Published in final edited form as:
ACM BCB. 2015 September ; 2015: 222–231. doi:10.1145/2808719.2808742.

Accelerated Molecular Mechanical and Solvation Energetics on Multicore CPUs and Manycore GPUs

Deukhyun Cha,
Bluware Inc., Houston, TX, USA.

Alexander Rand,
CD-adapco, Austin, TX, USA.

Qin Zhang,
CGG, Houston, TX, USA

Rezaul A. Chowdhury,
Dept. Computer Science, State University of New York, Stony Brook, NY, USA.

Jesmin Jahan Tithi,
Dept. Computer Science, State University of New York, Stony Brook, NY, USA.

Chandrajit Bajaj
Dept. Computer Science, Institute of Computational, Engineering and Sciences, University of Texas at Austin, Austin, TX, USA.

Abstract

Motivation.—Despite several reported acceleration successes of programmable GPUs (Graphics Processing Units) for molecular modeling and simulation tools, the general focus has been on fast computation with small molecules. This was primarily due to the limited memory size on the GPU. Moreover simultaneous use of CPU and GPU cores for a single kernel execution – a necessity for achieving high parallelism – has also not been fully considered.

Results.—We present fast computation methods for molecular mechanical (Lennard-Jones and Coulombic) and generalized Born solvation energetics which run on commodity multicore CPUs and manycore GPUs. The key idea is to trade off accuracy of pairwise, long-range atomistic energetics for higher speed of execution. A simple yet efficient CUDA kernel for GPU acceleration is presented which ensures high arithmetic intensity and memory efficiency. Our CUDA kernel uses a cache-friendly, recursive and linear-space octree data structure to handle very large molecular structures with up to several million atoms. Based on this CUDA kernel, we present a hybrid method which simultaneously exploits both CPU and GPU cores to provide the best performance based on selected parameters of the approximation scheme. Our CUDA kernels achieve more than two orders of magnitude speedup over serial computation for many of the molecular energetics terms. The hybrid method is shown to be able to achieve the best performance for all values of the approximation parameter.

deukhyun.cha@bluware.com.

Availability.—The source code and binaries are freely available as *PMEOPA* (Parallel Molecular Energetic using Octree Pairwise Approximation) and downloadable from http://cvcweb.ices.utexas.edu/software.

Categories and Subject Descriptors

I.6 [**SIMULATION AND MODELING**]: Numerical Algorithms and Problems; D.1.3 [**Concurrent Programming**]:[Applications]; Algorithm; Graphics Processing Units; Parallel Programming; Molecular Dynamics; Computational Chemistry; BioPhysics

1. INTRODUCTION

The aim of this paper is to accelerate the computations of configuration-dependent free energy terms of very large molecules which need to be estimated repeatedly for computational docking, energy minimization and molecular dynamics (MD) simulations. We achieve this by simultaneously utilizing multicore CPUs and manycore GPUs. The model of molecular free energy consists of a molecular mechanical portion E_{MM} and a solvation energy part E_{sol}. The mechanical energy has been empirically parameterized into three parts: $E_{\mathrm{MM}} = E_{\mathrm{bond}}+E_{\mathrm{vdw}}+E_{\mathrm{coul}}$. The bonded component E_{bond} is typically computed using efficient lookup tables based on expected bond lengths and angles, and thus is not particularly computationally challenging. The E_{vdw} and E_{coul} terms, which represent pairwise atomic energy due to van der Waals and electrostatic interactions, respectively, are given by the following pairwise summations:

$$E_{\mathrm{vdw}} = \sum_{i} \sum_{j>i} \left(\frac{a_{ij}}{r_{ij}^{12}} - \frac{b_{ij}}{r_{ij}^{6}} \right) \text{ and } E_{\mathrm{coul}} = \sum_{i} \sum_{j>i} \frac{q_i q_j}{\epsilon(r_{ij}) r_{ij}},$$

where r_{ij} is the distance between two given atoms indexed i and j, a_{ij} and b_{ij} are coefficients based on atom types, q_i and q_j are Columbic partial charges, and $\epsilon(r_{ij})$ is a distance dependent dielectric constant.

The solvation energy model can be further decomposed into non-polar and polar components [11, 16, 23, 43, 45]: $E_{\mathrm{sol}} = E_{\mathrm{nonpol}} + E_{\mathrm{pol}}$. Effective models for the non-polar energy E_{nonpol} have been empirically developed [10, 11, 14, 15, 25, 33, 35, 38, 52]. Although our method can also speed up the dispersive non-polar component of the interaction, we focus on polarization energy which has the following form:

$$E_{\mathrm{pol}} = \frac{1}{2} \int \left(\phi(\mathrm{x}) - \phi_{\mathrm{gas}} \right) \rho(\mathrm{x}) \mathrm{dx}, \tag{1}$$

where ρ is the charge density, ϕ and ϕ_{gas} are the electrostatic potential for the molecule in solution and in gas, respectively. Two models are usually employed to compute this energy: (a) modeling the potential with the *Poisson-Boltzmann (PB) equation* [44], or (b) approximating the energy with the *Generalized Born (GB) model* [47]. The following GB formula has been shown to give an effective approximation of polarization energy for large systems [47]:

$$E_{\text{pol}} = -\frac{1}{2}\left(1 - \frac{1}{\varepsilon_{\text{solv}}}\right)\sum_{ij} q_i q_j \left[r_{ij}^2 + R_i R_j \exp\left(-\frac{r_{ij}^2}{4R_i R_j}\right)\right]^{-\frac{1}{2}},$$

where $\varepsilon_{\text{solv}}$ is the dielectric constant of molecules in solution and R_i is the effective Born radius of atom i, based on the assumption that the electric displacement is in the Columbic form [2]. Several modifications of the GB model have been proposed [31, 30, 27, 46], and we have used Grycuk's GBr^6 formulation [20, 51], i.e., $R_i^{-1} = \left(\frac{3}{4\pi}\int_{\text{ex}}|\mathbf{r} - \mathbf{x}_i|^{-6}\text{d}\mathbf{r}\right)^{1/3}$, in our computation.

Summary of Prior Work.

Naïve Approach: Naïve computation of E_{vdw}, E_{col} and E_{pol} (pairwise interactions) requires $\mathcal{O}(N^2)$ arithmetic operations, and also $\Omega(N^2)$ memory for the intermediate Born radii computation in E_{pol}, where N is the number of atoms in the system. Many techniques have been developed to accelerate these computations, such as direct N^2 (or $N^2/2$) computation method by exploiting shared memory via tile-based decomposition of the N^2 matrix [9, 12, 17] or cell list method [49]. Here we briefly review GPU-based acceleration schemes, neighbor list method [37, 58, 36] and fast multipole methods [19, 3], which are commonly used in molecular simulations and mostly related to our current work.

Solutions using neighbor lists: The neighbor list of an atom contains only atoms within a given cutoff distance from that atom. The neighbor lists of all atoms can be represented as a matrix. Due to the irregularity of atomic distributions, the number of atoms in distinct neighbor lists may vary. Several optimization schemes have been used for mapping neighbor list structures on GPUs. In [57], 2D texture has been exploited for storing the neighbor list to utilize the texture caches on GPUs. When the construction is performed on GPU, the neighbor list is stored in device memory [34, 1, 56, 40]. In most algorithms, each thread takes care of one atom by looping over all of its neighbors to relieve the burden of device memory accesses by using registers for repeatedly visited data. It has been shown in [54] that this approach can enhance performance in spite of reducing thread occupancy. To increase the locality of atomic data accessed from the neighbor list, atoms are often spatially sorted using space filling curves [1].

Neighbor list vs. octree: Computation of pairwise interactions using the neighbor list scheme takes time linear in N and up to cubic in distance cutoff d, and it requires another $\mathcal{O}(N^2)$ distance computations for its construction. On the other hand, computation of pairwise interactions using octree-based methods take time linear or nearly linear in N and is independent of any distance cutoff, and octree construction takes $\mathcal{O}(N \log N)$ time [5, 4]. Moreover, the neighbor list requires $\Theta(N^2)$ memory in the worst case. Our parallel octree scheme requires only $\mathcal{O}(N)$ space, since it only considers leaf nodes of the octree as explained in Section 2. This enables our method to handle molecules and macromolecules with millions of atoms.

