Multiscale Approximation with Graphical Processing Units for Multiplicative Speedup in Molecular Dynamics

Ramu Anandakrishnan, Mayank Daga*, Alexey Onufriev, Wu-chun Feng
Department of Computer Science, Virginia Tech, Blacksburg, VA 24060
Correspondence: wfeng@vt.edu; *Work completed while at Virginia Tech.

ABSTRACT

The timescales and structure sizes accessible via simulations of atomistic molecular dynamics (MD) can be advanced substantially by two independent techniques: (1) many-core parallelization with graphics processing units (GPUs) and (2) multiscale approximation with hierarchical charge partitioning (HCP). Achieving efficient many-core parallelization on the GPU generally requires highly synchronized and regular computation across the GPU. However, multiscale methods can result in highly asynchronous and irregular processing. Thus, one might expect that realizing such multiscale algorithms on the GPU would result in an overall loss of performance and that the total speedup obtained would be less than the product of the individual speedups for the two techniques separately, i.e., less than multiplicative speedup.

To test this expectation in the context of atomistic MD, we designed and implemented our HCP multiscale method on NVIDIA GPU platforms. The HCP code was implemented in NAB, short for nucleic acid builder, and tested using the distance-dependent-dielectric, implicit solvent model. (NAB is the molecular dynamics module in the open-source AmberTools v1.4.) We show that for the HCP multiscale approximation and the common MD simulation model considered here, the degradation in performance due to asynchronous and irregular processing is mostly offset by a corresponding reduction in *other* asynchronous operations and slow global memory accesses. As a result, we realize near multiplicative speedups. For example, for a 475,000-atom virus capsid we were able to achieve a 11,071-fold combined speedup, only slightly less than the 11,706-fold multiplicative limit speedup – 48.0-fold from the parallelization on the GPU times 243.9-fold from the multiscale approximation. The overall speedup depends on structure size, with smaller structures having lower speedups. An additional benefit of the HCP implementation on the GPU is the reduced memory requirement, which allows the processing of much larger structures that would otherwise be impossible on the limited memory GPU platform.

BCB '16, October 02-05, 2016, Seattle, WA, USA
 ISBN 978-1-4503-4225-4/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2975167.2975214.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming techniques - parallel programming; I.6.3 [**Computing Methodologies**]: Appliations - simulation and modeling; J.3 [**Computer Applications**]: Life and medical sciences

General Terms

Algorithms, Performance

Keywords

Biomolecular electrostatics, multiscale approximation, molecular dynamics, graphical processing unit (GPU)

1. INTRODUCTION

Simulations of atomistic molecular dynamics (MDs) are used to gain insight into the structure and function of biological molecules. However, the timescales that can be studied via such simulations are often far smaller than those required to observe many biological processes, such as protein folding and ligand binding [1, 2, 3]. These limitations increase rapidly with the structure size. Two common approaches for extending the timescales and structure sizes accessible to atomistic MD simulations are (i) parallelization across multi- and many-core platforms [4, 5, 6] and (ii) algorithmic approximations [7, 8, 9, 10, 11, 12].

The widespread use of graphics processing units (GPUs) for general-purpose computations in desktops and workstations has made them attractive as accelerators for high-performance parallel programs [13]. This increased popularity has been propelled by its (i) sheer computing power, (ii) superior performance/price ratio, and (iii) compelling performance/power ratio. For example, an 8-GPU cluster, costing thousands of dollars, can simulate 52 ns/day of the JAC Benchmark as compared to 46 ns/day on the Kraken supercomputer at ORNL, which costs millions of dollars [14]. The emergence of GPUs as an attractive high-performance computing platform is also evident from the fact that three out of the top five fastest supercomputers on the November 2011 Top500 list employ GPUs as accelerators [15]. A wide range of applications in image and video processing, financial modeling and scientific computing have also been shown to benefit from the use of GPUs [16, 17, 6, 18, 19].

GPUs, however, are not a panacea for all of computing. For maximum efficiency on the GPU, computational operations across processing cores need to be synchronized. However, an important class of algorithmic approximations, i.e.,

multiscale approximations, have highly asynchronous computational requirements. These approximations involve numerous divergent branches depending on the relative distances between interacting atoms, thereby resulting in non-uniform computational requirements across processors. Thus, when multiscale algorithms are realized on the GPU, one may expect to achieve less than multiplicative speedups, i.e., the total speedup is less than the product of the individual speedups for the multiscale algorithm and the GPU, respectively [20]. To test this expectation, we implemented our hierarchical charge partitioning (HCP) [21] multiscale algorithm on the NVIDIA GPU. The HCP code itself is implemented in NAB, the open-source molecular dynamics module in AmberTools v1.4 [22], and tested using the distance-dependent-dielectric implicit-solvent model [23].

Contrary to our initial expectations, the implementation resulted in near multiplicative speedups. The loss in performance due to the additional divergent branches in HCP algorithm was mostly offset by a corresponding reduction in the number of other divergent branches that otherwise would have been needed to be considered. In short, these other divergent branches are bypassed by the HCP algorithm. In addition, the HCP algorithm benefits from a reduction in the number of accesses to slower global memory.

The rest of the paper is arranged as follows. In section 2, we briefly describe the HCP multiscale algorithm; the distance-dependent-dielectric, implicit-solvent model used for simulation; and the targeted GPU platforms. Then we discuss the mapping of the HCP algorithm to the GPU in section 3. In section 4, we describe the CPU and GPU platforms and the structures and protocols used for testing. Finally, in section 5, we discuss and analyze the results of our tests. Our findings are summarized in section 6.

2. BACKGROUND

A key objective of this study is to characterize the effect of combining multiscale approximations and many-core parallelization on a GPU towards the realization of multiplicative speedups in molecular dynamics. For the purpose of this study, we chose the hierarchical charge partitioning (HCP) multiscale approximation and accelerated it using the NVIDIA GPU platform. Our HCP software interfaces with NAB, the open-source molecular dynamics (MD) module in AmberTools version 1.4 [22]. The performance of the implementation is tested using the distance-dependent-dielectric, implicit solvent model. The HCP approximation, the distance-dependent-dielectric model, and the NVIDIA GPU platform are briefly described below.

2.1 Hierarchical Charge Partitioning (HCP)

The hierarchical charge partitioning (HCP) approximation [21] exploits the natural partitioning of biomolecules into constituent structural components to speed-up the computation of electrostatic interactions with limited and controllable impact on accuracy. Biomolecules can be systematically partitioned into multiple molecular *complexes*, which consist of multiple polymer chains or *subunits*, which are made up of multiple amino acid or nucleotide *groups*. These components form a hierarchical set with, for example, complexes consisting of multiple subunits, subunits consisting of multiple groups, and groups consisting of multiple atoms. Atoms represent the lowest level in the hierarchy while the highest level depends on the problem. The charge distribution of the above components, other than at the atomic level, is approximated by a small set of point charges. The electrostatic effect of distant components is calculated using the smaller set of point charges, while the full set of atomic charges is used for computing electrostatic interactions within nearby components. The level of approximation used in the computation varies depending on distance from the point in question: the farther away the charges, the higher the level of the component used in the approximation.

