentials, evaluation of the total interaction potential between all beads, N, in a protein molecule is an $O(N^2)$ operation and the chief performance bottleneck, a common feature of N-body simulations in general. The independence of each pairwise interaction means that the calculation of all such potentials suits the SPMD GPU architecture, promising good speedups over CPU-only implementations.

The difficult task of porting algorithms to the GPU architecture, while maintaining effective use of the CPU, has been made easier with the development of general application programming interfaces. In 2007, nVIDIA released the compute unified device architecture (CUDA) API, which allows the general programmer direct access to the nVIDIA GPU hardware. CUDA allows for operations not supported by graphics APIs, such as local data communication between kernels and scatter and gather operations. However, CUDA GPU programming is not trivial. Programmers must be mindful of the GPU memory hierarchy, which requires explicit management to minimize access latency and effective packing of data to enable a coalesced memory access pattern.[12] In addition, maximizing GPU performance often requires latency hiding through exploitation of the multithreading capabilities of the CPU cores,[13] adding the difficulties of conventional multithreaded asynchronous programming to the GPU-specific programming techniques.

However, despite these difficulties, there are increasing reports of successful CUDA implementations of N-body algorithms achieving good speedups over CPU implementations.[14−16] Specifically, GPU-based calculations of the expensive long-range electrostatics and other nonbonded forces necessary for molecular mechanics simulations are typically 10−100 times faster than heavily optimized CPU-based implementations.[17−19] Friedrichs et al. showed speedups over a single CPU implementation of up to 700 times for large all-atom protein MD running entirely on the GPU.[16] However, such great speedups are not always achievable. A recent implementation of an acceleration engine for the solvent−solvent interaction evaluation of MD simulations shows speed-ups of up to a factor of 54 for the solvent−solvent interaction component but only 6−9 for the simulations as a whole.[20]

Here we report our hybrid CPU−GPU parallel implementation of a coarse-grained model and an energy function for simulation of multiprotein complexes recently developed by Kim and Hummer[11] together with a replica exchange Monte Carlo (REMC) simulation protocol to enhance sampling. We leave the original model and simulation methods unchanged; our primary focus is on development of a general, highly scalable parallel implementation, with the ultimate goal of increasing the size of tractable simulations to the point where far more biologically relevant simulations can be attempted.

Previous implementations of related N-body dynamics on a GPU, such as the GRAPE implementation,[15] translate the potential evaluation into a convenient map-reduce problem.[14−16] However, in our case this approach is not feasible, as the Kim−Hummer coarse-graining model requires very frequent random-access lookups in evaluation of the interaction potential, and the memory resource limitations of the GPU prevent the use of standard optimal GPU memory-access models for this problem.[21,22] Therefore, in order to optimize the parallel performance of our implementation, we assessed the performance impact of storing the structural data and the potential lookup table in the various different types of memory available on a GPU to establish the optimal configuration. We validate our final CPU−GPU implementation against the original, demonstrating that our simulations reproduce the reference results of Kim and Hummer,[11] while showing a factor of 4 speedup for small systems. Further benchmarking analysis demonstrates that our implementation of this model achieves excellent speedups of over 1400 for large simulations. We demonstrate that the potential is capable of identifying the correct bound structure for a pair of protein fragments from the human hepatitis B (HBV) virus capsid, although there are also minor populations of incorrectly bound structures. This paves the way for larger scale simulation studies of capsid assembly mechanisms.

The parallelization approach developed in this work is generally applicable to N-body problems that require similar random access lookups. This often occurs where the aspects of the interaction between bodies are dependent on their type or state. One instance is the commonly used energy functions in all-atom MD simulations, in which the interactions depend on the type of each atom.[23]

2. Methods

In this section, we present details of the coarse-grained protein−protein simulation method and some high-level considerations for its parallelization, validation, profiling, and benchmarking.

2.1. The Kim−Hummer Coarse-Grained Model. In common with many current coarse-grained models, the

Kim−Hummer model (hereafter referred to as the model) represents a protein molecule as a chain of beads corresponding to specific amino acid residue types. Coarse-grain representations are generated from a protein's atomic structure by centering a bead on the C_α atom in each amino acid residue, with the radius of the bead being the van der Waals radius of the specific residue.

In the simplest version of the model, proteins are treated as rigid bodies, with the interaction potential, U_{tot}, comprising only the pairwise sum of short-range amino-acid-dependent Lennard-Jones potentials, $u_{ij}(r)$, and long-range electrostatic Debye−Hückel interactions, $u_{ij}^{el}(r)$:

$$U_{tot} = \sum_{ij} f_i f_j (u_{ij}(r) + u_{ij}^{el}(r)) \quad (1)$$

where r is the distance between residues i and j, and $0 \le f_i \le 1$ is the weighting factor for residue i. The weighting factor scales the contribution of a particular residue, allowing residues on the surface of the molecule to contribute more than residues buried within the protein.[24] In the simplest case, all interactions are weighted equally ($f_i = 1$ for all i). Since we use rigid bodies to represent proteins, these weights remain constant for each residue throughout a simulation.

Based on known contact potentials, ε_{ij}, between residues, short-range interactions are either attractive ($\varepsilon_{ij} < 0$) or repulsive ($\varepsilon_{ij} > 0$) and are defined as

$$u_{ij}(r) = \begin{cases} 4|\varepsilon_{ij}|[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^{6}], & (\varepsilon_{ij} < 0) \\ 4\varepsilon_{ij}[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^{6}] + 2\varepsilon_{ij}, & (\varepsilon_{ij} > 0, r < r_{ij}^0) \\ -4\varepsilon_{ij}[(\sigma_{ij}/r)^{12} - (\sigma_{ij}/r)^{6}], & (\varepsilon_{ij} > 0, r \ge r_{ij}^0) \end{cases} \quad (2)$$

where σ_{ij} is the distance between residues i and j, determined from the average of the respective van der Waals diameters of each residue,[11] and $r_{ij}^0 = 2^{1/6}\sigma_{ij}$ is the lowest energy separation.

The long-range electrostatic potential between residues is defined as

$$u_{ij}^{el} = \frac{q_i q_j \exp\left(\frac{-r}{\xi}\right)}{4\pi D r} \quad (3)$$

where q_i is the charge of residue i, D is the dielectric constant of the solvent, and ξ is the Debye screening length. Kim and Hummer use $D = 80$ as the dielectric constant of water and $\xi \simeq 10$ Å, corresponding to physiological salt concentrations.

Note that our implementation currently does not allow for the flexible linker peptides connecting rigid-protein domains in the original model. However, these can be included relatively cheaply, with the contribution of the appropriate stretching, bending, and torsion-angle potentials for the flexible linkers calculated in $O(N)$ time complexity on the CPU.

2.2. Methods for REMC Simulations. In our implementation, we use the REMC) protocol reported for the original model.[11] MC methods employ a sequence of random geometric mutations (moves) of the system (usually rotations or translations) to explore and sample configurational space in a desired ensemble. In our case, the Metropolis MC algorithm[25,26] is most suitable. In Metropolis sampling, the energy of the system is evaluated after each move, and mutations are accepted or rejected in such a way that configurations are sampled with probabilities given by the Boltzmann distribution (i.e., the probability of configuration X, $P(X) \propto \exp(-E(X)/k_B T)$, with k_B being the Boltzmann constant).

In each MC mutation, a randomly selected protein is either translated (by 0.5 Å) or rotated (by 0.2 radians about a protein's centroid) along a random axis within a periodic bounding box. Varying the volume of the box alters the concentration at which the simulation is performed. For these simulations, a value of less than $2k_B T = -1.186$ kcal/mol for the interaction energy is used to determine that the proteins are in a bound state,[2] although the results are not very sensitive to this value.[11] To determine the binding affinity of a complex, the fraction bound y (the proportion of bound samples out of the total number of samples) is determined as a function of the protein concentration $[A] = 1/V$, and the dissociation constant K_d is evaluated from the relation:[11]

$$y = \frac{[A]}{[A] + K_d} \quad (4)$$

The sampling properties of MC simulation are further improved by using replica exchange, in which concurrent simulations are run at different temperatures.[27] Such a REMC search maintains χ independent replicas of the system, with each replica run at a different temperature value (T_1, T_2, ..., T_χ). During replica exchange, configurations are swapped between replicas at neighboring temperature values with a probability proportional to their energy and temperature differences. Specifically for a pair of replicas at temperatures T_1 and T_2, with current coordinates X_1 and X_2, respectively, configurations are swapped using the criterion $\Delta = \exp[(\beta_1 - \beta_2)(U(X_1) - U(X_2))]$, with $\beta_1 = 1/k_B T_1$, $\beta_2 = 1/k_B T_2$, and $U(X)$ the potential energy. Analogous to standard Metropolis MC, moves are always accepted if $\Delta > 1$, otherwise they are accepted if $\rho < \Delta$, with ρ random numbers on the interval [0,1). This process runs until sufficient samples have been generated for the calculated fraction bound metric, y, to converge, typically requiring between 10^8 and 10^9 MC mutations per replica.