Fast-multipole like approaches: The fast multipole method (FMM) [39] approximates $\mathcal{O}(N^2)$ summations in $\mathcal{O}(N)$ or $\mathcal{O}(M\text{n}^{\alpha}N)$ arithmetic operations by using a near-far distance decomposition scheme. Increasing arithmetic intensity and balancing workload are crucial in FMM because of its intensive memory accesses of the coefficient matrices and vectors as well as the computation on nodes which are of irregular sizes. Most work on GPU-based FMM map each target node to a single CUDA thread block and store source variables and results into the shared memory for a significant reduction of the numbers of device memory accesses [48, 21, 60, 26, 22, 59]. The number of elements in each node is further matched to the number of threads in the thread block [48], or the number of thread blocks is fixed [22]. Darve *et al.* [8] cluster the nodes that use the same coefficient matrix to ensure coalesced memory access of multipole coefficients. Once the multipole coefficients are copied into the shared memory, each thread can access them without bank conflicts. In addition, Zhao and Feng [61] sort non-zero entries of the coefficient matrices and generate clusters which are run by separate kernels concurrently. Atoms are stored in Z-Morton layout to maximize data locality when the computation is performed on CPU [26, 8] or on GPU [24]. More recently, an adaptive FMM has been implemented on GPU [18].

Uniqueness of our approach: In spite of the similarities between our octree-based ϵ-approximation technique and the FMM approximation, there are certain differences in the optimization of our CUDA kernel. Our method uses Greengard-Rokhlin type near-far approximation [19] which dynamically determines and evaluates operations on different domains such as atom-atom, node-node, and atom-node while FMM separates those operations to several computational layers. Implementing an efficient CUDA kernel for this method brings new challenges on balancing workload among these computationally variable domains and quick fetching of data, which we address in our kernel design. There are previous studies on accelerating neighbor list based [41, 55] and FMM based [26, 24] applications that leverage heterogeneous computational resources exhaustively and asynchronously as their target systems. Although our aim is the same as theirs, i.e., achieving optimal performance by fully utilizing available resources in a given system, our approach is different from theirs since we focus on exploiting heterogeneous resources for computational kernels that are very simple but hardly separable. In our method, the computational characteristics and entire workload vary significantly based on the approximation parameter ϵ (i.e., desired error bound). Hence our combined use of CPU and GPU cores yields dissimilar hybrid CILK and CUDA implementations to ensure desirable performance on both of these architectures.

Our Contributions.

The major contributions of this paper are as follows:

- Acceleration on GPU and CPU and speed-accuracy tradeoff: In this paper, we present parallel methods that run on commodity multicore CPUs and manycores GPUs to accelerate and trade off speed for accuracy, i.e., ϵ-approximation [5] of pairwise, long-range atomistic energetics. We first develop a pure GPU-oriented method using CUDA programming model, and then extend it to a hybrid method that simultaneously exploits both CPU and GPU cores.

- Efficient use of linear-space octree: Our CUDA kernel ensures efficiency with highly balanced computational load on each thread, zero communication between them, and optimized use of on-chip memories. The space-efficiency and simplicity of the octree data structure enables us to handle extremely large molecular structures. The entire data for a molecular structure containing several million atoms can be loaded into 1.5 GB device memory and the computation can be performed with one CUDA kernel invocation. Furthermore, an optimal data decomposition is used for minimal CUDA kernel invocation when the size of the data for a given structure exceeds the size of the device memory.
- Simultaneous and selective use of CPUs and GPUs: Since the computational characteristics of our ϵ-approximation scheme changes based on the value of parameter ϵ, we develop a hybrid method that exploits both CPU and GPU cores in two different ways, i.e., simultaneously and selectively, according to the ϵ value to produce the best performance on a given system.
- Performance demonstration on modern CPUs and GPUs and comparison with Amber: We demonstrated empirically that the proposed methods can achieve significant performance enhancements and handle large structures for selected computations in molecular mechanical (Lennard-Johns and Columbic) and generalized Born solvation energetics. For instance, Lennard-Johns (short range) and generalized Born (long range) energetics were able to achieve two orders of magnitude speedups over their respective serial versions for molecules containing up to a couple of millions of atoms. We also present results showing that the proposed method computes generalized Born energy faster than the Amber GPU versions [6, 7] while producing less error in the energy values w.r.t. Amber, considering the GB equation in [47] as a baseline.

A brief description of the octree-based ϵ-approximation scheme for estimating pairwise atomic energetics with speed-accuracy tradeoff followed by details of our parallelization including the CUDA kernel and the hybrid CILK and CUDA methods are given in Section 2. Section 3 describes specific implementations of energetic terms. Performance results on the energetics approximation on multicore CPUs and GPUs are given in Section 4. Finally, Section 5 concludes the paper with some remarks.

2. METHOD

In this section we describe the algorithmic techniques we use to compute the molecular energetic terms on CPUs and GPUs. Our methods based on pairwise ϵ-approximation that uses a Greengard-Rokhlin type near and far decomposition of 3D data points [29] with an *octree* data structure as described in [5, 50].

2.1 ϵ-Approximation

Algorithm 1

Compute energy $E(n_i, n_j)$ between nodes n_i and n_j using the Octree-based Pairwise Approximation (OPA) algorithm.

```
1: procedure OPA($n_i$, $n_j$)
2:     Compute distance $d = d(n_i, n_j)$ between node pair $n_i, n_j$.
3:     if IsFar($d, n_i, n_j, \epsilon$) & IsSmall($n_i$) & IsSmall($n_j$) then
4:         Approximate $E(n_i, n_j)$ with a pseudo-atomic interaction
5:     else
6:         if IsLeaf($n_i$) & IsLeaf($n_j$) then
7:             Compute atomic pairwise $n_i, n_j$.
8:         else if IsLeaf($n_i$) then
9:             for all child $c_j$ of node $n_j$ do
10:                OPA($n_i, c_j$)
11:            end for
12:        else if IsLeaf($n_j$) then
13:            for all child $c_i$ of node $n_i$ do
14:                OPA($c_i, n_j$)
15:            end for
16:        else
17:            for all child $c_i$ of node $n_i$ do
18:                for all child $c_j$ of node $n_j$ do
19:                    OPA($c_i, c_j$)
20:                end for
21:            end for
22:        end if
23:    end if
24: end procedure
```

First we briefly sketch the Octree Pairwise Approximation algorithm (OPA) proposed in [5] and call it Algorithm 1. We used this Algorithm as our base algorithm. To compute the interaction energy $E(\mathscr{M}_X, \mathscr{M}_Y)$ between two molecules $\mathscr{M}_X$ and $\mathscr{M}_Y$($\mathscr{M}_Y = \mathscr{M}_X$ for the case of internal interactions in a single molecule), we construct two octrees $T_{\mathscr{M}_X}$ and $T_{\mathscr{M}_Y}$ for $\mathscr{M}_X$ and $\mathscr{M}_Y$, respectively, and perform a simultaneous recursive traversal of both trees starting from their respective root nodes. For each pair of nodes $(n_i, n_j)(n_i \in T_{\mathscr{M}_X}, n_j \in T_{\mathscr{M}_Y})$ examined during the traversal, the computation can be split into three cases based on a user-defined approximation parameter ϵ. If both n_i and n_j are leaf nodes (lines 6–7 in Algorithm 1), interaction between each atom of n_i and every atom of n_j is computed exactly. If nodes n_i and n_j are far from each other and small enough (lines 3–4 in Algorithm 1), the pairwise atomic interactions are approximated by a single interaction between two pseudo atoms defined for the two nodes, where the pseudo atom corresponding to a node is centered at geometric center of the atoms inside that node and has the smallest radius large enough to contain all atoms of the node. If neither of the two cases above holds, we subdivide the octree nodes n_i and/or n_j and examine all the new child node pairs recursively (lines 8–22 in Algorithm 1). The main benefit of this method is that we can trade off accuracy for computational speed through the user-defined parameter ϵ [5].

2.2 Parallelization

To develop an efficient parallel algorithm for a heterogeneous CPU + GPU platform, we need to develop efficient parallel algorithms for CPU and GPU separately to leverage architecture specific properties with sufficient optimizations, and then hybridize them with an appropriate decomposition of the overall problem. Since GPUs have much higher floating point arithmetic intensity per clock cycle and requires additional care for performance, we focus on optimizing the GPU-oriented parallel program and then hybridizing it with CPU threads. We chose to use Cilk++ and CUDA programming tools and run-time libraries to implement our parallel algorithms.