Consider, for example, a structure consisting of four levels, 0-3, as shown in Figure 1. A separate threshold distance, h_1, h_2, and h_3, is defined for levels 1, 2 and 3, respectively. For complexes (level 3) farther than h_3 from the point of interest, the approximate charges for the complex are used in the computation. Otherwise, for subunits (level 2) within the complex that are farther than h_2, the approximate charges for the subunit are used in the computation. Otherwise, for groups (level 1) within the subunit that are farther than h_1, the approximate charges for the group are used in the computation. Finally, individual atomic charges are used in the computations for charges within the level 1 threshold distance h_1. This top-down algorithm results in $\sim \ N \log N$ scaling based on assumptions generally consistent with realistic biomolecular systems.

2.2 Distance-Dependent-Dielectric, Implicit Solvent Model

Calculating the long-range electrostatic interactions in an N-atom system takes $O(N^2)$ time complexity. As a consequence, it is the primary computational bottleneck in molecular dynamics simulations. Implicit solvent models, which use an approximation for computing long-range interactions, reduce this computational cost considerably by analytically representing solvent atoms as a continuum. Solvent atoms typically represent a majority of the atoms in the system. Among other benefits, implicit solvent models can sample conformation space much faster and can instantaneously incorporate the effect of dielectric changes in the solvent due to changes in the solute charge distribution. For this study, we used a simple implicit solvent model, the sigmoidal distance-dependent-dielectric model [23]. This model computes the long-range electrostatic potential ϕ at a distance of r from a charge q as

$$\phi = \frac{q}{\epsilon(r) r} \quad (1)$$

$$\epsilon(r) = D - \frac{(D-1)}{2}\left[(rS)^2 + 2rS + 2\right] e^{-rS} \quad (2)$$

where $\epsilon(r)$ is the distant-dependent-dielectric function and D (= 78) and S (=0.16) are constants.

2.3 GPU Architecture and API

Graphics processing units (GPUs) have traditionally been used to accelerate image rendering. However, the evolution of the GPU into a compute-capable, parallel processing platform has accelerated its adoption to speed-up computations in various data-parallel applications. Thus, to speed-up the HCP algorithm and the distance-dependent-dielectric model for molecular dynamics simulations, we used NVIDIA GPUs and the CUDA programming interface.

NVIDIA Tesla GPUs consist of 240 to 512 execution units, grouped into 16 to 30 streaming multiprocessors (SMs). Each SM can run on the order of a thousand threads, thereby en-

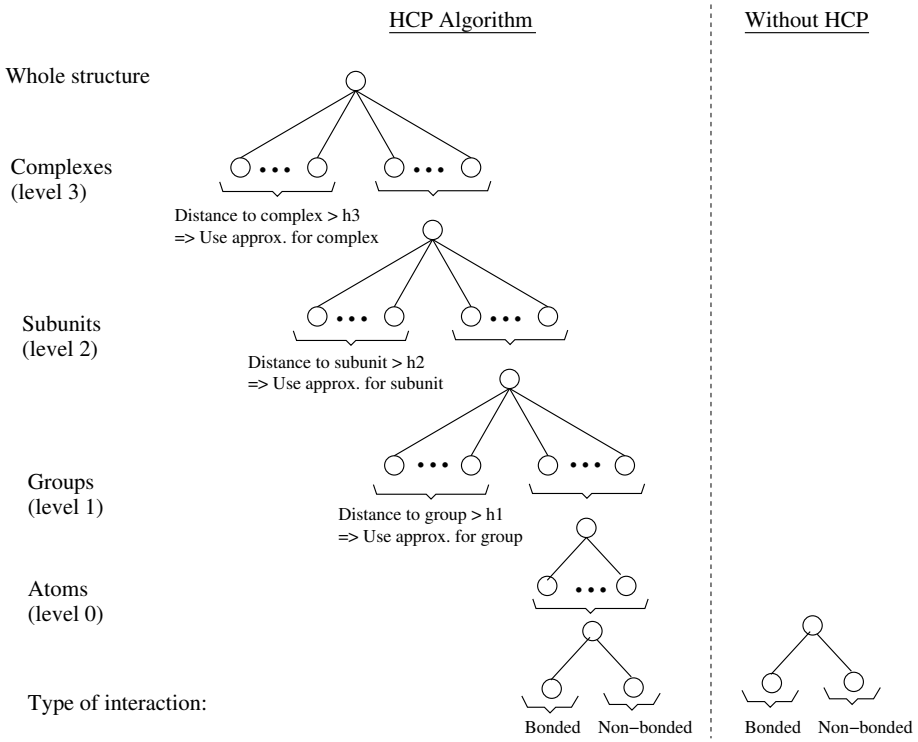


Figure 1: Illustration of the HCP multiscale algorithm. Predefined threshold distances (h_1, h_2, h_3) are used to determine the level of approximation used in the HCP approximation. At the lowest (atomic) level the computation involves either a bonded or a non-bonded interaction. This top-down algorithm results in $\sim NLogN$ scaling compared to a $\sim N^2$ scaling without HCP.

abling massively parallel computation. Multiple threads on a GPU execute the same instruction and hence, is a Single Instruction Multiple Thread (SIMT) architecture. This makes GPUs very suitable for applications that exhibit data parallelism, i.e., the operation on one data element is independent of the operations on other data elements. Therefore, it is well suited for molecular dynamics, where the force on one atom can be computed independently of all others.

On NVIDIA GPUs, threads are organized into groups of 32, referred to as a *warp*. When threads within a warp follow different execution paths, such as when encountering a conditional, a divergent branch takes place, thus affecting performance. Furthermore, GPUs have more transistors devoted to performing computations than for caching and managing control flow. This means that on a GPU, computations are much faster compared to a typical CPU, but memory accesses and divergent branching instructions are slower. The effect of slower memory access is mitigated by initiating thousands of threads, such that when one thread is waiting on a memory access, other threads can perform meaningful computations.

Every GPU operates in a memory space known as global memory. Data operands needed for computation on the GPU must first be transferred to the GPU. This process of transferring data to GPU memory is performed over the PCIe interface, making it an extremely slow process. In general, memory transfers should be kept to a minimum to obtain optimum performance. In addition, accessing data from the GPU global memory incurs a cost of 400 to 600 clock cycles, and hence, *on-chip* memory should be used to reduce global memory traffic when possible. On a NVIDIA GT200-based architecture, each SM contains a 16KB of high-speed, scratch-pad memory that is known as shared memory. Shared memory enables local re-use of data, thereby reducing traffic to global memory that is off-chip.