2.3. General Parallelization Approach. In our approach to parallelization of the Kim−Hummer model, we aim to exploit the various levels of concurrency inherent in the REMC algorithm, from fine-grained parallelism in the costly potential evaluation to relatively coarse-grained parallelism in the multiple replicas.

As we only had access to an implementation of the model within the CHARMM package,[28] we began with development of a serial implementation with full simulation functionality to provide a reference point from which to perform validation, profiling, and benchmarking of the subsequent parallel GPU implementation. We note that our CPU implementation shows slightly faster run times than the CHARMM version.

At a high level, a parallel version of the $O(R)$ REMC algorithm (where R is the number of replicas) is relatively

straightforward to implement, and the many concurrent independent replicas make this algorithm highly suitable for parallel execution across many CPU cores. From a multi-threading perspective, replicas encapsulate parallel work units; each replica performs a self-contained MC simulation. The Markov chain nature of these simulations means that an individual simulation cannot be split across threads, as each step of the MC simulation is dependent on the previous step. We therefore employ a multiple producer−consumer model, with R replica MC simulation threads and one replica exchange thread. Control is initially passed to the MC threads, with the replica exchange thread waiting until the MC threads complete their allotted iterations. At this point, control is passed to the replica exchange thread, and replica exchange is performed. This process is repeated for the specified number of MC steps. This high-level approach scales well to multiple CPU cores, with or without a GPU.

At a lower level, the $O(N^2)$ potential evaluation is by far the most costly operation and is therefore exported to GPU. Although it is considered best to associate thread contexts one-to-one with GPU runtime contexts,[22] our model also allows for sharing of one or more GPU's between multiple threads. Such concurrent potential evaluation by multiple replica threads can be handled in two ways: either by sharing the GPU between threads though multiple contexts or by using CUDA steams to perform asynchronous computation on the GPU. Asynchronous calls maximize resource utilization of both the GPU and CPU but require explicit management of the streaming process, whereas the use of pthreads to run concurrent MC simulations implicitly manages processing overlap between the GPU and CPU. For asynchronous GPU calls, each replica is assigned to a stream, and the application is configured to overlap MC mutations and acceptance/rejection sampling with interaction potential calculations. This allows for maximum resource utilization in the application, ensuring that the GPU is busy at all times, thereby maximizing the simulation throughput. For purposes of comparison, we implemented both approaches.

2.4. Simulations. *2.4.1. Benchmarking.* For benchmarking purposes, 20 replicas were simulated for 1000 MC steps each, for system sizes ranging from 100 to 7668 residues. (Note that a typical simulation of 20 replicas will realistically run for 10^7 steps per replica, 10 000 times longer than this benchmark.) The two smallest benchmark cases are the UIM/Ub (100 residues) and Cc/CcP (402 residues) systems also used for validation. For the larger systems, we simulate the interactions of viral capsid proteins from the human hepatitis B virus (HBV, PDB ID 2G33), where each protein dimer comprises 284 residues. We simulate systems consisting of 2 (568 residues) up to 27 (7668 residues) dimers. This system was chosen with the eventual application of virus assembly in mind, although for benchmarking, we are only concerned with the performance aspects.

All benchmarking simulations were run using an nVIDIA GTX280 (Asus ENGTX280) running at a stock clock speed of 600 MHz paired with a Intel Core 2 Duo 3 GHz E8400 CPU and 4GB of DDR2−800 MHz RAM.

2.4.2. Validation. Our implementation was validated in two stages. First, the single point interaction energies produced by both our single CPU and the GPU implementations were validated against those from the original model implementation for 10 reference structures (Youngchan Kim, personal communication). We note that the serial CHARMM implementation gives identical results to the original model. Second, to verify the MC simulation implementation, we reproduce the binding affinity for simulations of two protein complexes originally used in the development of the model,[11] namely: (i) the binding of ubiquitin to the UIM1 domain of the Vps27 protein (abbreviated to UIM/Ub, PDB ID 1Q0W) [29]and (ii) the binding of yeast cytochrome c to cytochrome c peroxidase (Cc/CcP, PDB ID 2PCC),[30] both at 300K. We reproduce the simulation protocol of the original study, using replica exchange with 20 replicas at various temperatures between 250 and 600 K, exchanged every 1000 MC steps. Calculation of the equilibrium binding affinity of each complex allows for direct comparison with the original results. Simulations were run for 10^7 steps per replica, with a total of 2×10^8 simulation steps for each of 6 concentrations between 100 and 1000 μM. The GROMACS utility g_cluster[31] was used to cluster bound configurations via the linkage algorithm using root-mean-squared deviations (rmsd) as a metric, with a cutoff of 0.1 nm.

2.4.3. Initial Application. As a first step toward the simulation of the assembly of an entire viral capsid, we simulate the interaction of two identical fragments from the human HBV capsid. Each unit comprises 4 chains, or a total of 582 residues, with the structure taken from PDB ID 2G34 (substructure of one fragment). Sampling is done by REMC with 10 replicas at temperatures between 300 and 416 K (temperatures follow a geometric progression), exchanged every 1000 MC steps. Average acceptance ratios are between 16 and 97% for the MC mutations. Simulations were run for 10 million steps per replica for a total of 100 million simulations steps. This produced ~40 000 bound instances at 300 K which were clustered according to rmsd, as described above.

3. Software Implementation

Here, we present specifics of the algorithm implementation: the object-based decomposition, the data bundling and transfer strategies, and the kernel implementation.

All code was written in C and C++ using the GNU Scientific Library, the C++ posix thread library, the C++ Standard Template Library and Linux 64-bit operating system. Versions 2.0−2.3 of the CUDA toolkit were used for development.

Our application accepts Protein Data Bank (PDB) format files of atomic protein structures. These files are parsed, and coarse-grained representations of each protein generated.

3.1. Multithreaded REMC Simulations. Simulations are initialized with the number of MC steps, the intervals at which replica exchange is performed, and the sampling interval. MC mutations were implemented with the aid of the GSL Mersenne twister pseudorandom number generator.[32] We ensure that the mutation operations preserve the integrity of molecular structures by performing only rigid-body rotations or translations. Uniform random selections of both molecule and mutation type are performed, followed

either by translation by 0.5 Å or by rotation about a molecules centroid by 0.2 radians, along a random axis. Translations are implemented trivially as a vector translation in 3D with single precision, but rotations, given the accumulation of error in simulations of such length, require full double precision. Since the relative error in the displacement of residues from their intended position is approximately 10^{-4} after 1 million rotations, the error accumulated from single-precision rotation would quickly render the data from simulations unusable. Rotations are thus performed by generating a quaternion representation along the axis of rotation and applying it to the residues in the selected protein. Residue positions are cast to double precision for the rotation operation and cast back to single precision afterward.

Our implementation of replica exchange is extremely lightweight, with a negligible contribution to simulation time. Each replica contains counters for various metrics, such as the fraction bound and the MC acceptance/rejection ratio, which are in turn written at the sampling interval to simulation output files. Because we perform rigid-body docking, structural information is stored as the rotation and displacement of each molecule relative to its initial state.

3.2. Data Representation. We employed a hierarchy of classes to encapsulate the simulation. At the top of the hierarchy, a replica object encapsulates the data specific to a replica instance, such as pointers to the specific arrays in GPU memory, where the replica residue data is stored for the duration of a simulation. The MC simulations are encapsulated within each replica. To reduce transfer times, after any mutation, only the altered molecule is updated on the GPU before invocation of the GPU kernel.

Each residue bead is represented as a data tuple comprising the position (relative and absolute) of each residue, the residue type, the van der Waals radius, and the electrostatic charge. Proteins consist of a contiguous array of residues, which ensures that as many residues as possible occur in the same cache line, consequently improving the rate at which residues are loaded into cache memory on the CPU for mutation.