Algorithm 2

Compute energy $E(l_i, l_j)$ between leaf nodes l_i and l_j using the Leaf Octree Pairwise Approximation (LOPA) algorithm.

```
1: procedure LOPA
2:     Compute leaf pair indices $i$ and $j$.
3:     Get node info $l_i$ and $l_j$ and compute distance $d = d(l_i, l_j)$.
4:     if IsFar($d, l_i, l_j, \epsilon$) & IsSmall($l_i$) & IsSmall($l_j$) then
5:         Approximate $E(l_i, l_j)$ with a pseudo-atomic interaction
6:     else
7:         Compute atomic pairwise interaction $E(l_i, l_j)$.
8:     end if
9: end procedure
```

2.2.1 Parallelizing using Cilk++ on multicores—

The recursive divide-and-conquer schemes in [5] are quite suitable for multithreaded execution on multicore CPUs. We implemented a parallel ϵ-approximation algorithm using Intel's Cilk++ [32] that uses shared-memory parallelism as explained in [50]. When two octree nodes containing atoms are close to each other at any stage of the traversal according to the given ϵ, each child of a node is paired with each child of the other, thereby allowing a quick spawn of multiple independent parallel threads and an easy recursive implementation. We have allocated explicit buffers to accumulate partial sums produced by each thread, since it is often more efficient than using a *reducer* [13] for a small number of partial sums.

2.2.2 Parallelizing using CUDA on GPUs—

Our CUDA kernel has been designed to ensure the following properties which are necessary for efficiency.

- capability to handle very large atomistic structures (enabling efficient memory load)
- maximal occupancy of resident threads on GPU (hiding memory access latency through warp switching)

- minimal device memory access overheads (enabling compute-intensive kernel)
- control diversity of the threads (balancing workload)
- smaller or equal approximation error w.r.t. the original scheme

Basic algorithm.: Our simple CUDA kernel computes an interaction between a given pair of leaf nodes, using a octree based near-far pairwise approximation algorithm (OPA) presented in [5]. We call our scheme the Leaf based Octree Pairwise Approximation (LOPA) method. Once indices of two leaves are calculated from the thread ID, the pseudo-atomic center and radius of each node are loaded from the device memory into the registers to perform a near-far test. If the test result is *far*, pseudo-atomic approximations are performed for the given leaf pair. Otherwise all pairwise interactions between the atoms of the two leaf nodes are explicitly calculated. The algorithm for this CUDA kernel is shown in Algorithm 2.

Data layout and thread efficiency.: To reduce device memory access overheads, we exploit the on-chip memories by arranging the leaf node pairs as shown in Figure 1. The leaf pairs (i, j) are arranged so that the same leaf index i is accessed by all threads in the same block. Next the data of atoms and pseudo atoms of the leaf i is loaded into the shared memory once, and are accessed by all threads in the same block during the computation. The loop of i runs inside the loop of j to maximize the use of the data in the shared memory. By shifting the index i in the loop using the thread ID, bank conflict can be reduced for this shared memory access. To prevent significant device-to-host data transfer, each thread stores its partial sum in the shared memory, and the values are accumulated by a single thread after all threads in the same block finish their computation. Then a single value per thread block is stored into device memory and the entire buffer for all thread blocks is copied into the host memory to perform the final accumulation.

When each leaf contains k atoms, $(N/k)^2$ threads are spawned, and each thread computes up to k^2 loop iterations for the case of *near*. If k is small, our method spawns many computationally light CUDA threads and reduces inefficiency caused by divergence of near and far cases in the warp. The size of required shared memory is also decided by k, and the number of active warps in each SM, i.e., the thread occupancy, is accordingly changed. However, too small a value of k can eliminate the computational benefits of pseudo atomic approximations. Finding an optimal k and space requirement will be discussed later in Section 3.

2.2.3 Parallelizing using hybrid method—The computational expense of our octree-based approximation scheme changes significantly with the approximation parameter ϵ, because the total number of node pairs quickly decreases as ϵ value gets bigger. Therefore we developed a hybrid method that uses both CPU and GPU processors in simultaneous and selective ways based on the computational characteristics determined by the ϵ parameter. The criteria for an efficient hybrid method on a heterogeneous CPU + GPU node can be:

- maximal simultaneous use of both processors
- minimal data transfer between CPU and GPU threads

- optimal problem decomposition which optimally matches the characteristics of each processor

We first present three variants (A, B, C) of a hybrid method demonstrating how optimized hybrid methods can be developed. Based on our experience with these three methods we finally develop method (D).

Method A, shown in Figure 2(a), computes LOPA on both CPU and GPU cores by allocating user-defined proportions of leaf pairs $(0 \leq P_{Cilk} \leq 1)$ to Cilk threads and the rest to CUDA threads. In this method, a user-defined number of Cilk threads is spawned, and one of them invokes the CUDA kernel while others compute LOPA for the given leaf pairs. Method A brings performance enhancement by distributing the work loads to CPU cores which were in idle state for the pure GPU-oriented method. Method B, shown in Figure 2(b), performs the near-far test, and computes pseudo atomic approximation for the *far* case on the CPU using Cilk threads, and exact atomic interactions for the *near* case on the GPU. To do so, this method stores indices of *near* leaf pairs during the computation on CPU and copies it to device memory. In this way, we can fully exploit OPA on both processors by decomposing it into memory access and conditional branching intensive part and compute-intensive part. Method C modifies method B by performing the near-far test and pseudo atomic approximation only before a user-defined octree depth d, and performing LOPA for all the remaining leaf pairs on the GPU. This method reduces the number of near-far tests in method B which is a waste when most computation are performed inside leaf pairs for small values of ϵ. If we control the parameter d with respect to inverse proportion of ϵ, then the deeper octants will be traversed only when shorter distance is allowed to be approximated.

We have tested our three methods for the Lennard-Jones potential computation where OPA performs very efficiently while producing relatively small error even for large values of ϵ. To see how these methods work, we use a fairly large molecular structure (PDBID: 1N2C, 39,946 atoms) so that the thread spawning time becomes negligible compared to the actual computational load. Figure 3 shows performance of A, B, and C hybrid methods on 2 Intel Xeon E5640 CPUs and 1 Nvidia Geforce GTX 580 GPU for molecule 1N2C. In the hybrid method A, eight Cilk threads are used and one of them is allocated for CUDA kernel invocation. Figure 3(a) shows that the computation time of method A decreases until 20% of the leaf pairs are allocated to Cilk threads for all error ranges. Clearly, the parameter P_{Cilk} needs to be set to make CPU and GPU run-times equal so that an optimal performance of method A can be achieved. We have found that when $P_{Cilk} = 0.2$, method A achieve a speedup of 1.16× w.r.t. the pure GPU (LOPA) version.

Figure 3(b) shows that even though methods B and C both run faster than GPU-oriented LOPA for large values of ϵ, they are slightly slower than OPA using only Cilk threads (i.e., pure CPU version). Since the number of leaf pairs computed on GPU is very small, computation time is dominated by overheads such as CUDA kernel invocation, host-to-device copy of leaf pair indices, and additional device memory accesses in CUDA thread for the leaf indices. On the other hand, for small ϵ values, most computations fall into leaf pairs and octree traversal with near-far tests can be a total waste. Method B suffers from overheads of traversing the octrees for small values of ϵ where most of the near-far tests end up being *near*, and as a result a large number of leaf pairs are sent to the GPU. Method C gets worse for small ϵ, since it stops near-far tests at a high level of the octree and fully traverses it without early termination to find remaining leaf pairs for the LOPA on GPU. Therefore, running pure LOPA on GPU instead is much faster for those cases.

Based on all these observations, we developed method D as shown in Figure 4(a). Since the OPA using Cilk threads is the most efficient for large values of ϵ, we select it for ϵ values greater than a user-defined parameter ϵ_{Cilk}. Otherwise we use method A which utilizes both CPU and GPU simultaneously by allocating a certain proportion of leaf pairs to CPU threads with a parameter P_{Cilk}. This method requires no communication between the two processors during the computation. Figure 4(b) shows that the best performance graph is obtained by method D for the LJ potential computation of the molecular structure 1N2C. The best performance is obtained for the entire range of values of by setting the parameters $P_{Cilk}(=0.2)$ and $\epsilon_{Cilk}(=0.7)$ properly. In general, optimal values for these parameters depend on the system's spec and our experiments for a large number of molecular structures indicates that a reliable ϵ_{Cilk} can be found for a given system fairly easily.

3. IMPLEMENTATION

In this section we provide some implementation details especially for our CUDA algorithm explained in Section 2.2.

LJ Coefficients.

We use constant memory to store the LJ coefficients precomputed for each combination of different atom types. Although there is a limitation on the number of atom types that can fully exploit the constant cache, this allows us to access the coefficients for both the atomic and pseudo atomic computations very efficiently. Without losing too much accuracy, atoms having similar chemical properties can be grouped to reduce the number of coefficients for a given constant cache.

GB Energy Computation.