For the NVIDIA GF100-based Fermi architecture, each SM contains 64KB of on-chip memory, which can either be configured as 16KB of shared memory and 48KB of L1 cache or vice versa. Each SM also consists of a L2 cache of size 128KB. The hierarchy of caches on the Fermi architecture allows for more efficient global memory access patterns.

CUDA provides a C/C++ language extension with application programming interfaces (APIs). A CUDA program is executed by a *kernel*, which is effectively a function call to the GPU, launched from the CPU. CUDA logically arranges the threads into blocks which are in turn grouped into a grid. Each thread has its own ID that provides for a one-to-one mapping. Each block of threads is executed on a SM and shares data via shared memory.

3. MAPPING HCP ONTO THE GPU

In this section, we present our approach to mapping the hierarchical charge partitioning (HCP) algorithm onto the GPU. The approach consists of four phases, many of which should be co-designed together: (1) identifying the parallelizable parts of HCP, particularly those parts that involve a significant amount of computation; (2) selecting the appropriate granularity of computation on the GPU; (3) minimizing or amortizing the overhead of data transfer between the CPU and GPU; and (4) minimizing the memory footprint, and in turn, supporting the simulation of larger structures.

3.1 Approach

The parallel decomposition of molecular dynamics for het-

erogeneous machines consists of a partitioning of work for the host CPU and the accelerator cores. Molecular dynamics encompasses the computation of neighbor-list and forces between the atoms of a biomolecule, of which the computation of forces is the more frequent operation. Thus, one approach to partition the entire work is by accelerating force computation on the GPU and computing the neighbor-list on the CPU. The atomic forces result from the bonded and non-bonded interactions. The bonded interactions arise due to bond-stretching, angle-bending and bond-rotation. Realistically, an atom is bonded to only a few of its neighbors and hence, the bonded interactions are not compute intensive. On the other hand, the non-bonded interactions, which include the electrostatics and the van der Waals' interactions, are the primary computational bottleneck [24]. The long-range nature of the electrostatic interactions make them particularly difficult to compute because they scale as $O(n^2)$ or as $\sim O(n \log n)$ when approximation algorithms like HCP are used, where n is the number of atoms. Our experiments on simulating various molecules, ranging from 600 atoms to 475,500 atoms, with NAB on the Intel Xeon E5404 CPU, show that the computation of electrostatic interactions accounts for more than 90% of the total execution time in all the cases. Thus, it is sufficient to accelerate only the non-bonded interactions on the GPU. However, as explained in Section 3.2, partitioning the non-bonded and bonded interactions on two compute devices results in an increase in the data transfers to and from the GPU.

One of the artifacts of today's GPUs is that they require data to be transferred from the host to the device memory, which is of limited size (4 GB on the GPU that we have used in our experiments). Therefore, the charges and coordinates of all the components (atoms, group, subunits and complexes) are transferred to the GPU memory. Computing the neighbor-list on the CPU requires it to be transferred to the device memory as well. The storage complexity of maintaining a neighbor-list is $O(n^2)$ and hence, transferring it to the device dramatically reduces the size of the biomolecule that can be simulated. In order to reduce the memory footprint of our application, we eliminate the neighbor-list and instead determine the neighbors on the fly on the GPU, as explained in Section 3.3.

In our implementation, the force on each atom is computed by a GPU thread. A pseudocode for force computation is shown in Figure 2. We launch a kernel with 256 threads within a CUDA block. We limited our execution to only 256 threads within a CUDA block to avoid register spilling. For a large molecule, we divide the total atoms into 'n' chunks where 'n' is the total number of threads launched. We then assign each thread to compute the force on every n^{th} atom. This is done to avoid the kernel overhead multiple times. The scaling of GPU performance hits a plateau when the number of threads that can be executed simultaneously reaches the upper limit. This limit is governed by the amount of per-thread register utilization by the implementation. Increasing number of threads beyond this limit does not result in any improved performance as the *extra* threads must to wait until some of the threads finish execution.

We store the atomic coordinates in the per-SM shared memory to take the advantage of the significant reuse of atomic coordinates. Therefore for each atom, we store 12 bytes (4 bytes each x, y, z coordinates) in the shared memory. As the maximum number of threads that can be launched

```
1  global void ForceComp(float *g_coords,
2     float *forces, int* cutoffs) {
3     int tid = get_global_id(0);
4     int lid = get_local_id(0);
5     float force_comp = 0.0;
6     shared float* l_coords[1024];
7
8     // copy atomic coordinates to shared memory
9     l_coords[lid] = g_coords[tid];
10    barrier;
11
12    // compute force using the HCP algorithm
13    while (component_1) {
14       if (dist. bet. atom_i > cutoffs[1]) {
15          /* compute force via approximation
16             and update force_comp */
17       }
18       else {
19          while (component_2) {
20             if (dist. bet. $atom_i$ > cutoffs[2]) {
21                /* compute force via approximation
22                   and update force_comp */
23             }
24             else {
25                /* ...so on until all the components
26                   and atoms are exhausted... */
27             }
28          }
29       }
30    }
31
32    // update the force
33    forces[tid] = force_comp;
34 }
```

Figure 2: Pseudocode of the Kernel for Force Computation. Atomic coordinates are first loaded into the shared memory and then each thread is tasked to compute force on one atom using the HCP algorithm.

per SM is 1K, in theory we require 12KB of shared memory which is well within the amount of shared memory available. For larger molecules, the coordinates of the atoms already been computed are over-written by the next chunk of atoms.

3.2 Minimizing CPU-GPU Data Transfers

The total computation time for a molecular dynamics simulation, T, can be summed up as follows:

$$T = t_{CPU} + t_{data-copy} + t_{GPU} \quad (3)$$

The time to transfer data is bound by the bandwidth of the PCIe 2.0 interface, typically 4-5 GB/s in each direction due to the API overhead incurred in an application. Therefore, data transfers over the PCIe should be avoided or minimized to achieve better performance.

Computing the bonded and non-bonded interactions on different compute devices results in significant data-transfer overhead. In simulating various molecules with the HCP approximation in NAB on the NVIDIA C1060 GPU, our experiments show that data-transfer time can account for as much as 70% of the total time spent on the GPU for the virus capsid molecule. Why? After every timestep of the MD simulation, the non-bonded forces must be transferred back to the *host* main memory in order to be summed up with the bonded forces that were computed on the CPU. The summation of the forces is necessary to compute the atom coordinates for the next timestep of MD simulation. Thus, for every timestep, $12 * n$ bytes (4 bytes each for x, y, z coordinates in single precision) of non-bonded forces are transferred to the host and then $12 * n$ bytes (4 bytes each for x,

y, z coordinates in single precision) of the atom coordinates are transferred back to the device, where n is the number of atoms in the molecule. The amount of data to be transferred becomes substantial for a large molecule, e.g., 500,000 atoms or more. For meaningful simulations wherein nanosecond scales need to be achieved, the transfer of data must then take place at least a million times (for 10^6 timesteps). As evident, this approach involves extremely *slow* CPU-GPU memory transfers at every step of the simulation, leading to performance degradation.