The short-range contact potential, ε_{ij}, values for the 210 unique residue interaction pairs are stored in a lookup table. Both the residue array and this look-up table are transferred once to the GPU, where they persist for the duration of the simulation. The residue data is stored in the GPU global memory in arrays of type float4 and float3 to ensure coalesced reads when kernels access residue data.[21] The look-up table can be stored in several ways: as a noncached global array, a cached array using constant memory, or a texture mapped to global memory. We determined the best among these storage locations via careful profiling of the algorithm, as discussed above.

3.3. GPU Kernel Implementation. The kernel is the fragment of code executed by every GPU thread. Our kernel calculates the pairwise interaction potential (algorithm 1) which, using the implicit addressing model provided by the GPU, is applied to residues according to their position in global memory, as determined by thread and block indexes and by block dimension. Residue pairs are allocated across a grid of thread blocks, partitioning the work done by the GPU into tiles containing subsets of the pairwise interactions. We use a GPU-optimized parallel reduction[33] to compute a partial sum of the interaction potential for the tile assigned to each thread block. As the matrix is symmetric, only thread blocks above the major diagonal contribute to the interaction potential, and the partial sum is halved for thread blocks on the diagonal. Partial sums are written back to global memory on the CPU host, for addition to the total interaction potential, U_{tot} (eq 1). We found no benefit to performing this accumulation on the GPU for larger simulations, and this shift of work to the CPU speeds up the single GPU/multiple pthread configuration.

Algorithm 1 Residue Pairwise Interaction

$r \leftarrow eps + \| position_i - position_j \|$
$e_{ij} \leftarrow \lambda(LJ(type_i, type_j) - e_0)$
$\sigma_{ij} = \frac{1}{2}(radius_i + radius_j)$
$u_{ij} \leftarrow -4 \cdot e_{ij} \cdot (\sigma_{ij}/r)^6((\sigma_{ij}/r)^6 - 1)$
if $e_{ij} > 0$ and $r < \sqrt[6]{2} \cdot \sigma_{ij}$ **then**
  $u_{ij} \leftarrow -u_{ij} + 2e_{ij}$
end if
$u_{ij}^{\text{el}} \leftarrow charge_i \cdot charge_j \cdot e^{-r/\xi}/r$
$U_{ij} = u_{ij} + u_{ij}^{\text{el}}$

Our default implementation stores the data for residues assigned to a thread block in the shared memory for the associated symmetric multiprocessor (SM) in a coalesced manner, as in astronomical N-body CUDA implementations.[14,15] However, as we require more shared memory than previous implementations, we experience lower occupancy on each processor because all threads assigned to a processor jointly access its 16 KB of shared memory. An alternative is the use of texture or global memory for these residues.

Further, the short-range interaction potential requires random access retrieval from a lookup table of LJ(type$_i$, type$_j$), the van der Waals contact potential for a pair of residues. Unfortunately, this table consists of 210 distinct values, which may be required in any order, which does not fit the ideal data model for a GPU.[22] Texture memory tolerates random memory accesses better than other types of memory on the GPU,[21,22] providing us with an alternate mechanism for implementing the lookup table. However, the latency hiding effects of multiple threads and caching afforded by constant memory do not necessarily make this the best choice. The impact of the type of GPU memory used for both the residue data and the ε_{ij} (eq 2) contact potential lookup table on application performance is discussed below.

4. Performance Tuning

We followed a staged approach to tuning our application for optimal performance. First, kernel parameters were adjusted to ensure that the kernel execution time is as fast as possible. As the CUDA kernel is an indivisible unit of work, its optimization is independent of multithreading and

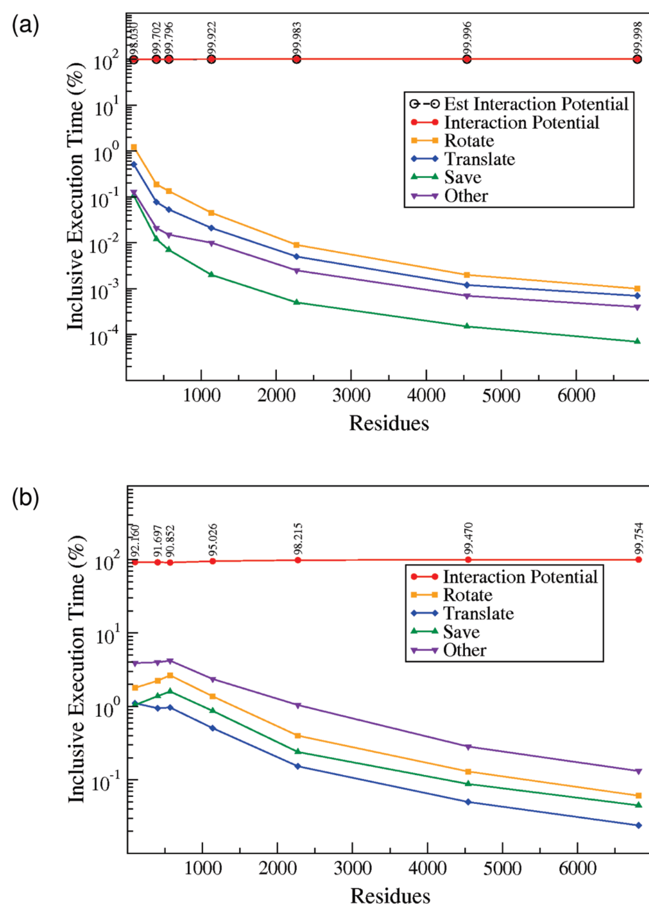

Figure 1. (a) Serial CPU and (b) parallel GPU profiles of the percentage execution time consumed by various operations (logarithmic scale). The interaction potential calculation accounts for over 97% of the simulation time, approaching almost 100% for large systems. Analytical estimates for the expected proportion of time devoted to interaction potential (Est.) are close to the observed values.

asynchronous calls. (Note, however, that the most recently released CUDA-based compute devices can launch different kernels concurrently.) Once an optimal kernel configuration was found, multithreading was used to divide work between CPU cores and overlap CPU and GPU computation. Streaming was undertaken as the last stage of application tuning.

As expected, profiles of a single-threaded CPU implementation for each of the benchmark cases (Figure 1a) demonstrate that the $O(N^2)$ evaluation of the interaction potential consumes the vast majority of simulation time; rotation, translation, and other operations in the MC simulation are relatively inexpensive in comparison. For this reason there is no real benefit in improving the runtime of these operations.

For the kernel optimization, the balance of random access lookup, occupancy, shared memory allocation and the block size provide a myriad of options. We found best performance in most cases for a kernel comprising a thread block size of 64 threads, texture memory for constant potential lookups, and a shared memory model mimicking the GRAPE hardware model.[15] We therefore used this configuration to profile our GPU implementation.

Exporting the interaction potential evaluation to the GPU leads to a massive reduction in execution time, in the range of one to three orders of magnitude improvement. Predictably, larger problem sizes benefit most, due to the greater degree of parallelism in their computation. However, interaction potential calculations still require 90%+ of a simulation's runtime (see Figure 1). Note that the GPU implementation requires that data be copied to GPU memory before and after kernel invocation, which accounts for the increased "other" operations profile on the GPU as compared to the CPU.

Instruction-level optimization yielded little improvement in our kernels due the dependence of the kernel on lookup table access. Reordering of the pairwise interaction formulas (eqs 2 and 3, as listed in algorithm 1) results in a few key optimizations. For one, $(\sigma_{ij}/r)^6$ can be calculated once and stored in a local variable, allowing the powf function to be called once, as opposed to six times in the original equation (eq 2). Algorithm 1 also minimizes the divergence in the calculation by ensuring the branching required to calculate u_{ij} only contains two multiplies and one add operation. Further, we use the intrinsic GPU functions expf and rsqrtf to increase instruction throughput. The drawback of this type of reordering is that more registers are required to store temporary values, but since it is shared memory rather than register memory that limits our occupancy, this does not negatively impact performance.

4.1. Kernel Tuning. The optimal kernel configuration for a GPU application is dependent on many factors, ranging from high (thread block size and the GPU grid size) to very low (use of specific memory types and instruction-level code optimization) levels.

We load residues within a kernel in the conventional N-body manner,[14] using an array of shared memory within each thread block to store the residues so that they can be accessed quickly. However, we require seven 32-bit values per residue, three more than required for gravitational N-body simulations, which only store position and mass in shared memory.[14,15] In addition, a patch of shared memory is required for the parallel sum at the end of each thread block to produce an overall interaction potential value for all of the electrostatic interactions of the block. In the gravitational case, each thread computes and stores the updated velocity and position of a single body resulting in a much simpler kernel.