Unlike other energetic terms, GB energy computation produces a pairwise loop during the pseudo atomic computation due to an approximation of the Born radii into $\mathcal{O}(\log N)$ groups based on the ϵ value where N is the number of atoms [5, 50]. This makes it difficult to utilize limited GPU caches. Moreover the accuracy of the GB energy computation is very sensitive to the Born radii values and it requires a sufficient number of samples of the Born radii. We found that sometimes exact computations involving only pairwise atomic interactions without any Born radii approximation runs even slightly faster than the original LOPA because of the elimination of the conditional branching of near-far cases. Approximation can make the Born radii values more erroneous in this case, as the GB energy computation achieves the accuracy of the Naïve computation because of using leaf pairs only. For the Born radii computation, a special case of calculating the value on each atom, we allocate all leaf pairs (i,j) with the same i value into each CUDA thread block. The Born radii values for all atoms in leaf i are evaluated using threads in the thread block and accumulated in the shared memory. After the computation, the final values are copied to device memory. The atomic and pseudo atomic information in leaf *i* is stored in the shared memory, and the CUDA kernel performs LOPA with an additional outer loop for more than one *j*, if needed.

Optimal *k*.

Even though finding an optimal k is not trivial, it can be bounded for the theoretical maximum thread occupancy in each energetic computation. As explained before, k is one of the parameters that decides the size of the shared memory used in the thread block. For example, each block requires $S_B(= 20k + 4T_B)$ (T_B stands for *number of threads in a thread block*) bytes of the shared memory to compute the generalized Born energy. If the GPU supports compute capability (CC) 1.3, then it needs to satisfy $16\text{KB}/S_B \geq 8$ for thread occupancy 1 while $T_B \times 8 \leq 1024$. If we set T_B to 128, then k needs to be less than 76. Although the choice of value for k can affect the performance of the CUDA kernel in various ways, one can quickly test the program with a few sample values of k under the given bound to find a value close to the optimal. Table 1 shows how different choices of k changes performance of GB energy computation when input is a molecule with 3,924 atoms. In this example, $k = 32$ gives the best warp execution efficiency while issuing minimum number of instructions.

Space Requirement.

As one of the major advantages, the proposed CUDA kernel requires fairly small and controllable amount of device memory, which makes it possible to handle large molecular structures. For given molecules $\mathscr{M}_X$ and $\mathscr{M}_Y$ having $N_{\mathscr{M}_X}$ and $N_{\mathscr{M}_Y}$ atoms, respectively, the octrees have $N_{\mathscr{M}_X}/k = N_{\mathscr{L}_X}$ and $N_{\mathscr{M}_X}/k = N_{\mathscr{L}_Y}$ leaves, respectively, assuming each leaf contains k atoms. Therefore, if S_Z is the byte size of an element (atom or node) in Z, we need to allocate $S_{\mathscr{M}_X}N_{\mathscr{M}_X} + S_{\mathscr{M}_Y}N_{\mathscr{M}_Y} + S_{\mathscr{L}_X}N_{\mathscr{L}_X} + S_{\mathscr{L}_Y}N_{\mathscr{L}_Y}$ bytes of device memory. In addition to this, our method also allocates an intermediate buffer for each partial sum of the thread block. It requires $4\left\lceil \left(N_{\mathscr{L}_X}N_{\mathscr{L}_Y}/T_B\right)/N_{\text{Call}} \right\rceil$ bytes if we divide node pairs into N_{Call} chunks and invoke the CUDA kernel N_{Call} times. Suppose we compute GB energy for a single molecule with the Born radius given as an input and have S_{DRAM} bytes of device memory. Then $S_{\mathscr{M}_X}N_{\mathscr{M}_X} + S_{\mathscr{L}_X}N_{\mathscr{L}_X} + 4\left\lceil \left(N_{\mathscr{L}_X}{}^2/T_B\right)/N_{\text{Call}} \right\rceil \leq S_{\text{DRAM}}$. This relation can be further generalized if we split the set of whole leaves into chunks with each chunk containing $N_C (\leq N_{\mathscr{L}_X})$ leaves for the case of extremely large molecular structures that do not fit into the given memory. Since $N_{\mathscr{M}_X} = kN_{\mathscr{L}_X}$ and $N_{\text{Call}} = \lceil N_{\mathscr{L}_X}/N_C \rceil$, the relation turns out to be $2kS_{\mathscr{M}_X}N_C + 2S_{\mathscr{L}_X}N_C + 4\left\lceil \left(N_C{}^2/T_B\right)/\lceil N_{\mathscr{L}_X}/N_C \rceil \right\rceil \leq S_{\text{DRAM}}$ where the first two terms are scaled by 2 to take into account pairing different leaf chunks. By finding the largest value of N_C which satisfies this relation, our CUDA kernel can perform efficiently for molecular structures of any size using minimal kernel invocation (N_{Call}) with a given size of the device memory. Our **LOPA is able to handle up to** 9.5 **millions of atoms** in GB energy computation with $k = 32$ and $N_{\text{Call}} = 1$ for 1536MB device memory as available in Nvidia Geforce GTX 580 GPU used in our experiments.

Although our kernel design satisfies important optimization properties needed for GPU implementation, it computes always constant $(N/k)^2$ leaf pairs, regardless of the parameter ϵ. On the other hand, $\sum_{i=1}^{\log 8(N/k)}\left\{min\left(8^{i-1}, N\right) \times min\left(8/\epsilon^3, 8^{i-1}, N\right) \times 8^2\right\}$ node pairs will be considered, if the OPA is fully implemented. Even though one can easily see that the number of node pairs in OPA will quickly decrease as ϵ value gets bigger, the implementation is not suitable for the GPU architecture because of the extensive device memory accesses. OPA produces up to 64 additional node pairs when a given node pair is determined to be *near*. To balance workload, therefore, these new pairs of nodes need to be reallocated to active threads properly. This requires frequent and even serialized device memory accesses. These pros and cons in both methods suggest a hybrid approach similar to what we have used in this paper.

4. EXPERIMENTAL RESULTS

In this section, we summarize the performance results of our CUDA implementation as well as the hybrid approaches. Experiments were performed for the Coulomb potential (CP), Lennard-Jones potential (LJP), and generalized Born polarization (GBP) energy computations. The generalized Born energy calculation includes the Born radius computation. For all the experiments shown in this section, the molecular complexes in bounded state were selected from ZDock benchmark 2.0 where the number of atoms in each complex lies between 6,000 and 32,000. We used state-of-the-art multicore CPUs and manycore GPUs as listed in Table 3 to conduct these experiments.

In Table 2, we summarize the performance results of our CUDA and Cilk+CUDA implementations. The number of threads in a thread block, T_B, was set to 256. This experiment was performed on GTX580. Maximum and average speedup factors over Naïve and average computational time per atom are shown for CP, LJP, and GBP energy computations in 4th, 5th, and 6th rows, respectively. All CUDA timings include device memory allocation, device-to-host data copy, texture binding, kernel run, host-to-device data copy, and final accumulation. Speedup factors over conventional serial implementation (Naïve) for these three energetics terms are close to two orders of magnitude except for CP which requires execution of more atomic pairs than other terms to bound the same error.

4.1 GB Energy

In this subsection we present detailed performance analysis of our methods for the generalized Born energy computation. Figure 5 shows the runtime performance of GBP-CuLOPA implementation on large molecular structures with number of atoms from half a million to nearly two millions. Because of the complex shapes of the molecules a large number of quadrature points are also required. Figure 5(b) shows a comparison of GBP-CuLOPA with GBP-Cilk, Amber 12 and Amber 14 for the Cucumber Mosaic Virus Capsid (CMVC) having 509,640 atoms with 1,929,128 quadrature points. Although relatively large errors were produced during the approximation of Born radii for this large molecular structure, GBP-CuLOPA brought a significant benefit in error-speedup tradeoff (less than 0.006% error versus more than 660× speedup) compared to GBP-Cilk. GBP-Cilk was run on 2 Intel Xeon-E2680 SandyBridge machines. GBP-CuLOPA and Amber 12-k20 were run on

Kepler-20, Amber 14-GTX780 was run on GTX 780 and Amber 14-k80 (0.82GHz) was run on Kepler 80 GPUs. Since Amber GPU versions do not directly report the GB energy time, we consider 98% of the non-bonded energy time reported by Amber as the GB time as suggested by an Amber GPU developer [53]. Note that, Amber GPU timings do not include data uploading/downloading time to/from GPUs, whereas our timings include all of them.