To reduce the number of CPU-GPU memory transfers, we computed both the non-bonded and bonded interactions on the GPU. This approach mitigated the per timestep memory transfers and allowed for calculating the new atomic coordinates on the GPU itself.

3.3 Minimizing Memory Footprint on the GPU

The disadvantage of the GPUs is that they require data to be transferred from the host to the device memory. Therefore, the size of the molecule that can be simulated is now bound by the amount of device memory. In order to most efficiently use the device the memory, we eliminated the most prominent data-structure in our application, which was the neighbor-list, a $4 * n^2$ bytes array (n being the number of atoms in the molecule). As a result, we are bound to simulate a molecule with at most 32K atoms on a GPU with 4 GB of device memory. HCP also requires pair-lists for every component for which the interactions are to be approximated along with the atomic neighbor-list in the GPU memory. This makes the total amount memory, M, required by pair-lists alone to be

$$M = 4 * n(n + r + s + c) \text{ bytes} \quad (4)$$

where n, r, s, c are number of atoms, groups, subunits and complexes in the molecule, respectively. Therefore, the pair-lists can account for a large amount of memory on the GPU, e.g., for virus capsid, $n = 475,500, r = 30,700, s = 60, c = 1$, which results in almost a terabyte of GPU memory required by the pair-lists.

To reduce memory utilization, we eliminated the need for a pre-calculated pair-list, and instead, determined if a pair of coordinates should be included in the computation, on the fly. Our approach, akin to the one used by Brown et al. in [25], significantly extended the structure size that could be simulated and thus enabled us to simulate the 475,000-atom virus capsid, which was not otherwise possible.

4. EXPERIMENTAL SETUP

We tested our GPU implementation on two NVIDIA GPUs (Tesla C1060 & Fermi Tesla C2050) and compared it to the baseline CPU code running on the host machine. The host machine consists of a 2.0-GHz Intel Xeon E5404 CPU with 8-GB DDR2 SDRAM and runs 64-bit Ubuntu 9.04 with the 2.6.28-18 Linux kernel. Programming the GPU was facilitated by the CUDA 4.0 toolkit for the C2050 and CUDA 3.2 for the C1060 with NVIDIA driver version 285.05.23.

4.1 Test Structures

To test the scalability of our approach we used nine different structures ranging in size from 632 to 475,500 atoms. Table 1 shows the characteristics of the structures that we used; it also lists the threshold distances used for each level of HCP for each structure. These are the recommended threshold distances as described in Anandakrishnan et al. [26]. For the purpose of the analysis presented in this work, we use the 1-charge HCP approximation, where the charge distribution of components is approximated by a single charge. Increasing the number of charges used in the approximation would increase accuracy and computational cost.

4.2 Molecular Dynamics Protocol

Unless otherwise stated, the following parameters and protocol were used for all simulations. The simulations use the sigmoidal distant-dependent-dielectric [23] implicit-solvent model. The HCP threshold distances used are listed in Table 1; 6-12 van der Waals interactions for HCP were computed using only the atoms that are within the level 1 threshold distance, i.e., atoms that are treated exactly. The simulations used the Amber ff99SB force field [28]. Langevin dynamics with a collision frequency of 50 ps^{-1} (appropriate for water) was used for temperature control, and the integration time step was 2 fs. Default values were used for all other parameters.

The simulation protocol consisted of five stages. First, the starting structure was minimized using the conjugate gradient method with a restraint weight of 5.0 kcal/mol/Å^2. Next, the system was heated to 300 K over 10 ps with a restraint weight of 1.0 kcal/mol /Å^2. The system was then equilibrated for 10 ps at 300 K with a restraint weight of 0.1 kcal /mol/Å^2, and then for another 10 ps with a restraint weight of 0.01 kcal/mol/Å^2. Finally, all restraints were removed for the production stage.

5. RESULTS AND DISCUSSION

Here we present an analysis of the following: (i) speedups due to the GPU, the HCP and the combined speedup due to both, (ii) the impact of divergent branching on the speedup, (iii) the limitation on structure size due to the limited GPU memory, (iv) the scaling with structure size, and (v) the stability of simulations. We show that near multiplicative speedups were achieved despite the introduction of additional divergent branching by the HCP algorithm. The largest structure that can be processed by our implementation is approximately 500,000 atoms, which is significantly larger than other GPU implementations [29]. We tested both single as well as double-precision floating point on the C2050 GPU even though the theoretical performance of double-precision is only half as good as that of single precision. We also show that using single precision and the HCP approximation in simulations does not lead to gross instabilities, and hence, it is not imperative to use double precision and suffer the performance degradation.

5.1 Speedup

Figures 3a and 3b depict the speedups obtained using single precision on two NVIDIA GPUs, i.e., Tesla C1060 and Tesla Fermi C2050, respectively. The figure presents four speedups: (i) speedup obtained due to GPU alone, (ii) speedup obtained due to HCP alone, (iii) the multiplicative limit which is the product of speedups due to GPU and HCP and lastly, (iv) the actual speedup that was realized due to the combination of HCP and GPU. For all these speedups,

[1]The microtubule structure was constructed as described in Wang and Nogales [27]

Table 1: Characteristics of Structures.

Structure	PDB ID	No. of Atoms	HCP Threshold Distance (Å)	
			Level 1	Level 2
10 bp B-DNA fragment	2BNA	632	15	n/a
Immunoglobulin binding domain	1BDD	726	15	n/a
Ubiquitin	1UBQ	1,231	15	n/a
Thioredoxin	2TRX	1,654	15	n/a
Carboxypeptidase A	1CBX	4,793	15	66
Hemoglobin	2HHD	8,782	15	66
Nucleosome core particle	1KX5	25,101	21	90
Microtubule sheet	[1]	158,016	15	48
Virus capsid	1A6C	475,500	15	66

the baseline was a single-threaded, CPU, all-atom computation, without any approximation. Speedup due to the GPU was computed by comparing the execution times of the all-atom computation on the CPU with that of the all-atom computation on the GPU, without the use of HCP approximation in both cases. Speedup due to HCP was computed by comparing the execution times of an all-atom simulation with and without the HCP approximation on the CPU. The idealistic goal is to achieve multiplicative speedup due to the combination of GPU and HCP.