This increased shared memory requirement proves to be the limiting factor on the kernel performance, as the occupancy of the SMs on the GPU is limited by the amount of shared memory available to each thread block. This is illustrated in Figure 2a. For this configuration, occupancy on the GPU is constrained to 37.5% in the cases of 64 or 128 threads per block and 25% occupancy in the cases of 32 or 256 threads per block. A thread block of 512 threads will not execute due to insufficient memory. For smaller problem sizes, where very few thread blocks are required, the amount of padding can have a negative impact on the simulation. For example, 32 threads per block results in the least padding and thread blocks to execute the kernel, making it fastest for small problems. Conversely, 256 threads per block requires the most padding and results in times much longer than the other configurations. Predictably, performance

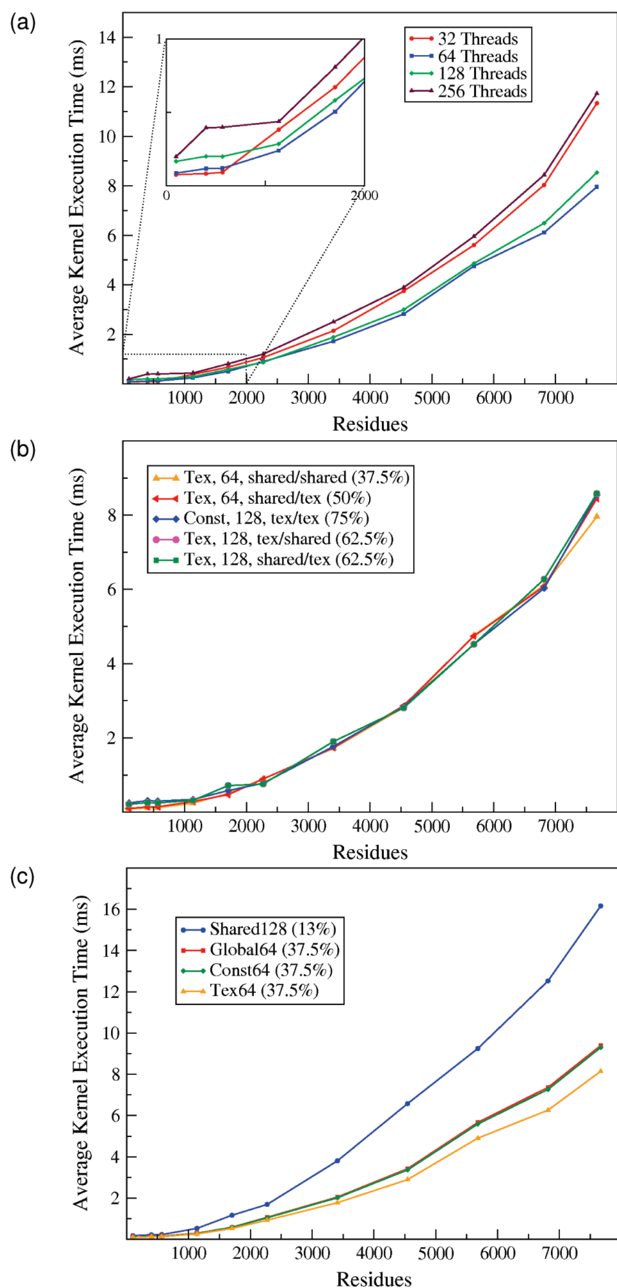

Figure 2. (a) Kernel execution time with varying number of threads per block, using texture memory for the contact potential lookups. Configurations of 32, 64, 128, and 256 threads per block execute in times indicative of their percentage occupancy. (b) Relative performance of alternate memory locations for storing the residue data. Each entry in the key lists the location for contact potential lookups storage, the number of threads per block, the storage location for the residue data (molecule identifier and position vs rest of data), with kernel occupancy values in brackets, e.g., Tex, 64, shared/shared refers to texture memory for contact potential lookups, 64 threads per block, and shared memory for all residue data (shared/shared). (c) Relative performance of various GPU memory locations (global, constant or shared) for storing the potential lookup table.

for larger residues is linked to occupancy in this configuration, as illustrated in Figure 2a. With occupancies of 37.5%, 64 and 128 threads per block perform best, outperforming 32 and 256 threads per block, which can only achieve 25% occupancy. Thread blocks of 64 threads run the fastest, as they strike the best balance between occupancy, warps per block and blocks per SM. This is because, ultimately, the random access of the contact potentials is a bottleneck on the GPU, and the fastest configuration is the one that best hides the cost of serializing this operation. For this reason, it is important to maximize kernel occupancy.

We experimented with the use of the different GPU memory types—texture, global, constant, or shared—for storing the contact potential lookup tables on the GPU. Figure 2b shows the best benchmark performance for each type of memory. As expected, the poorest performance is experienced using shared memory, which is already limited in availability. Using shared memory for the lookup table as well as the residue data results in an even lower occupancy −6% for 32 and 64 threads and 12.5% for 128 threads, with a proportional degradation of performance. However, if there were sufficient shared memory on each SM to fit many blocks, each containing a constant potential lookup table, then we would expect performance to improve due to increased occupancy and faster memory access, even though this does not accord with the prescribed memory access model.[22] The performance of constant and global memory is interesting. Since they perform almost identically, it is clear that the diversity of lookup values prevents the single value caching afforded by constant memory from having any impact. This is also the reason why texture performs best; because of the spacial caching feature of texture memory, any read from texture memory results in the entire table being cached on the GPU for 3 SMs. This results in faster lookup times for all thread blocks running on the SMs which belong to the texture unit. Both global and constant memory performance is generally 10−20% slower than its texture memory lookup equivalent because of this caching effect.

Unexpectedly, some alternative configurations perform almost as well as the best configuration (using texture memory for contact potential lookups and shared memory for prefetching residues for fast access in thread blocks, with 64 threads per block). As shown in Figure 2c, performance may be improved by increasing the occupancy, using texture instead of shared memory for the residue data, in part or entirely. For example, storing only the molecule identifier and position in shared memory and the rest of the residue data in texture memory results in a negligible drop in kernel performance. In this case, the higher access latency of texture memory is offset by the benefit of higher occupancy (50−75%). A possible advantage of this configuration is that large simulations (which we are aiming at) of up to 131 072 residues can be performed using a thread block size of 512 (previously prohibited because of shared memory constraints allowing a maximum simulation size of only 65 536 residues). In the case of constant memory, this strategy improves the performance of all configurations, but not to the extent that they are faster than the original configuration. As another example, using constant memory for lookups performs surprisingly well if all residue data is moved to texture memory (Figure 2b, red line). With a block size of 128 threads per block, this increases the SMs occupancy to 75%,

Table 1. Conformation Energies[a]

Conf.	CHARMM	CPU	GPU	η
1	−0.294085	−0.293705	−0.293705	2.03×10^{-7}
2	−1.056417	−1.056291	−1.056291	2.26×10^{-7}
3	−10.278304	−10.277435	−10.277431	4.64×10^{-7}
4	−7.584171	−7.580382	−7.580391	1.20×10^{-6}
5	−0.000079	−0.000079	−0.000079	3.67×10^{-7}
6	−5.564564	−5.562238	−5.562239	2.57×10^{-7}
7	−5.452568	−5.480216	−5.480217	2.61×10^{-7}
8	−10.670303	−10.711964	−10.711967	2.67×10^{-7}
9	−9.904111	−9.900359	−9.900359	0.00
10	−8.518124	−8.527744	−8.527749	5.59×10^{-7}

[a] Comparison of U_{tot} values (kcal/mol) for 10 reference conformations in the CHARMM implementation (equivalent to the original implementation)[11] and our serial CPU and GPU implementations. Relative errors (η) are on the order of 10^{-7}, with an average GPU error of 3.8×10^{-7}.

almost matching our fastest kernel and affording the best latency hiding of any configuration.

Generally we observe that block sizes of 32 threads per block perform best for simulations of fewer than 1000 residues, and for larger simulations a block size of 64 threads is optimal. As a result, we incorporated a simple autotuning feature to achieve better kernel occupancy, simply selecting a dynamic block size on-the-fly, based on prior kernel performance. We limit autotuning to block sizes of 32, 64, 128, 256, and 512 threads to satisfy the reduction algorithm.[33] We find that the relative differences between our kernels is minor, provided that a sensible kernel configuration is selected (Figure 2b and c).