Figure 5(c) shows a performance table of GBP-CuLOPA for the large molecular structures shown in Figure 5(a). Each number within the parenthesis in the second and third columns of the table indicates the number of leaf nodes in the corresponding octree data structure. The number within parenthesis in the speedup column represents relative 'rate of computation time increase'/'square rate of atom increase' based on the values for the molecule in the first row. This table shows that our CUDA implementation scales almost the same rate as the square rate of atomic increment which is quite reasonable in this N^2 pairwise computation. The speedup over Naïve decreases because the Naïve scales better as the #atoms gets larger. The device memory required in the entire GBP-CuLOPA computation is $\left(20N_{\mathscr{M}_X} + 20N_{\mathscr{L}_X} + 28N_{\mathscr{M}_Y} + 36N_{\mathscr{L}_Y} + 4N_{\mathscr{L}_X}N_{\mathscr{L}_Y}/T_B\right)$ bytes as explained in Section 3

Overall, the performance of GBP-CuLOPA nicely scales with the number of atoms in the molecule while producing only 0.006% error and using fairly small amount of device memory which can be further reduced through multiple CUDA kernel invocations. It demonstrates an ability of efficient handling of huge systems which distinguishes CuLOPA from other GPU-based methods. Two other popular MD packages, Gromacs [42] and NAMD [28] programs crash for large biomolecular complexes whereas our implementations are able to successfully compute the energy terms. However, these MD packages are in general highly optimized for moderate sized molecules. They nevertheless lack a GPU-based implicit solvent/GB energy function or occasionally produced invalid results for some large structures in our experiments which made a comparison with those difficult.

Figure 6 shows performance and error comparison of various methods for the generalized Born polarization energy computation. GBP-CuLOPA achieves average speedup factors of 252 and 184 on Kepler 20 and Geforce GTX 580 GPUs, respectively, while producing small errors. GBP-CuLOPA also runs faster and has less errors (w.r.t. Naïve) than Amber 12 GPU version. As explained in Section 3, spatial domain decomposition using octrees nicely balances workload of each thread and controls on-chip memory usage. Therefore, GBP-CuLOPA can compute $\mathscr{O}(N^2)$ atomic pairwise interactions efficiently. Furthermore, the performance scales reasonably from Geforce GTX 580 to Kepler 20 which has about twice the number of cores with slightly higher clock speed and larger on-chip memories.

4.2 LJ and Columb Energy

Figure 7 shows the performance of our CUDA implementation (CuLOPA) for LJP (Lennard-Jones Potential) and CP (Coulomb Potential) energy computations. The CUDA timing covers data-loading and kernel execution as before. Data-loading includes device memory allocation, host-to-device memory copy, and texture binding. Kernel execution includes actual kernel run, device-to-host memory copy for the result buffer, and final accumulation of the result. For the sake of comparison, maximum and average speedups of CP/LJP-CuLOPA over the fully atomic N^2 computation (Naïve) and the OPA computation using 12 Cilk threads (12Cilk) are shown. Timing results were measured for the largest 50 protein complexes in ZDock benchmark 2.0.

Figure 7 shows that CP-CuLOPA runs over 27.5× and 4× faster on the average than CP-Naïve and CP-12Cilk, respectively, for all three error bounds presented. LJP-CuLOPA, which is more efficient in error-speed tradeoff due to its quickly decaying distance-based kernel and complex arithmetic operations, achieved average speedup factors of 101× and 1.2× over LJP-Naïve and LJP-12Cilk, respectively, while the errors were less than 0.5%. Although LJP-CuLOPA runs two orders of magnitude faster than LJP-Naïve for larger errors, it runs slower than LJP-12Cilk using OPA. The OPA for LJP significantly reduces the number of node pairs computed by quickly terminating a large number of them in intermediate levels. Then the computation can get even more efficient since near atom pairs can be highly localized. On the other hand, for a given molecule LJP-CuLOPA always spawns the same number of threads regardless of the value of ϵ. Therefore, our hybrid method is necessary for LJP-CuLOPA to overcome this inefficiency and enhance its performance further.

Figure 8 shows speedup factors achieved by two hybrid methods for computing Lennard-Jones potential on the same set of molecular complexes used in Figure 7. Figure 8(a) shows the speedup factors of Hybrid A over CuLOPA as P_{Cilk} varies, while Figure 8(b) shows how Hybrid D speeds up over Naïve when P_{Cilk} and ϵ_{Cilk} are kept fixed at 0.2 and 0.7, respectively, but is varied. Hybrid D achieves optimal performance, i.e., faster than both LJP-CuLOPA and LJP-Cilk, for all values of by fully utilizing the given cores in the system for $\epsilon < 0.7$ and selecting OPA with Cilk threads as an optimal algorithm when $\epsilon \geq 0.7$. In our hybrid method, reliability in choo sing values for two parameters (P_{Cilk} and ϵ_{Cilk}) is crucial.

In Hybrid A, 90% of the complexes exhibited speedup until P_{Cilk} reached 0.2. Although 40% of them started to lose performance at $P_{Cilk} = 0.3$, the average speedup peaks at $P_{Cilk} = 0.2$ (Figure 8(a)) since those 40% of the complexes are relatively large and speedup is much higher than the smaller ones. Therefore, choosing P_{Cilk} with average speedup is reasonable to achieve a speedup close to optimal. For the parameter ϵ_{Cilk}, LJP-CuLOPA ran faster than LJP-12Cilk for 66% of the complexes with ϵ values less than 0.7. However, the rates at which performance degrades for the remaining 34% of the complexes are relatively small especially for the large structures, so it generally works well. Therefore, a small set of complexes representing general range of sizes of the input structures should be sufficient to find close to optimal values for those parameters for a given system.

5. CONCLUSION

This paper presents ϵ-approximation-based parallel algorithms using multicore CPUs and manycore GPUs that achieve load balancing, extensive utilization of GPU resources, and achieve minimal memory requirement through simple leaf node retrieval and CUDA kernel optimizations. Average speedup factors of two orders of magnitude compared to a serial code have been demonstrated with various molecular structures and error ranges for the selected energy terms. In our hybrid approach a simple yet reliable parametrization for distributing workload to cores of CPUs and GPUs was adopted, and experimental results show that our method can produce the best performance for all error ranges.

Our technique requires no intermediate communication between threads except for the final accummulation of the intermediate results. Furthermore, our node pair ordering scheme can enable efficient data splitting and loading on each node once octrees are generated. Therefore, it can be easily extended to run on heterogeneous multi-node CPU and GPU systems with increased scalability.

Acknowledgement

This research of DC, ZQ, AR, CB was supported in part by NIH grant R01- EB004873, NSF grant OCI-1216701 and Sandia subcontract SNL-1439100; and of JJ, RC by NSF grants CCF-1162196 and CCF-1439084, and by XSEDE (NSF grant OCI-1053575). Research on this project was conducted when DC, ZQ, AR and in part by RC, when they were at the Computational Visualization Center, Institute of Computational Engineering and Sciences, at the University of Texas at Austin. We thank Carlos Simmerling for his help with Amber-14.

6. REFERENCES

[1]. Anderson JA, Lorenz CD, and Travesset A. General purpose molecular dynamics simulations fully implemented on graphics processing units. J. Chem. Phys, 227(10):5342–5359, 2008.

[2]. Bashford D and Case DA. Generalized Born models of macromolecular solvation effects. Annu. Rev. Phys. Chem, 51:129–152, 2000. [PubMed: 11031278]

[3]. Carrier J, Greengard L, and Rokhlin V. A fast adaptive multipole algorithm for particle simulations. SAIM J. Sci. Stat. Comput, 9(4):669–686, 1988.

[4]. Chowdhury R, Beglov D, Moghadasi M, Paschalidis I, Vakili P, Vajda S, Bajaj C, and Kozakov D. Efficient maintenance and update of nonbonded lists in macromolecular simulations. J. Chem. Theor. Comput, 10(10):4449–4454, 2014.

[5]. Chowdhury RA and Bajaj C. Algorithms for faster molecular energetics, forces and interfaces. ICES report 10–32, Institute for Computational Engineering & Science, The University of Texas at Austin, Austin, TX, USA 78712., 8 2010.

[6]. Case DA et al. AMBER 12, 2012 University of California, San Francisco.

[7]. Case DA et al. AMBER 14, 2014 University of California, San Francisco.

[8]. Darve E, Cecka C, and Takahashi T. The fast multipole method on parallel clusters, multicore processors, and graphics processing units. Comptes Rendus Mecanique, 339(2–3):185–193, 2011.

[9]. Dynerman D, Butzlaff E, and Mitchell JC. CUSA and CUDE: GPU-accelerated methods for estimating solvent accessible surface area and desolvation. J. Comput. Biol, 16:523–537, 4 2009. [PubMed: 19361325]

[10]. Dzubiella J, Swanson JMJ, and McCammon JA. Coupling hydrophobic, dispersion, and electrostatic contributions in continuum solvent models. Phys. Rev. Lett, 96(8), 2006.