Figure 3 indicates that we are close to achieving the *multiplicative limit* in terms of speedup due to the combination of GPU and HCP. The speedup achieved is closer to the multiplicative limit for the larger structures. Why? The larger structures can more efficiently utilize the GPU. For example, for the nucleosome core particle, the achieved speedup is within 3% and 8% of the multiplicative speedup, on C1060 and Fermi C2050, respectively, and for the virus capsid, the largest structure we tested, we are within 2.5% and 5%, on the C1060 and Fermi C2050, respectively. Another observation is that the performance improvement on the Fermi C2050 is better than on the C1060. This is as expected since the Fermi C2050 consists of L1 and L2 caches, which mitigates the impact of the random global memory accesses of the HCP algorithm. For memory accesses to be consecutive, all 32 atoms (atoms in a warp) need to follow similar execution paths, i.e., either they are approximated or not. However, due to the fact that each atom may meet different threshold requirements, it is common to have random memory accesses.

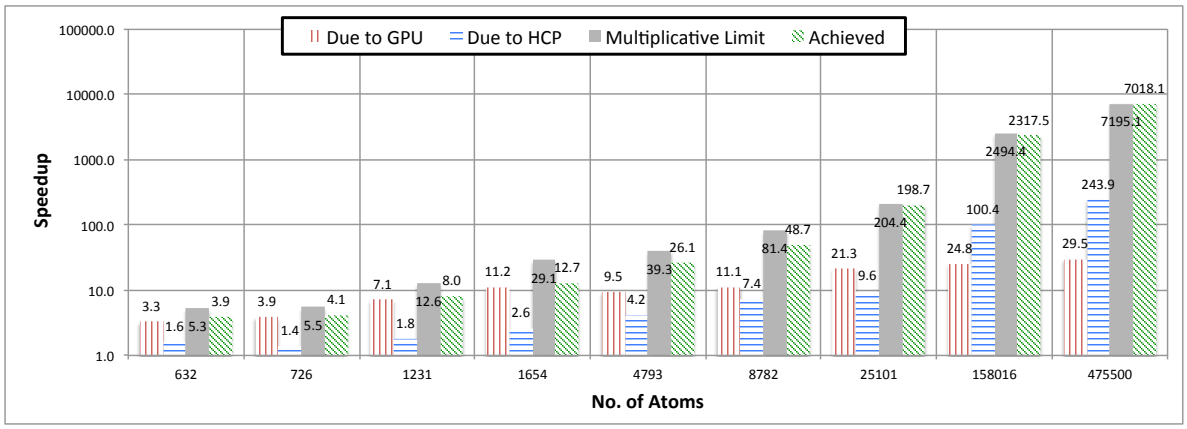
Figure 4 shows our experimental results on a C2050, with the basic data type being a double-precision, floating-point value. The speedup achieved with the double-precision implementation on the C2050 is roughly one-half or better than the single-precision implementation. The product specification indicates one-half peak performance when comparing double to single precision, so the results here indicate better-than-expected performance from the C2050 for double precision. Specifically, speedup for double precision compared to single precision is between 93% for the smallest structure tested and 54% for the largest. Since the number of computations scales as $\sim N^2$ (without the HCP approximation), where N is the number of atoms, smaller structures are more likely to be bandwidth-bound, whereas larger structures are more likely to be compute-bound. Thus, there is little or no penalty for double-precision computations on smaller structures, while larger structures incur nearly the full penalty for double-precision computation.

We achieved near multiplicative speedup by the judicious optimization of our GPU application. Ryoo et al. point out that GPUs have a large optimization search space [30] and hence, narrowing it down is imperative. We first checked whether HCP is memory bound or compute bound and as presented in [31, 32], HCP was found to be *memory bound*. Since, HCP is memory bound, we focused on optimizations that reduce the number of global memory transactions. We utilized the shared memory and constant memory available on GPUs, both of which help in the reduction of global memory accesses. Shared memory was used to store the coordinates and atomic charges while the constant memory, which acts as a read-only cache, was used to store all the constants that were required to compute the forces and energies. We also kept the number of memory transfers to a minimum since, they use the slow PCIe interface, thereby leading to performance degradation. We also re-structured our code so as not to use atomic operations for the computation of bonded interactions. All these strategies, in combination, helped us achieve near multiplicative speedup.

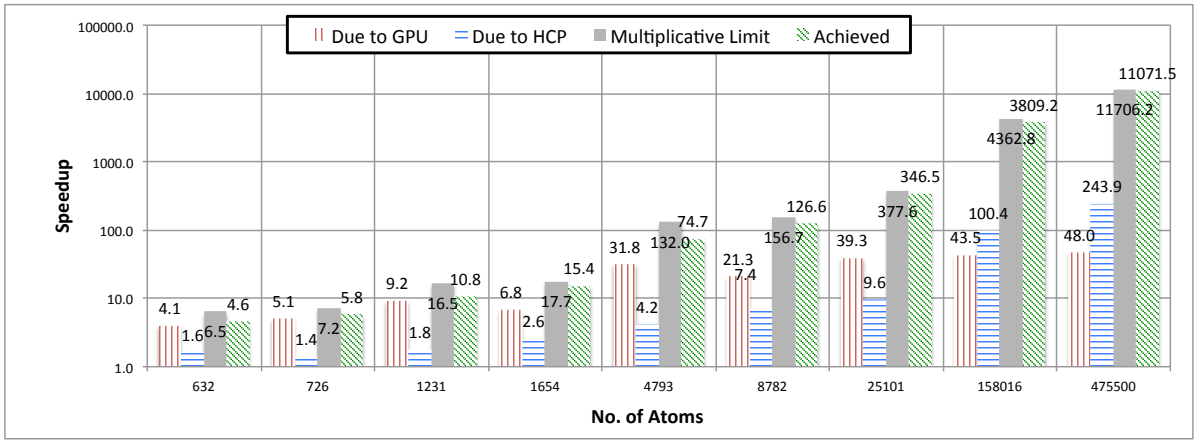
5.2 Divergent Branching

In our implementation, we assign each GPU thread with the task of computing the force at *one* atom of the molecule. From Figure 1, one would intuitively expect the HCP to introduce many divergent branches on the GPU, which lead to performance degradation. However, HCP actually reduces the total number of divergent branches when compared to an all-atom simulation. This is because for an all-atom simulation, a conditional is required to determine the type of interaction (*bonded or non-bonded*), which needs to be carried out for every atom of the molecule. Our analysis indicates that for some structures, this conditional results in as many divergent branches as all other conditionals combined. To exemplify, we present the number of divergent branches for a subset of structures on the C1060 GPU in Table 2. Looking at the difference in number of divergent branches, it becomes apparent that the said conditional increases the number of divergent branches by an order of magnitude. Use of HCP mitigates the effect of this conditional. For distant components, the HCP algorithm does not reach the stage where it is necessary to determine the type of interaction (*bonded or non-bonded*).