5. Results and Discussion

5.1. Accuracy of the GPU Implementation. Of particular importance for a successful CUDA implementation of an algorithm is the accuracy of the mathematical functions employed by the GPU kernel. A limitation of the GTX280 architecture is that a double precision arithmetic is performed at one-eighth of the speed of a single precision (30 double vs 240 single precision units). Although the latest nVIDIA architectures have better double precision perfromance, single precision is still significantly faster. On the Fermi GF100, double precision is performed at half the rate of single precision with 256 FMAs per clock vs 512 single precision FMAs.[34] Therefore the use of single precision arithmetic is still required to achieve the best performance on GPU architectures.

However, care must be taken to ensure that the use of single precision arithmetic does not result in an unacceptable reduction in computational accuracy. We find a mean relative error of 0.00146 for interaction potential calculations in our GPU implementation and CHARMM, almost identical to the mean relative error between our CPU implementation and CHARMM (0.00143). The largest differences occur in the calculation of the van der Waals component of the interaction potential. The mean relative error between our double precision CPU and single precision GPU implementations is 3.8×10^{-7} (Table 1). In cases where the GPU values differ most from the CPU, use of the powf function in calculating $(\sigma_{ij}/r)^6$ proved to be the cause, due to its maximum ULP error of 8.[21] A benefit of our GPU kernel is

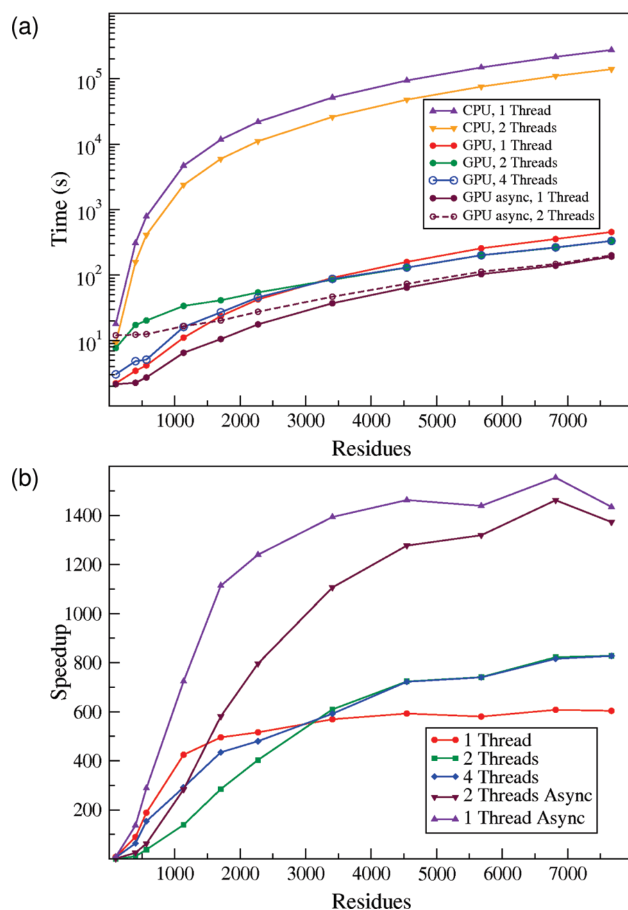

Figure 3. (a) Simulation times for various configurations of GPU and CPU. The GPU simulations generally outperform the CPU simulations by two orders of magnitude from as little as 500 residues. Asynchronous GPU usage proves to be fastest in all cases, provided the GPU is not shared between contexts. The overhead of context switching is evident in the constant difference of 10 s between the multithreaded and serial asynchronous benchmarks. (b) Speedup of the GPU implementation over our serial implementation. The use of CUDA streams results in the best speedup of our implementation, with the serial asynchronous GPU solution performing up to 1400 times faster than the serial CPU solution for larger benchmarks. A clear crossover point occurs for the synchronous multithreaded benchmarks at approximately 3000 residues; for larger simulations, the cost of context switches is amortized by the thread level parallelism of the simulation.

that pairwise summation is implemented implicitly in the reduction used to calculate the total interaction potential from each pairwise potential. This results in the error rate due to round-off errors growing at a low rate proportional to $O(\varepsilon\sqrt{\log n})$, as compared to $O(n)$ for a simple summation.

5.2. Performance of the GPU Implementation. We benchmarked simulation times for various options ranging from a thread on the CPU to multiple threads in combination with asynchronous calls to the GPU kernel. Multithreading across CPU cores results in performance improvement proportional to the number of cores (Figure 3 a) on a dual core machine, two threads perform twice as fast as a single thread, illustrating the high scalability of parallel MC simulations. The benefits of the GPU implementation are clear. Even a straightforward, single-threaded, single GPU

configuration shows a remarkable peak speedup of 600 times that of the serial CPU solution (Figure 3b, red line). For large problems (>3000 residues), sharing the GPU among multiple CPU threads provides an even better performance of up to 800 times that of the CPU (Figure 3b, green line). This is because multiple threads ensure that the GPU is better utilized, even though the card has to perform costly CUDA context switching between threads. It is clear that the benefits of using a GPU are only fully realized when simulating large enough problems. For example, our smallest benchmark shows a speedup factor of 8 for a single thread and actually performs 30% slower for two threads.

However, the best performance was obtained not with multithreading but with asynchronous calls to the GPU with CUDA streams. For streaming with one thread of execution, we found that any configuration of 2, 4, 5, or 10 streams performed equally well. Specifically, a configuration comprising a single thread with asynchronous GPU calls (Figure 3b, purple line) performs best across all benchmarks, over 1400 times for systems larger than 4000 residues. The relatively poor performance of multithreaded confirgurations is partly attributable to the cost of context switches between threads, a fact highlighted by the constant 10 s performance difference between the streaming and multithreaded benchmarks for all system sizes (Figure 3a). Indeed, for smaller simulations, where the relative penalty of performing a context switch between threads is greater, a single-thread configuration (Figure 3b, red and purple lines) always outperforms the equivalent multithreaded code. For multithreaded asynchronous configurations (Figure 3b, maroon line), the cost of context switching is largely hidden by the overlap in GPU and CPU computation. Remaining differences in performance are governed by the kernel efficiency of a particular implementation. For example, the "kinks" in the speedup curves are due the scheduling of kernels on the GPU, occurring for various block dimensions at differing simulation sizes. Although seemingly large in Figure 3b, the speedup graph merely exaggerates the effect of the variations in Figure 2b.

We conclude that the GPU is fully utilized if streams are used for relatively small problem sizes. However, if streams are unavailiable, a configuration with as many CPU threads as cores is likely to provide optimal performance. Macromolecular simulations performed using this system run successfully on a cluster of 1.0 T C870 (G80) cards, where asynchronous CUDA calls are not supported and each card only possesses 128 CUDA cores, with half the amount of shared memory per SM of the GTX280.

5.2.1. Comparison with other CUDA Implementations. The performance of our GPU kernel is consistent with that achieved by Friedrichs et al.[16] for similar molecular potential evaluations on the same GTX280 architecture (Figure 4). The potential evaluation components of our REMC MC simulations and their CUDA implementation of MD map similarly to the GPU. However, there are some key differences in the MC algorithm that makes it more computationally expensive and accounts for the better performance of the MD code. Our kernel is required to perform a $\log_2 n$ reduction of the n^2 pairwise potential to a single energy value

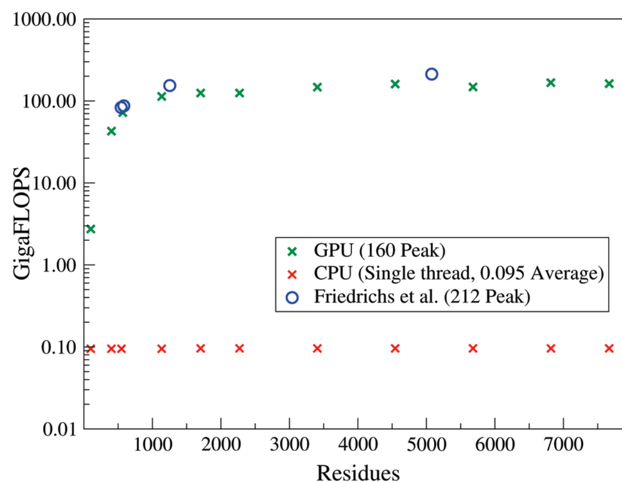

Figure 4. Calculation of the interaction potential using a heterogeneous GPU−CPU calculation achieves peak performance of 160 GFLOPS. A comparable MD simulation by Friedrichs et al. achieves peak performance of 212 GFLOPS. Branching and reduction operations required to generate our potential result in lower overall throughput than the MD simulations, where integration of each atom's velocity and position ports more amenably to the GPU architecture.