[11]. Eisenberg D and McLachlan A. Solvation energy in protein folding and binding. Nature, 319:199–203, 1986. [PubMed: 3945310]

[12]. Friedrichs MS, Eastman P, Vaidyanathan V, Houston M, Legrand S, Beberg AL, Ensign DL, Bruns CM, and Pande VS. Accelerating molecular dynamic simulation on graphics processing units. J. Comput. Chem, 30:864–872, 2009. [PubMed: 19191337]

[13]. Frigo M, Halpern P, Leiserson CE, and Lewin-Berlin S. Reducers and other cilk++ hyperobjects. In Proc. 21st Annu. Symp. Parl. Algo. Archi., pages 79–90. ACM, 2009.

[14]. Gallicchio E, Zhang LY, and Levy RM. The SGB/NP hydration free energy model based on the surface generalized Born solvent reaction field and novel nonpolar hydration free energy estimators. J. Comput. Chem, 23:517–519, 2002. [PubMed: 11948578]

[15]. Gibson KD and Scheraga HA. Minimization of polypeptide energy, I: Preliminary structures of bovine pancreatic ribonuclease s-peptide. Proc. Natl. Acad. Sci. (USA), 58:420–427, 1967. [PubMed: 5233450]

[16]. Gilson M, Davis M, Luty B, and McCammon J. Computation of electrostatic forces on solvated molecules using the Poisson-Boltzmann equation. J. Phys. Chem, 97:3591–3600, 1993.

[17]. Götz AW, Williamson MJ, Xu D, Poole D, Le Grand S, and Walker RC. Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. generalized Born. J. Chem. Theor. Comput, 8(5):1542–1555, 2012.

[18]. Goude A and Engblom S. Adaptive fast multipole methods on the GPU Technical Report 2012–012, Uppsala University, Electricity, 2012.

[19]. Greengard L and Rokhlin V. A fast algorithm for particle simulations. J. Comput. Phys, 73(2):325–348, 12 1987.

[20]. Grycuk T. Deficiency of the Coulomb-field approximation in the generalized Born model: An improved formula for Born radii evaluation. J. Chem. Phys, 119:4817–4826, 2003.

[21]. Gumerov NA and Duraiswami R. Fast multipole methods on graphics processors. J. Chem. Phys, 227:8290–8313, 2008/09/10/ 2008.

[22]. Hamada S. GPU-accelerated indirect boundary element method for voxel model analyses with fast multipole method. Comput. Phys. Comm, 182(5):1162–1168, 2011.

[23]. Hermann R. Theory of hydrophobic bonding. II. correlation of hydrocarbon solubility in water with solvent cavity surface area. J. Phys. Chem, 76:2754–2759, 1972.

[24]. Hu Q, Gumerov NA, and Duraiswami R. Scalable fast multipole methods on distributed heterogeneous architectures. In Proc. 2011 Intl Conf. High Perf. Comput. Ntw. Stg. Anlys., SC ’11, pages 36:1–36:12. ACM, 2011.

[25]. Hummer G, Garde S, Garcia AE, and Pratt LR. New perspectives on hydrophobic effects. Chem. Phys, 258(2–3):349–370, 2000.

[26]. Lashuk I. A massively parallel adaptive fast-multipole method on heterogeneous architectures; Proc. Conf. High Perf. Computing Ntw. Stg. Anlys.; ACM; 2009. 1–58.

[27]. Im W, Lee MS, and Brooks CL. Generalized Born model with a simple smoothing function. J. Comput. Chem, 24:1691–1702, 2003. [PubMed: 12964188]

[28]. Phillips JC et al. Scalable molecular dynamics with NAMD. J. Comput. Chem, 26(16):1781–1802, 2005. [PubMed: 16222654]

[29]. Jackins CL and Tanimoto SL. Oct-trees and their use in representing three-dimensional objects. Comput. Graph. Image Proc, 14(3):249–270, 1980.

[30]. Lee MS, Feig M, Salsbury FR, and Brooks CL. New analytic approximation to the standard molecular volume definition and its application to generalized Born calculations. J. Comput. Chem, 24:1348–1356, 2003. [PubMed: 12827676]

[31]. Lee MS, Salsbury FR, and Brooks CL. Novel generalized Born methods. J. Chem. Phys, 116:10606–10614, 2002.

[32]. Leiserson CE. The Cilk++ concurrency platform. In Proc. Annu. Des. Auto. Conf., pages 522–527. ACM, 2009.

[33]. Levy RM, Zhang LY, Gallicchio E, and Felts AK. On the nonpolar hydration free energy of proteins: Surface area and continuum solvent models for the solute-solvent interaction energy. J. Am. Chem. Soc, 125:9523–9530, 2003. [PubMed: 12889983]

[34]. Liu W, Schmidt B, Voss G, and MÃijller-Wittig W. Accelerating molecular dynamics simulations using graphics processing units with CUDA. Comput. Phys. Comm, 179(9):634–641, 2008.

[35]. Lum K, Chandler D, and Weeks JD. Hydrophobicity at small and large length scales. J. Phys. Chem. B, 103:4570–4577, 1999.

[36]. Maximova T and Keasar C. A noval algorithm for non-bonded-list updating in molecular simulation. J. Comput. Biol, 13(5):1041–1048, 2006. [PubMed: 16796550]

[37]. Plimpton S. Fast parallel algorithms for short-range molecular dynamics. J. Chem. Phys, 117(1):1–19, 1995.

[38]. Rajamani S, Truskett TM, and Garde S. Hydrophobic hydration from samll to large length scales: Understanding and manipulating the crossover. Proc. Natl. Acad. Sci, 102:9475–9480, 2005. [PubMed: 15972804]

[39]. Rokhlin V. Rapid solution of integral equations of classical potential theory. J. Chem. Phys, 60(2):187–207, 1985.

[40]. Ruymgaart AP, Cardenas AE, and Elber R. Moil-opt: Energy-conserving molecular dynamics on a GPU/CPU system. J. Chem. Theor. Comput, 7(10):3072–3082, 2011.

[41]. Ruymgaart AP and Elber R. Revisiting molecular dynamics on a CPU/GPU system: Water kernel and shake parallelization. J. Chem. Theor. Comput, 8(11):4624–4636, 2012.

[42]. Pronk S et al. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. Bioinformatics, page btt055, 2013.

[43]. Sharp K. Incorporating solvent and ion screening into molecular dynamics using the finite-difference Poisson-Boltzmann method. J. Comput. Chem, 12:454–468, 1991.

[44]. Sharp K and Honig B. Calculating total electrostatic energies with the nonlinear Poisson-Boltzmann equation. J. Phys. Chem, 94:7684–7692, 1990.

[45]. Simonson T and Bruenger A. Solvation free energies estimated from macroscopic continuum theory: An accuracy assessment. J. Phys. Chem, 98:4683–4694, 1994.

[46]. Srinivasan J, Trevathan M, Beroza P, and Case D. Application of a pairwise generalized Born model to proteins and nucleic acids: inclusion of salt effects. Theor. Chem. Accts, 101:426–434, 1999.

[47]. Still WC, Tempczyk A, Hawley RC, and Hendrickson T. Semianalytical treatment of solvation for molecular mechanics and dynamics. J. Am. Chem. Soc, 112:6127–6129, 1990.

[48]. Stock MJ and Gharakhani A. Toward efficient GPU-accelerated N-body simulation. In Proc. 46th AIAA Aeros. Sci. Mtg. Exhib, pages 1–13, 2008.

[49]. Stone JE, Phillips JC, Freddolino PL, Hardy DJ, Trabuco LG, and Schulten K. Accelerating molecular modeling applications with graphics processors. J. Chem. Phys, 28:2618–2640, 9 2007.

[50]. Tithi JJ and Chowdhury RA. Polarization energy on a cluster of multicores. In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, pages 569–578. IEEE, 2013.

[51]. Tjong H and Zhou H-X. GBr^6: A parameterization-free, accurate, analytical generalized Born method. J. Phys. Chem. B, 111:3055–3061, 2007. [PubMed: 17309289]

[52]. Wagoner J and Baker NA. Assessing implicit models for nonpolar mean solvation forces: The importance of dispersion and volume terms. Proc. Natl. Acad. Sci. USA, 103:8331–8336, 2006. [PubMed: 16709675]

[53]. Walker R, 2014 Personal communication.

[54]. Walters JP, Balu V, Chaudhary V, Kofke D, and Schultz A. Acclerating molecular dynamics simulations with GPUs. In Proc. 21st Intl Soc. Computers and their Appl. Parl. Dist. Computing Commun. Sys, ISCA-PDCCS’08, page 6, 2008.

[55]. Wu Q, Yang C, Tang T, and Lu K. Fast parallel cutoff pair interactions for molecular dynamics on heterogeneous systems. Tsinghua Science and Technology, 17(3):265 –277, 6 2012.

[56]. Xu J, Ren Y, Ge W, Yu X, Yang X, and Li J. Molecular dynamics simulation of macromolecules using graphics processing unit. J. Mol. Struct. (Theochem), 36:1131–1140, 2010.