The HCP also reduces the number of global memory transactions by reducing the number of atomic coordinates that need to be fetched. For distant components, the coordinates of only the higher level component are required, bypassing all other molecular constituents. Table 2 portrays the number of per SM global memory transactions that occur, with and without the use of HCP. It can be seen that HCP re-



NVIDIA Tesla C1060



NVIDIA Fermi Tesla C2050

Figure 3: Speedups for single precision implementation (i) due to GPU, (ii) due to HCP, (iii) the multiplicative limit, and (iv) achieved. We are off by less than 8% of the multiplicative limit in terms of speedup due to the combination of GPU and HCP. The speedup on the Fermi C2050 is better than the C1060 due to the presence of L1 and L2 caches. For all the results, the baseline used is a no-approximation-CPU-serial implementation.

sults in an order of magnitude decrease in the number of global memory transactions. Being a memory-bound algorithm [31, 32], HCP benefits from the reduction in number of global memory accesses. Therefore, the combined effect of reduction in the number of global memory transactions as well as reduction in the number of divergent branches brings about performance improvement of HCP on the GPU which is as follows:

$$\begin{aligned} T_{HCP} &= T_{No\_HCP} - T_{gMem} - T_{divBranch} \qquad (5) \\ \therefore T_{HCP} &< T_{No\_HCP} \end{aligned}$$

where T_{HCP} is the execution time with HCP approximation on the GPU, T_{No_HCP} is the execution time without HCP approximation on the GPU, T_{gMem} is the time for global memory transactions, $T_{divBranch}$ is the time for divergent branching.

5.3 Memory Footprint

One of the constraints of GPU programming is dealing with the limited memory space available on the GPU. Due to this limitation, there arises a trade-off between the performance benefits that can be obtained on the GPU and the size of the structure that can be simulated. For example, current CUDA-based implementation of the MD module (pmemd) of AMBER [22] molecular modeling package (AMBER GPU) focuses on maximizing speedup on the GPU, whereas, our implementation tries to maximize speedup while being able to handle large structures. Hence, the largest structure that AMBER GPU can simulate is *1/20th* of what we can simulate, albeit the AMBER GPU has a higher speedup than our GPU implementation of NAB running without HCP.

AMBER GPU uses *scratch arrays* [29] to maximize speedup whereas we took advantage of the HCP algorithm to produce even higher speedup, and at the same time are able to handle much larger structures, though at the potential price of reduced accuracy. We also reduced memory utilization by using the strategy of not storing the *pair-list* on the GPU and hence, were able to simulate much larger structures. Table 3 depicts the amount of GPU memory our implementation requires. To simulate the 25,000-atom nucleosome core particle, our implementation uses only 17 MB of GPU memory. The nucleosome is the largest structure that the AMBER GPU implementation can simulate on the Fermi C2050 GPU, whereas on the NVDIA GTX295 GPU, with 896 MB of memory, the memory is insufficient for running the nucleosome [33].

The minimal memory requirement of our GPU implementation lets us run molecular dynamics simulation of nucle-

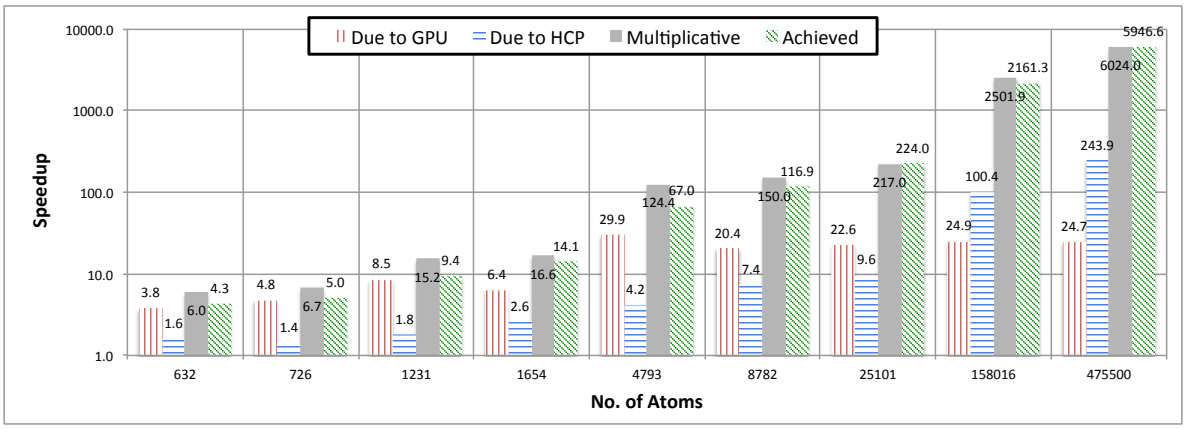


Figure 4: Speedup for double precision implementation on NVIDIA Fermi Tesla C2050

Table 2: Number of Divergent Branches and Global Memory Transactions for the C1060 GPU

No. of Atoms	Divergent Branches		Global Memory Transactions	
	all-atom	HCP	all-atom	HCP
632	92,610	51,022	753,238	423,679
726	57,586	42,378	442,936	324,421
1,231	59,572	37,226	393,862	237,830
1,654	127,260	54,150	1,013,492	433,213
25,101	8,088,980	382,275	157,252,461	12,074,672

Table 3: Memory Usage

No. of Atoms	Amount of Memory Used (MB)
632	0.43
726	0.50
1,231	0.83
1,654	1.13
25,101	16.88
158,016	107.06
475,500	323.46

osome core particle even on a Apple MacBook Pro laptop, which is not possible using the AMBER GPU program.

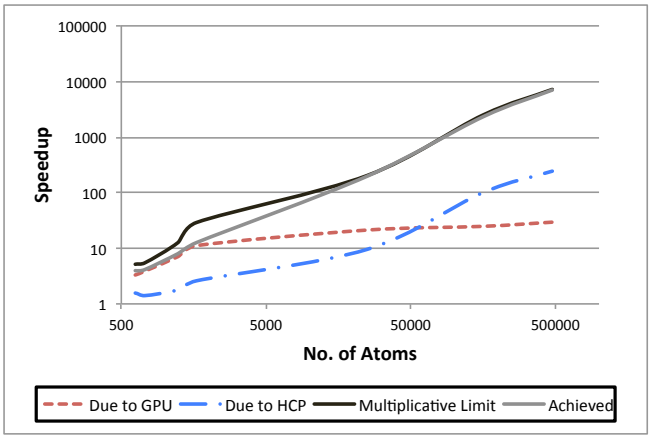
5.4 Scaling

In Fig. 5a and 5b, we show the scalability of our implementation with respect to structure size (number of atoms). The absolute speedup obtained is higher for the Tesla Fermi C2050 GPU due to (i) the greater number of processing cores and (ii) presence of L1 and L2 caches that improve the performance of the memory subsystem.