(which is used to either accept or reject the MC move). Reduction requires synchronization before each of its $\log_2 n$ iterations and performs one arithmetic operation to two loads and one store, which impacts negatively on the kernel performance relative to the MD code. Reduction operations are memory bound with low algorithmic intensity and therefore have relatively poor gigaFLOP performance on the GPU architecture.[33] Conversely, the MD kernel includes an $O(n)$ integration step after the pairwise accumulation, the higher the algorithmic intensity, separability of these force calculations and lack of synchronization is better suited to the GPU architecture and results in slightly better GFLOP performance for this kernel.

Indeed, it has been noted that, while MD simulation techniques can be fully implemented on the GPU with relative ease, this does not hold for MC methods.[35] Many particle MC simulations are difficult to parallelize, both on conventional parallel architectures and on SIMD hardware, because the random acceptance moves cause unpredictable branching and the requirement for global synchronization.[36] We do not implement the MC mutation in the kernel but leave it to the CPU, for reasons of higher double precision accuracy in the geometric transformations. This also enables concurrent execution on both GPU and CPU for multiple replicas, achieving higher overall performance for multiple replicas as opposed to single replica performance.

In addition, the LJ potential employed coarse-grained model (eq 2) is more complex that that used by Friedrichs et al. It is either attractive or repulsive and thus dependent not only on the distance between residues but also on the sign of the contact potential. This additional branching conditional makes the LJ computation more costly to compute on a GPU. We also do not employ a cut-off radius for nonbonded interactions, as is the case for the MD code (on the GPU, a branch in which no computation occurs is effectively free), our kernel always has to calculate the LJ

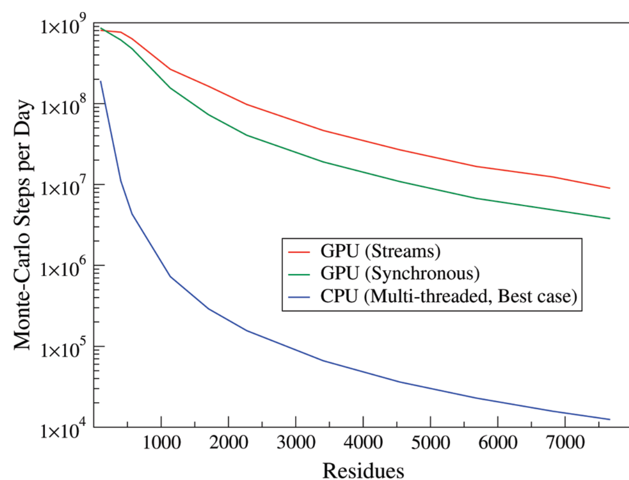

Figure 5. One a dual core machine with one GT280 card, our hybrid CPU−GPU implementation performs almost 1 billion MC iterations per day, outperforming the CPU by 2 orders of magnitude for as little as 500 residues.

force. These differences explain why, though our performance follows a similar profile to the MD code, we do not achieve their same maximum performance level (160 vs 212 GFLOPS, Figure 4). However, measures of FLOPS do not provide a reasonable metric by which to evaluate the real-world utility of an implementation. The overall performance of the simulation is critical, not merely the performance of the most parallel component. The comparatively lower performance of our potential evaluation kernel on the GPU is offset by streaming and multiple replicas in the REMC algorithm. We keep the branching and synchronization off the GPU as much as possible, sacrificing single stream performance for overall throughput. That this is an effective strategy for MC algorithms is clearly shown by the number of MC iterations per day achieved by our system (Figure 5). On a dual core machine with one GTX280 card, our GPU-accelerated code can perform almost 1 billion iterations per day for a simulations of 2 molecules totalling 100 residues. (In general, an unbound simulation requires on the order of 1 million to over 10 million iterations per replica before it reaches equilibrium, depending on the complexity and number of structures and on at least as many iterations more to perform sufficient sampling.) The largest simulations of 7668 residues are capable of up to 9 million iterations per day, whereas the CPU, fully utilizing both cores, manages only 12 000.

5.3. Validation and Initial Applications. To validate the correctness of the entire system (the energy function implementation, the MC algorithms for moving the proteins, and the replica exchange protocol), we have tested the code on two systems studied before: the binding of yeast cytochrome c to the cytochrome c peroxidase complex (Cc/CcP, Figure 6) and ubiquitin to the UIM1 domain of the Vps27 protein (UIM/Ub, Figure 7).[29] Binding curves from the present implementation closely match those from the previous CHARMM code; a clear validation of the correctness of our GPU implementation.

For the UIM/Ub system, we have simulated a “full-length” version of ubiquitin (residues 1−76) in addition to the truncated ubiquitin (residues 1−72 with flexible C-terminus

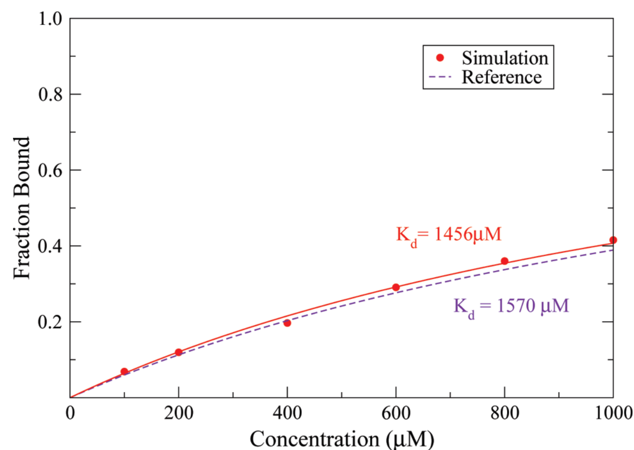

Figure 6. Binding affinity for Cc/CcP. Titration curves obtained with simulations using the GPU implementation are compared with the binding curve from the original reference implementation.[11]

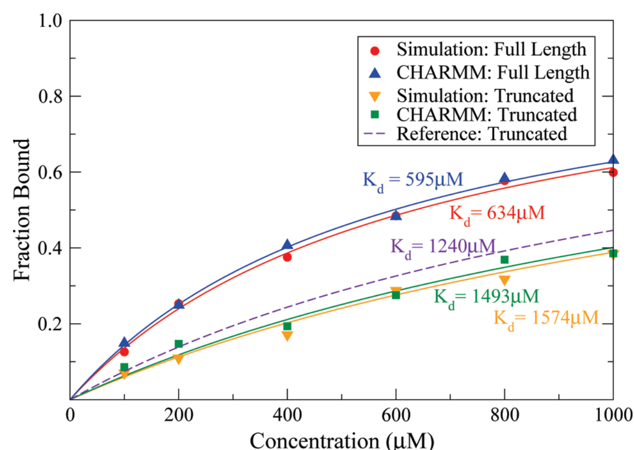

Figure 7. Binding affinity for UIM1/Ub. Binding data are given for a “full length” version of ubiquitin (residues 1−76) as well as the truncated ubiquitin (residues 1−72) in which the flexible C-terminus was removed, as in the original publication.[11] “Simulation” refers to the present implementation, “CHARMM” refers to the CHARMM implementation, and “Reference” refers to the original publication. The small discrepancy from the reference implementation is probably due to the use of spherical boundaries in that case, in contrast to the other implementations which use periodic boundaries.

removed) reported originally by Kim and Hummer.[11] The results for this system are interesting. The original study used a truncated form of ubiquitin because the C-terminus has a flexible tail. This resulted in a dominant population of bound complexes with native-like structures, with a population exceeding 40%. In addition, however, approximately 20% of the bound population comprised structures occuping the native binding pocket in the ubiquitin but with an inverted orientation of the UIM1 helix. In our study, we find a similar result. A cluster analysis of the bound conformations of the truncated ubiquitin simuilations with a 1 Å rmsd cut-off gives a 1.0:4.0 ratio of incorrect:correct helix orientations. For the full length version of ubiquitin, however, there is a slight increase in overall binding affinity (Figure 7), suggesting additional favorable interactions between the UIM1 and the flexible C-terminus of ubiquitin. Furthermore, we find that