[57]. Yang J, Wang Y, and Chen Y. GPU accelerated molecular dynamics simulation of thermal conductivities. J. Comput. Phys, 221:799–804, 2 2007.

[58]. Yao Z, Wang J-S, Liu G-R, and Cheng M. Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method. Comput. Phys. Comm, 161(1–2):27–35, 2004.

[59]. Yokota R and Barba L. Comparing the treecode with FMM on GPUs for vortex particle simulations of a leapfrogging vortex ring. Computers & Fluids, 45(1):155–161, 2011.

[60]. Yokota R, Narumi T, Sakamaki R, Kameoka S, Obi S, and Yasuoka K. Fast multipole methods on a cluster of GPUs for the meshless simulation of turbulence. Comput. Phys. Comm, 180(11):2066–2078, 2009.

[61]. Zhao X and Feng Z. Fast multipole method on GPU: tackling 3-d capacitance extraction on massively parallel SIMD platforms. In Proc. 48th Des. Autom. Conf., DAC ’11, pages 558–563, 2011.

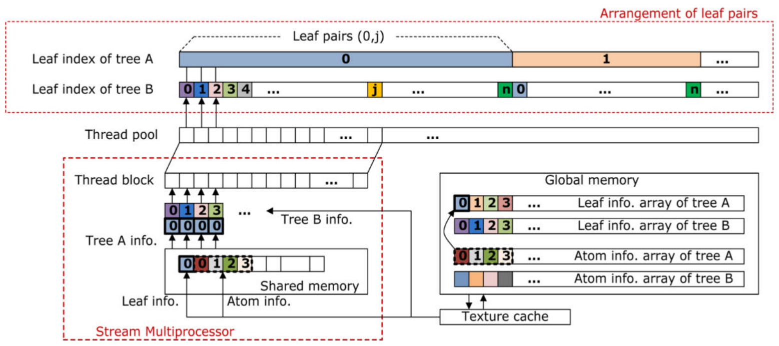


Figure 1:
Arrangement of indices in each leaf node pair to exploit the hierarchical cache structure.

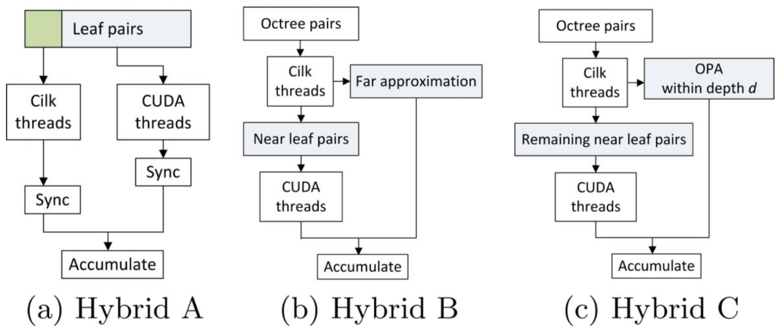
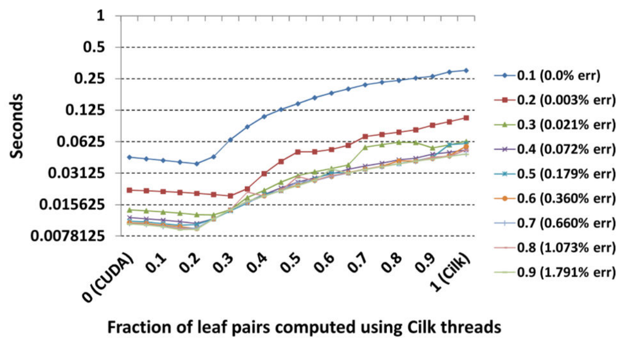
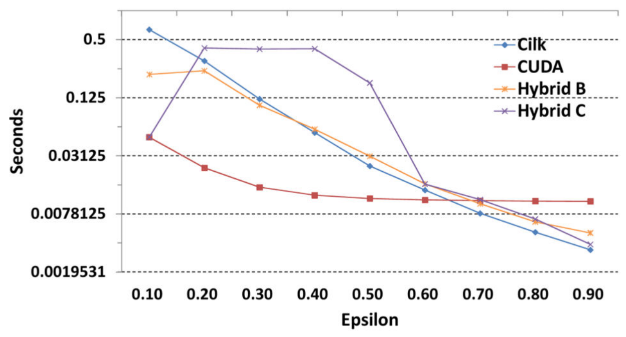


Figure 2:
Our hybrid algorithms exploiting both CPU and GPU threads.



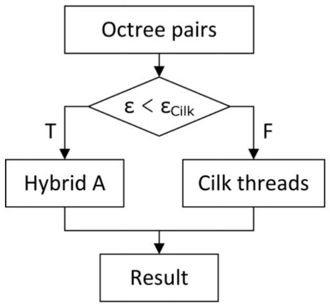
(a) Hybrid A



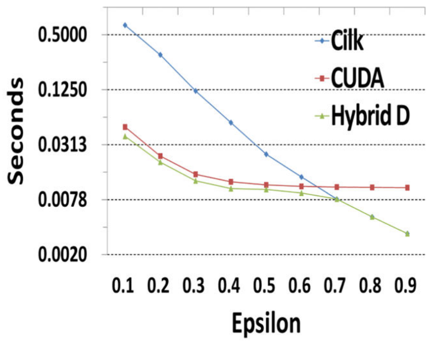
(b) Hybrid B & C

Figure 3:

(a) Performance of method A for different P_{Cilk} and ϵ values. (b) Performance of methods B and C w.r.t. pure CPU/GPU-oriented computation.

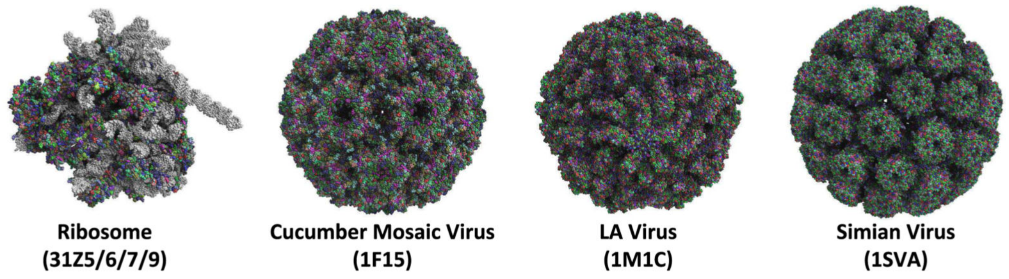


(a) Hybrid D

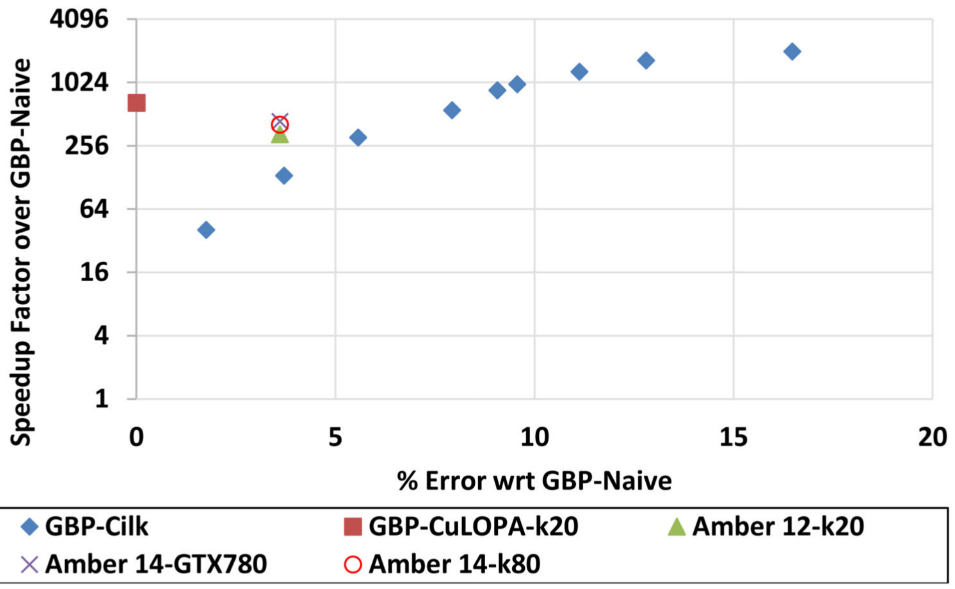


(b) Performance comparison

Figure 4:
(a) Hybrid algorithm D. (b) Performance of Hybrid D w.r.t. pure CPU/GPU methods for different ϵ.



(a) Structures of the molecules shown in the table.



(b) GBP-CuLOPA speedup over GBP-Naïve for the CMVC.