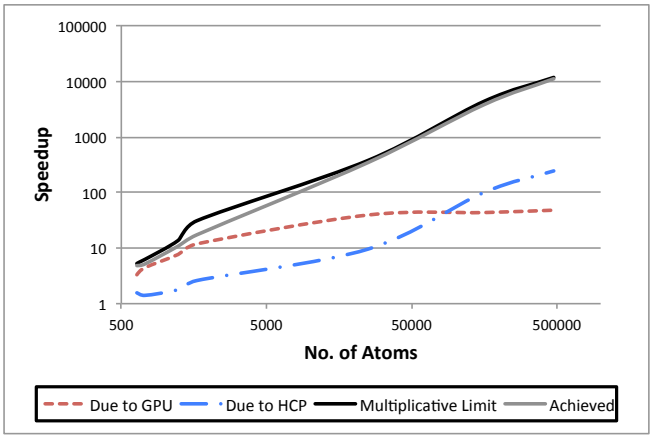
From Fig. 5a and 5b, it is clear that, as a function of structure size, the speedup due to the GPU grows fast at first, then levels off after the structure size is increased into the range of $\sim 10^4$ atoms. This behavior can be explained as follows. In order to realize the full potential of a GPU, its occupancy needs to be high, i.e., none of the processing cores of the GPU should be idle. Better occupancy leads to better performance. In our case, once the size of the structure reaches a several thousand atoms, occupancy of the GPU reaches its maximum, as described in Section 3.1. Beyond which, increasing the size does not increase the speed up as much, since the additional atoms need to "wait" for the computation on the preceding atoms to finish, only after which they can be operated upon. Performance on Fermi C2050 is better than C1060 as it allows for a greater number of threads to be executed simultaneously, primarily due to the presence of a larger register file. At the same time, the speedup due to the HCP continues to increase with the increase in number of atoms in the molecule. This is because, the HCP algorithm scales as $\sim N \log N$, where N is the number of atoms. Hence, it is almost entirely due to the benefits of HCP, that both the 'multiplicative limit' as well as the 'achieved' speedup continue to increase as the system size is increased beyond $\sim 10^4$ atoms.

5.5 Stability in MD Simulations

We have performed a standard all-atom 2 ps long (10^3 steps) constant temperature (300K) implicit solvent simulation for each of the structures list in Table . The simulation was extended to 50 ns (2.5×10^7 steps) for the four smallest structures, ranging in size from 623 atoms (a 12-base-pair long fragment of B-DNA) to 1654 atoms (protein thioredoxin). No numerical instabilities were noticed. For all but one of the structures, the RMS deviation, compared to the starting crystal structure, was within the range of about 2 Å expected for implicit solvent molecular dynamics[26]. For the B-DNA fragment, the RMS deviation was considerably larger than typically expected, ∼5-6 Å instead of ∼2 Å. However, the same deviation was observed for a control MD run on a CPU with double precision and without the HCP, showing that the larger than expected RMS deviation for the B-DNA fragment was due to the use of the simplified solvent model (distance-dependent dielectric) rather than inherent instabilities of either the HCP algorithm or its GPU implementation. That, of course, does not mean that the single precision arithmetic used here for the GPU implementation is completely "safe" to use, but that any errors due to the use of single precision on the GPU or the errors due to HCP do not accumulate or/and combine in a way to produce gross instabilities in the MD simulation, at least for the structures



NVIDIA Tesla C1060



NVIDIA Fermi Tesla C2050

Figure 5: Scalability. Speedup due to the GPU peaks off after a certain thousand atoms due to the inability to launch more threads. Speedup due to HCP continues to increase with the increase in the number of atoms and hence, scales as $\sim N \log N$. Connecting lines are shown to visually guide the eye.

tested here. However, there may be more subtle ways in which the single precision arithmetic may skew the results of MD simulation over millions of steps, that may not be evident from the RMS deviation metric. These issues are well beyond the scope of this work that focuses on demonstrating the ability to achieve multiplicative speedup from the combined use of the HCP and GPU. On GPU cards such as C1060 the performance loss of about a factor of ten in speed due to the use of double vs. single precision certainly warrants closer examination of the possibility of the use of single precision arithmetics in practical MD simulations. At the same time, for GPU cards such as C2050 the gain of only about a factor of two in speed due to single precision vs. the tried-and-true double precision may not be worth potential risks or even efforts associated with thorough assessment of those risks.

6. CONCLUSION

Molecular dynamics simulations are routinely used to analyse the structure of biomolecules, and to study their functional activities such as ligand binding, complex formation and proton transport. However, the timescales associated with many of these processes are much greater than those achieved via present-day atomistic molecular dynamics. Thus, it is imperative to extend these timescales, which can be done by (i) parallelization across multi- and many-core processors, and (ii) use of approximation algorithms. In the present work, we combine the two techniques. Specifically, we parallelize molecular dynamics simulations using graphical processing units (GPUs), and use the hierarchical charge partitioning (HCP) approximation.

Presence of asynchronous computations in approximation algorithms make these algorithms less than ideal candidates for implementation on the GPU platform. Hence, there is an expectation that the combination of these two techniques would not result in multiplicative speedups, i.e., total application speedup being the product of speedup due to each technique. However, our hybrid approach of the combination of HCP and GPUs does result in nearly multiplicative speedups. For example, for the 25,101-atom nucleosome and 475,500-atom virus capsid, the difference between multiplicative speedup and the actual speedup realized is only *8% and 5%*, respectively, for the single-precision implementation. The near-multiplicative speedup achieved, despite the additional asynchronous computations introduced by the HCP due to the additional divergent branching, is due to two factors: (i) HCP eliminates a number of other divergent branches that would have been executed without the HCP, and (ii) HCP reduces the number of slow global memory accesses. We also show that there were no gross instabilities in the MD simulations due to the use of single-precision. So a single precision implementation may be acceptable where the computational cost of double precision is significantly higher, such as the 10-fold penalty on the C1060 GPU.

The limited amount of GPU memory available (∼4 GB), is an additional challenge that has to be dealt with while implementing applications on GPUs. Unlike typical implementations of MD algorithms, we do not use pre-calculated pair-lists for identifying interacting charges. Instead, we determine, at the point of computation, if two charges should be included in the computations. Eliminating the pair-list significantly reduces memory utilization allowing us to simulate much larger structures, such as the 475,500-atom virus capsid. Thus, we expect our implementation of the HCP implicit solvent algorithm on the GPU to be most useful for the preliminary refinement of large biomolecular structures, which can then be used as input to more accurate simulation methods such as the explicit solvent particle mesh Ewald.