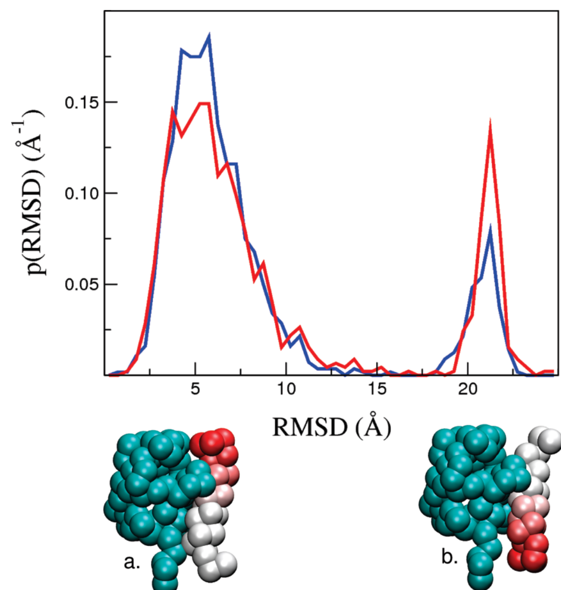

Figure 8. The distribution of rmsd to the correct binding site for the UIM1/Ub system in the full length version of ubiquitin (blue line) as well as the truncated ubiquitin (red line) in which the flexible C-terminus was removed, as in the original publication.[11] The small but significant population shift from the incorrectly (b) toward the correctly (a) bound conformation is clear for the full-length version of ubiquitin.

inclusion of the C-terminus slightly improves the specificity of binding. The ratio of incorrect:correct bound orientations improves to 1.0:6.2, indicating that the C-terminal interactions also preferentially stabilize the correctly bound conformation. Plots of the distribution of rmsd to the correct binding site for the both full length and the truncated ubiquitin make the shift in population clear (Figure 8). Since our study still somewhat unrealistically treats the C-terminus as rigid even when it is included, it will be interesting in the future to examine whether allowing flexibility in the tail further favors the correctly bound state.

One of our goals for the application of this code was to study the assembly mechanism of virus capsids from their constituent proteins. To test the feasibility of such a study, we have carried out trial simulations of a pair of protein fragments (each corresponding to the four protein chains from the PDB entry 2G33). The results for our trial simulations of two fragments from the HBV capsid are very encouraging. The bound conformations were initially clustered using the same method as for the UIM/Ubq complex; some of the clusters were then combined once the symmetry of the complex (i.e., AB:CD is equivalent to CD:AB) was accounted for, resulting in five clusters overall. In Figure 9 we have plotted the distance root-mean-square (DRMS) of structures from each cluster against their energy. We find that by far the largest cluster is that representing the correctly bound conformation, with a DRMS of around 4 Å from the experimental structure, comparable to that obtained for other complexes with this potential.[11] The correctly bound structures also have the lowest potential energy. A number of subsidiary, higher energy, bound clusters was also identified, indicative of mild frustration on the binding energy landscape.

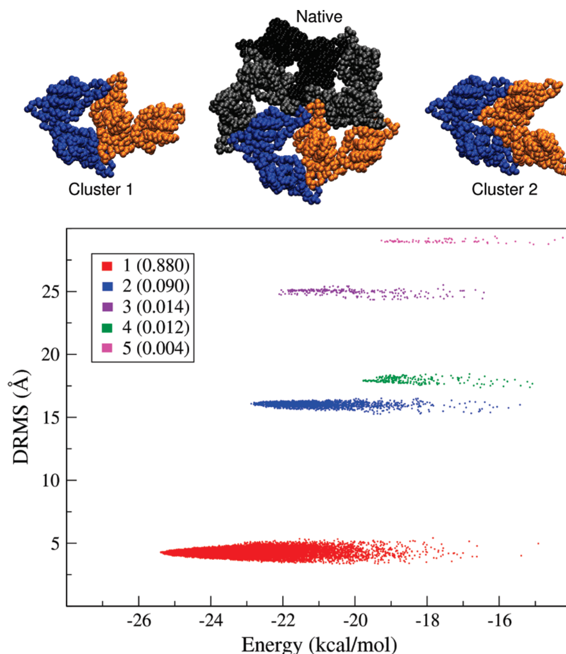

Figure 9. Bound structures of two viral capsid fragments. The DRMS from the bound structure is plotted against the potential energy for two fragments from the HBV virus capsid. Colors identify the different clusters of bound conformations. Population relative to the total bound conformations appears in brackets in the key. Conformations representative of clusters 1 and 2 are shown above the graph, together with a five-unit fragment of the HBV native structure.

6. Conclusions

We report a successful parallel CPU−GPU CUDA implementation of the Kim−Hummer coarse-grained model for replica exchange Monte Carlo (REMC) protein simulations. Our software is designed for new hybrid high-performance computing architectures combining multicore with GPU accelerators. This type of relatively low-cost parallel architecture has great relevance for researchers in developing countries, where High Performance Computing centers are generally not available.

Our hybrid parallel implementation employs multithreading for the Monte Carlo (MC) replicas and asynchronous calls to the potential evaluation kernel on the GPU. We did not explore CPU optimization further than multithreading, as the real performance gains are to be found in optimizing the GPU code. The simulation runtime is wholly dependent on the interaction potential calculations, which accounts for upward of 98% of the runtime. For the GPU kernel implementation, we found that the random access contact potential lookups are the chief performance bottleneck, as they prohibit defined optimal GPU memory usage patterns.

In general, best performance of our GPU kernel was achieved through adherence to the three chief tabled "CUDA best practices": maximizing parallel execution and optimizing memory and instruction usage,[22] although our solution illustrates that, for algorithms that do not fit the perfect GPU programming model, counterintuitive configurations may perform surprisingly well. Careful assessment of the performance of various memory locations for storing contact potential lookup table was key to achieving optimal perfor-

mance of our code on the nVIDIA GTX280 GPU architecture, which we finally achieved by bypassing the use of shared memory entirely in favor of texture memory.

Although the optimal balance between occupancy, memory usage, and type of memory used we identified is specific to the GTX280 hardware, in general nVIDIA's adherence to their architecture ensures that the optimizations performed on our code are applicable to past and future hardware.[21] With respect to the general applicability of our code, we note that we found that different memory resource configurations do not cause extreme changes to the overall simulation runtime and to our implementation autotunes to achiev a block size for optimal kernel occupancy. Tuning of memory usage and problem decomposition remains relevant for the older GPU cards, which are likely to be around for a while. The new nVIDIA Fermi architecture has altered the resources on the GPU. There are four times as many cores per SM, and L2 cache has been introduced, along with coherent caching, making the GPU caching model more like the CPU caching model.[34] As the CUDA model has not changed for the Fermi architecture, performance of our code is likely to improve because of the increase in the number of cores. Our simulations require a high degree of accuracy for rotational transformations, which involves square roots and sine and cosine functions. These functions still do not meet the Institute of Electrical and Electronics Engineers (IEEE) 754 compliance on GF100 nVIDIA GPUs. Therefore, it is still necessary to perform the MC mutations on the CPU with the new generation of GPU hardware. The adoption of OpenCL for general GPU programming means that the same code, when ported to OpenCL, can be compiled to run on both nVIDIA and ATI devices. The generalization of OpenCL removes vendor concepts, such as texture memory, as the effects of texture memory are afforded by the true caching abilities of new hardware. Porting of our code to OpenCL is likely to be straightforward. The CUDA Driver API and OpenCL are very similar, with a high correspondence between functions and most differences relating to syntax.[37] Critically, the model and manner in which a GPU must be used and the topics discussed here in macro-tuning our kernels are relevant for both Cuda and OpenCL. We found very good performance of our accelerated parallel implementation, achieving over 1400 times speedup over a serial solution for simulations of systems larger than 4000 residues. On a dual-core machine with one GTX280 card, our GPU-accelerated code is capable of up to 9 million MC iterations per day with our largest benchmark simulation of 7668 residues, whereas the CPU, fully utilizing both cores, manages only 12 000. This allows for more thorough testing of the Kim−Hummer coarse-grained model. We find that inclusion of the ubiquitin C-terminus tail increases both the binding affinity and the specificity of binding for the UIM/Ub system, even with a rigid model. We also report successful preliminary simulations using the Kim−Hummer potential of the binding of two HBV capsid components, where by far the largest cluster is that representing the correctly bound conformation.

Simulations of massive protein structures are now within reach. We intend to use our system for a novel complete simulation of the assembly of viral proteins into a viral capsid, a task previously prohibited by the excessive computation time required. Investigation of this process will give new insights into molecular self-assembly and may yield insights useful in the development of therapeutic drugs.