PDB ID	N-Atoms	N-Quads	% Err	Speedup	Mem (MB)
3IZ5/6/7/9	346,817 (39,396)	1,582,380 (211,900)	0.4%	647.1 (1/1)	181.3 (88.0)
1F15	509,640 (72,258)	1,929,128 (88,888)	0.4%	802.7 (2.2/2.2)	161.4 (89.6)
1M1C	1,220,160 (174,159)	1,859,404 (252,702)	0.8%	553.2 (10.6/12.4)	740.7 (248.9)
1SVA	1,908,060 (314,140)	2,097,224 (285,845)	0.6%	406.7 (33.8/30.3)	1446.3 (442.7)

(c) Performance of GBP-CuLOPA on large molecules using GTX580.

Figure 5:
Performance of GBP-CuLOPA on different GPUs and comparison with Amber.

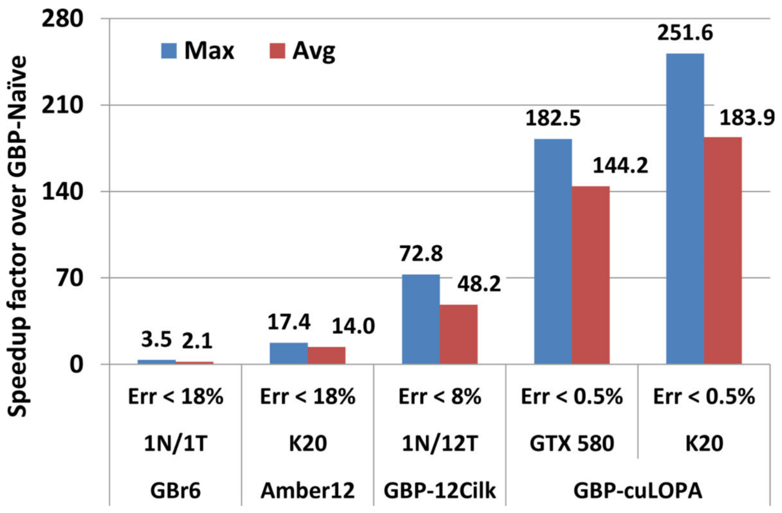


Figure 6:
GBP-CuOPA speedup results over GBP-Naïve. The result for Amber12 GPU version on Kepler 20 GPU [6], GBr6 [51], GBP-12Cilk are also shown for comparison. The titles in each row under horizontal axis represent % error with respect to GBP-Naïve, the number of nodes (N) and threads (T) for CPU-based method and the GPU model for GBP-CuLOPA, and name of method(program) used. 100 molecules having number of atoms between 1,304 and 8,336 were used from ZDock benchmark suite.

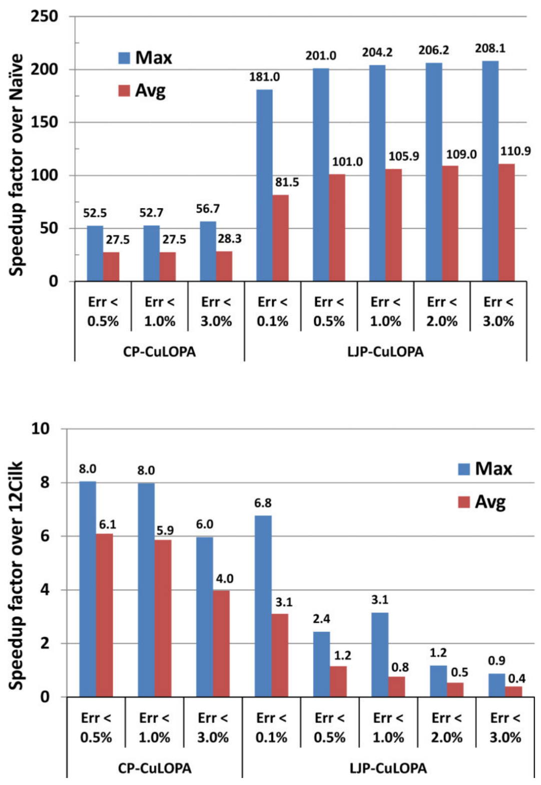
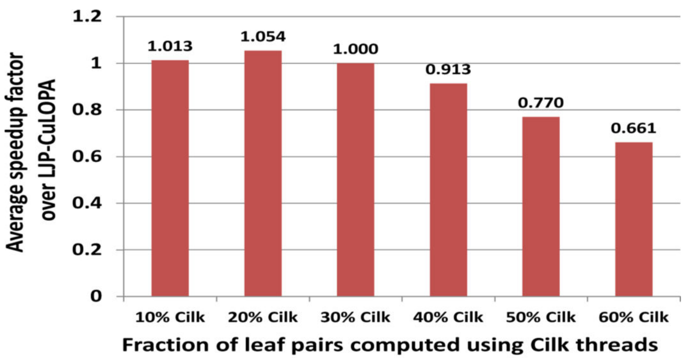


Figure 7:
CUDA speedup results for Coulomb and Lennard-Jones potential computations. The maximum and average speedup factors of CP/LJP-CuLOPA (system: GTX 580) over (Top) Naïve and (Bottom) 12Cilk (system: 12-core Xeon E5680) are shown for the largest 50 proteins in ZDock.



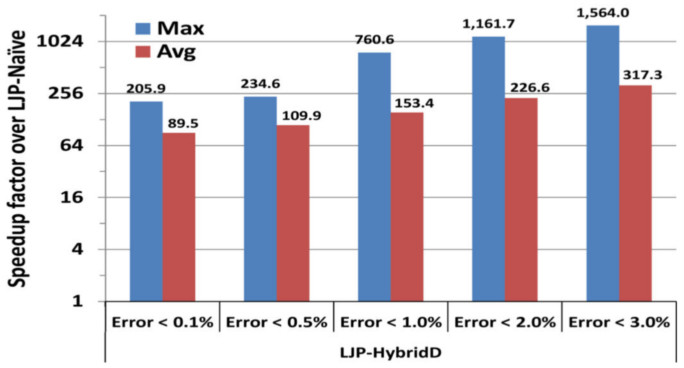


Figure 8:
Performance enhancement by hybrid methods A and D on LJP calculation. This experiment was done with 2 Xeon E5640 and GTX 580. (Top) An average speedup graph of hybrid method A over CuLOPA is shown as the fraction of leaf pairs using Cilk threads varies. (Bottom) Maximum and average speedup factors of LJP-Hybrid D over LJP-Naïve for various error bounds are shown. Eight Cilk threads were used for Hybrid A and twelve Cilk threads were used for OPA when $\epsilon_{Cilk} \geq 0.7$.

Table 1:

Impact of different k values.

k values =>	**16**	**32**	**48**	**64**
kernel execution time (ms)	4.4	2.2	3.5	4.0
warp execution efficiency (%)	30.9	49.7	39.1	33.6
instructions per cycle	1.5	1.6	1.3	1.3
multiprocessor efficiency (%)	98.6	94.5	91.8	83.3
achieved occupancy	0.56	0.61	0.61	0.57
instructions issued (millions)	64M	34M	43M	50M

Table 2:

Performance results of CUDA (cuLOPA) and Cilk+CUDA hybrid (Hybrid D) implementation.

Energetic		**CP**	**LJP**		**GBP**
Method		**cuLOPA**	**cuLOPA**	**Hybrid D**	**cuLOPA**
Bounded Err		0.5%	0.1%	0.1%	0.7%
Speedup over Naïve	Max.	52.5	181.0	205.9	302.6
	Avg.	27.5	81.5	89.5	243.9
Avg. time (μs) / atom		0.45	0.25	0.23	2.76

Table 3:

Comparison of parallel testing platforms for multicore CPUs and manycores GPUs.

Parallel target	**CPU**			**GPU**			
Processor	**Intel Xeon E5640**	**Intel Xeon E5680**	**Intel Xeon E2680**	**Nvidia Geforce GTX 580**	**Nvidia Geforce GTX 780**	**Nvidia Kepler K 20**	**Nvidia Kepler K 80**
Core architecture	**Nahalem**		**Sandy Bridge**	**Fermi**	**GK 104**	**GK 110**	**GK210**
#cores/PU	4	6	8	512	2304	2688	2880
Core clock	2.67 GHz	3.33 GHz	2.70 GHz	0.77 GHz	1.02 GHz	0.71 GHz	0.82 GHz
L1 cache	64 KB		32KB	16–48 KB	64 KB	64 KB	128 KB
L2 cache	256 KB			768 KB	1536KB	1536 KB	1536 KB
L3 cache	12 MB		20MB	-	-	-	-
compiler	-	icc 12.0	icc 13.0	cuda 3.2	cuda 5.0	cuda 5.5	cuda 6.5