The results shown here are for the distant-dependent-dielectric implicit solvent model. We plan to extend our implementation to use more widely used implicit solvent models such as the generalized Born model.

ACKNOWLEDGMENTS

This work was supported in part by NIH R01 grant GM076121 and by NSF I/UCRC grant IIP-0804155.

7. REFERENCES

[1] G. G. Dodson, D. P. Lane, and C. S. Verma. Molecular simulations of protein dynamics: new windows on mechanisms in biology. *EMBO reports*, 9:144–150, 2008.

[2] M. Karplus and J. Kuriyan. Molecular dynamics and protein function. *Proc Natl Acad Sci U S A*, 102(19):6679–6685, May 2005.

[3] J. L. Klepeis, K. Lindorff-Larsen, R. O. Dror, and D. E. Shaw. Long-timescale molecular dynamics simulations of protein structure and function. *Curr Opin Struct Biol*, 19(2):120–127, April 2009.

[4] C. C. Gruber and J. Pleiss. Systematic benchmarking of large molecular dynamics simulations employing GROMACS on massive multiprocessing facilities. *J. Comput. Chem.*, 32(4):600–606, 2011.

[5] R. Schulz, B. Lindner, L. Petridis, and J. C. Smith. Scaling of Multimillion-Atom Biological Molecular Dynamics Simulation on a Petascale Supercomputer. *J Chem Theory Comput*, 5(10):2798–2808, October 2009.

[6] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, and V. S. Pande. Accelerating molecular dynamic simulation on graphics processing units. *J Comput Chem*, 30(6):864–872, 2009.

[7] T. Darden, D. York, and L. Pedersen. Particle mesh Ewald: An N.log(N) method for Ewald sums in large systems. *J Chem Phys*, 98(12):10089–10092, 1993.

[8] U. Essmann, L. Perera, M. L. Berkowitz, T. Darden, H. Lee, and L. G. Pedersen. A smooth particle mesh Ewald method. *J Chem Phys*, 103(19):8577–8593, 1995.

[9] G. D. Hawkins, C. J. Cramer, and D. G. Truhlar. Parametrized models of aqueous free energies of solvation based on pairwise descreening of solute atomic charges from a dielectric medium. *J. Phys. Chem.*, 100:19824–19836, 1996.

[10] A. Onufriev, D. Bashford, and D. A. Case. Modification of the generalized Born model suitable for macromolecules. *J Phys Chem B*, 104(15):3712–3720, 2000.

[11] J. A. Grant, B. T. Pickup, M. J. Sykes, C. A. Kitchen, and A. Nicholls. The gaussian generalized born model: application to small molecules. *Phys. Chem. Chem. Phys.*, 9(35):4913–4922, 2007.

[12] M. Feig. Kinetics from implicit solvent simulations of biomolecules as a function of viscosity. *J. Chem. Theory Comput.*, 3(5):1734–1748, September 2007.

[13] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *International Conference on Computer Graphics and Interactive Techniques*, pages 777–786. ACM New York, NY, USA, 2004.

[14] Jen-Hsun Huang. Opening Keynote, NVIDIA GTC 2010. http://livesmooth.istreamplanet.com/nvidia100921/.

[15] The Top500 Supercomputer Sites. http://www.top500.org/list/2011/11/100.

[16] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879 –899, May 2008.

[17] D. J. Hardy, J. E. Stone, and K. Schulten. Multilevel summation of electrostatic potentials using graphics processing units. *Parallel Comput*, 35(3):164 – 177, 2009.

[18] J. A. Baker and J. D. Hirst. Molecular Dynamics Simulations Using Graphics Processing Units. *Mol. Inf.*, 30(6-7):498–504, 2011.

[19] N. Ganesan, B. A. Bauer, T. R. Lucas, S. Patel, and M. Taufer. Structural, dynamic, and electrostatic properties of fully hydrated DMPC bilayers from molecular dynamics simulations accelerated with graphical processing units (GPUs). *J. Comput. Chem.*, 32:2958–2973, 2011.

[20] R. Anandakrishnan, T.Scogland, A.T. Fenley, J.C. Gordon, W. Feng, and A.V. Onufriev. Accelerating electrostatic surface potential calculation with multiscale approximation of graphical processing units. *J Mol Graphics Model*, 28:904–910, 2010.

[21] R. Anandakrishnan and A.V. Onufriev. An $n \log n$ approximation based on the natural organization of biomolecules for speeding up the computation of long range interactions. *Journal of Computational Chemistry*, 31(4):691–706, 2010.

[22] The Amber Molecular Dynamics Package, 2011. http://www.ambermd.org.

[23] J. Ramstein and R. Lavery. Energetic coupling between dna bending and base pair opening. *Proc. Natl. Acad. Sci. U. S.Ȧ.*, 85(19):7231–7235, 1988.

[24] J.C. Gordon, A. T. Fenley, and A. Onufriev. An analytical approach to computing biomolecular electrostatic potential. II. Validation and applications. *J Chem Phys*, 129:075102, 2008.

[25] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington. Implementing Molecular Dynamics on Hybrid High Performance Computers - Short Range Forces. *J Comput Phys Comm*, 182:898–911, 2012.

[26] R. Anandakrishnan, M. Daga, and A. V. Onufriev. An n log n Generalized Born Approximation. *J. Chemical Theory and Computation*, 7(3):544–559, March 2011.

[27] H. Wang and E. Nogales. Nucleotide-dependent bending flexibility of tubulin regulates microtubule assembly. *Nature*, 435(7044):911–915, June 2005.

[28] V. Hornak, R. Abel, A. Okur, B. Strockbine, A. Roitberg, and C. Simmerling. Comparison of multiple Amber force fields and development of improved protein backbone parameters. *Proteins*, 65(3):712–725, 2006.

[29] AMBER 11 NVIDIA GPU Acceleration Support, 2011. http://www.ambermd.org/gpus.

[30] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO '08: Proceedings of the 6th IEEE/ACM International Symposium on Code Generation and Optimization*, pages 195–204, New York, NY, USA, 2008. ACM.

[31] A. M. Aji, M. Daga, and W. Feng. CampProf: A Visual Performance Analysis Tool for Memory Bound GPU Kernels. Technical report, Virginia Tech, 2010.

[32] A. M. Aji, M. Daga, and W. Feng. Bounding the Effect of Partition Camping in GPU Kernels. In *ACM International Conference on Computing Frontiers*, Ischia, Italy, May 2011.

[33] Amber (PMEMD) NVIDIA GPU Support - Benchmarks, 2011. http://ambermd.org/gpus/benchmarks.htm#Benchmarks.