Finally, we note that, although we have only considered a specific model for protein−protein interactions, our implementation could easily be generalized to other types of interaction functions, as well as to other types of coarse-grained macromolecules (e.g., DNA). Finally, the effective parallelization approach developed in this work is generally applicable to N-body problems that require similar random access to lookup tables, where aspects of the interaction between bodies are dependent on their type or state.

Acknowledgment. The authors thank the Royal Society and the South African National Research Foundation (NRF) for a SA-UK Science Networks award. R.B. is supported by a Royal Society University Research Fellowship.

References

(1) Tramontano, A. Problem 7. *In The Ten Most Wanted Solutions in Protein Bioinformatics; Chapman & Hall/CRC mathematical biology and medicine series*; Etheridge, A., Gross, L., Lenhart, S., Maini, P., Safer, H. Voit, E., Eds.; CRC Press: Boca Raton, FL, 2005; pp 117−139.

(2) Halperin, I.; Ma, B.; Wolfson, H.; Nussinov, R. Principles of docking: an overview of search algorithms and a guide to scoring functions. *Proteins* **2002**, *47*, 409–443.

(3) Levitt, M. A simplified representation of protein conformations for rapid simulation of protein folding. *J. Mol. Biol.* **1976**, *104*, 59–107.

(4) Friedrichs, M. S.; Wolynes, P. G. Toward protein tertiary structure recognition by means of associative memory Hamiltonians. *Science* **1989**, *246*, 371–373.

(5) Shaknovich, E. I.; Gutin, A. M. Implications of thermodynamics of protein folding for evolution of primary sequences. *Nature* **1990**, *346*, 773–775.

(6) Skolnick, J.; Kolinski, A. Simulations of the folding of a globular protein. *Science* **1990**, *250*, 1121–1125.

(7) Karanicolas, J.; Brooks, C. L., III. The origins of asymmetry in the folding transition states of protein L and protein G. *Protein Sci.* **2002**, *11*, 2351–2361.

(8) Buchete, N.-V.; Straub, J. E.; Thirumalai, D. Anisotropic coarse-grained statistical potentials improve the ability to identify nativelike protein structures. *J. Chem. Phys.* **2003**, *118*, 7658–7671.

(9) Tozzini, V. Coarse-grained models for proteins. *Curr. Opin. Struct. Biol.* **2005**, *15*, 144–150.

(10) Marrink, S. J.; Risselada, H. J.; Yefimov, S.; Tieleman, D. P.; de Vries, A. H. The MARTINI force field: coarse grained model for biomolecular simulations. *J. Phys. Chem. B* **2007**, *111*, 7812–7824.

(11) Kim, Y. C.; Hummer, G. Coarse-grained Models for Simulations of Multiprotein Complexes: Application to Ubiquitin Binding. *J. Mol. Biol.* **2008**, *375*, 1416–1433.

(12) Kirk, D. B.; Hwu, W.-M. W. Performance Considerations. In *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed.; Morgan Kaufmann: Burlington, MA, 2010.

(13) Tomov, S.; Nath, R.; Ltaief, H.; Dongarra, J. Dense Linear Algebra Solvers for Multicore with GPU Accelerators. In *Proceedings of IPDPS '10*; Sigarch, A., Ed.; IEEE Computer Society Press: Washington, D.C., 2010.

(14) Nyland, L.; Harris, M.; Prins, J. Fast N-Body Simulation with CUDA. In *GPU Gems 3*; Nguyen, H., Ed.; Addison Wesley Professional: Reading, MA, 2007; pp677−695.

(15) Belleman, R. G.; Bedorf, J.; Zwart, S. P. High Performance Direct Gravitational N-body Simulations on Graphics Processing Units II: An implementation in CUDA. *New Astron.* **2007**, *13*, 103–112.

(16) Friedrichs, M. S.; Eastman, P.; Vaidyanathan, V.; Houston, M.; Legrand, S.; Beberg, A. L.; Ensign, D. L.; Bruns, C. M.; Pande, V. S. Accelerating molecular dynamic simulation on graphics processing units. *J. Comput. Chem.* **2009**, *30*, 864–872.

(17) Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. Accelerating Molecular Modeling Applications with Graphics Processors. *J. Comput. Chem.* **2007**, *28*, 2618–2640.

(18) Rodrigues, C. I.; Hardy, D. J.; Stone, J. E.; Schulten, K.; Mei, W. GPU acceleration of cutoff pair potentials for molecular modeling applications. In *Proceedings of the 5th conference on Computing frontiers (CF'08)*; ACM: New York, 2008; pp 273−282.

(19) Hardy, D. J.; Stone, J. E.; Schulten, K. Multilevel summation of electrostatic potentials using graphics processing units. *Parallel Comput.* **2009**, *35*, 164–177.

(20) Schmid, N.; Bötschi, M.; van Gunsteren, W. F. A GPU Solvent-Solvent Interaction Calculation Accelerator for Biomolecular Simulations Using the GROMOS Software. *J. Comput. Chem.* **2010**, *31*, 1636–1643.

(21) CUDA Programming Guide, version 2.3; NVIDIA Corporation: Santa Clara, CA, 2009; http://developer.download.nvidia.com/compute/cuda/2/toolkit/docs/NVIDIAUDArogramminguide.3.pdf. Accessed: August 19, 2010.

(22) NVIDIA, CUDA Best Practices Guide 2.3; NVIDIA Corporation: Santa Clara, CA, 2009; http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf. Accessed: August 19, 2010.

(23) Mackerell, A. D.; Feig, M.; Brooks, C. L. Empirical force fields for biological macromolecules: overview and issues. *J. Comput. Chem.* **2004**, *25*, 1584–1604.

(24) Fraczkiewicz, R.; Braun, W. Exact and efficient analytical calculation of the accessible surface areas and their gradients for macromolecules. *J. Comput. Chem.* **1998**, *19*, 319–333.

(25) Metropolis, N.; Rosenbluth, A. W.; Rosenbluth, M. N.; Teller, A. H.; Teller, E. Equation of State Calculations by Fast Computing Machines. *J. Chem. Phys.* **1953**, *21*, 1087–1092.

(26) Hastings, W. K. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika* **1970**, *57*, 97–109.

(27) Sugita, Y.; Okamoto, Y. Replica-exchange molecular dynamics methods for protein folding. *Chem. Phys. Lett.* **1999**, *314*, 141–151.

(28) Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S.; Karplus, M. CHARMM: A Program for MacroMolecular Energy, Minimization and Dynamics Calculations. *J. Comput. Chem.* **1983**, *4*, 187–217.

(29) Swanson, K. A.; Kang, R. S.; Stamenova, S. D.; Hicke, L.; Radhakrishnan, I. Solution structure of Vps27 UIM-ubiquitin complex important for endosomal sorting and receptor downregulation. *EMBO J.* **2003**, *22*, 4597–4606.

(30) Pelletier, H.; Kraut, J. Crystal structure of a complex between electron transfer partners, cytochrome c peroxidase and cytochrome c. *Science* **1992**, *258*, 1748–1755.

(31) Hess, B.; Kutzner, C.; van der Spoel, D.; Lindahl, E. GROMACS 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J. Chem. Theory Comput.* **2008**, *4*, 435–447.

(32) Matsumoto, M.; Nishimura, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.* **1998**, *8*, 3–30.

(33) Harris, M. Optimizing Parallel Reduction in CUDA; NVIDIA Corporation: Santa Clara, CA, 2007; http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf. Accessed August 19, 2010.

(34) nVIDIA's Next Generation CUDA Compute Architecture: Fermi. 2010; NVIDIA Corporation: Santa Clara, CA, 2010; http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Accessed August 19, 2010.

(35) van Meel, J. A.; Arnold, A.; Frenkel, D.; Portegies; Belleman, R. G. Harvesting graphics power for MD simulations. *Mol. Simul.* **2008**, *34*, 259–266.

(36) Tomov, S.; McGuigan, M.; Bennett, R.; Smith, G.; Spiletic, J. Benchmarking and implementation of probability-based simulations on programmable graphics cards. *Comput. Graph.* **2005**, *29*, 71–80.

(37) nVIDIA OpenCL Jumpstart Guide; NVIDIA Corporation: Santa Clara, CA, 2009; http://developer.download.nvidia.com/OpenCL/NVIDIA_OpenCL_JumpStart_Guide.pdf. Accessed August 18, 2010.

CT1003884