J. Parallel Distrib. Comput. 87 (2016) 91–101

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A load-balancing workload distribution scheme for three-body interaction computation on Graphics Processing Units (GPU)

Ashraf Yaseen [a], Hao Ji [b], Yaohang Li [b,*]

[a] Department of Electrical Engineering and Computer Science, Texas A&M University-Kingsville, Kingsville, TX 78363, United States
[b] Department of Computer Science, Old Dominion University, Norfolk, VA 23529, United States

HIGHLIGHTS

- Load-balancing scheme for calculating three-body interactions on GPU.
- Perfect load-balancing is achieved if N is not divisible by 3.
- Nearly perfect load-balancing is obtained if N is divisible by 3.
- Parallel efficiency demonstrated in three-body potentials.

ARTICLE INFO

Article history:
Received 17 July 2014
Received in revised form
19 September 2015
Accepted 14 October 2015
Available online 24 October 2015

Keywords:
GPU
Three-body interactions
Load balancing

ABSTRACT

Three-body effects play an important role for obtaining quantitatively high accuracy in a variety of molecular simulation applications. However, evaluation of three-body potentials is computationally costly, generally of $O(N^3)$ where N is the number of particles in a system. In this paper, we present a load-balancing workload distribution scheme for calculating three-body interactions by taking advantage of the Graphics Processing Units (GPU) architectures. Perfect load-balancing is achieved if N is not divisible by 3 and nearly perfect load-balancing is obtained if N is divisible by 3. The workload distribution scheme is particularly suitable for the GPU's Single Instruction Multiple Threads (SIMT) architecture, where particle's data accessed by threads can be coalesced into efficient memory transactions. We use two potential energy functions with three-body terms, the Axilrod–Teller potential and the Context-based Secondary Structure Potential, as examples to demonstrate the effectiveness of our workload distribution scheme.

1. Introduction

Although many molecular simulations are typically confined to evaluating interactions between molecular pairs, recent studies show that three-body or even higher order effects play an important role for quantitatively accurate computation in a variety of molecular simulation applications [10,33,24,26]. For example, the three-body effects strongly influence solid–liquid and vapor–liquid equilibrium of fluids [34,32,25,2]. The context-based secondary structure potential (CSSP) taking three-body statistical terms into account leads to significant accuracy enhancement in evaluating protein secondary structures [21]. A three-body potential incorporating interaction between a DNA base and a protein residue with regard to the effect of a neighboring DNA base outperforms two-body potentials in specific protein—DNA site recognition [37]. A four-body residue–residue contact potential has demonstrated its effectiveness compared with pairwise potentials in discriminating native protein conformations [11]. Inclusion of three-body effects in the additive CHARMM protein CMAP potential also results in enhanced cooperativity of α-helix and β-hairpin formations [7].

Despite the advantages of evaluating three-body interactions in molecular simulations, the main obstacle of a potential energy involving three-body or higher order terms is its high computational cost. In general, when external influences are not presented, the potential energy of a system with N particles can be evaluated as

$$E = \sum_{i \neq j} U(p_i, p_j) + \sum_{i \neq j \neq k} U(p_i, p_j, p_k) + \sum_{i \neq j \neq k \neq l} U(p_i, p_j, p_k, p_l) + \ldots,$$

* Corresponding author.
E-mail addresses: ashraf.yaseen@tamuk.edu (A. Yaseen), hji@cs.odu.edu (H. Ji), yaohang@cs.odu.edu (Y. Li).

http://dx.doi.org/10.1016/j.jpdc.2015.10.003
0743-7315/

where $U(p_i, p_j)$ is the two-body term involving two particles p_i and p_j, $U(p_i, p_j, p_k)$ is the three-body term, $U(p_i, p_j, p_k, p_l)$ is the four-body term, and so on. Three-body potentials explicitly calculate the three-body terms which are ignored in two-body potentials together with other higher order terms. Therefore, provided that the three-body terms are calculated accurately, a three-body potential usually yields better accuracy than the two-body ones. However, the tradeoff is significantly more computing time. The computing time increases largely when higher order terms are included. Generally, two-body terms require $O(N^2)$ operations, while $O(N^3)$ for three-body terms, $O(N^4)$ for four-body terms, and so forth. Computation reduction approaches such as Barnes–Hut method, fast multipole, and particle-mesh [4,9,14,17] can significantly reduce the overall computation complexity by simplifying interactions between far apart particles. Nevertheless, simulation using computation involving three- or higher-body terms is still unrealistic for a system with relatively large number of interacting molecules until recent performance improvements have been achieved in computer systems.

Graphical Processing Units (GPUs) have been a powerful computational tool in many science and engineering related problems. Today's GPUs greatly outpace CPUs in arithmetic throughput and memory bandwidth, forming a dramatic shift in the field of high performance computing (HPC) [30]. With the massively parallel computing mechanisms, GPUs are able to deliver performance speedups magnitudes of times higher than the CPU, to solve problems in a few minutes instead of hours or days [22,38,35]. Hence, GPUs has become the ideal processor to accelerate a wide variety of data parallel applications. Being able to simultaneously calculate forces on N particles, GPU has been employed to accelerate the N-body molecular simulation in a variety of applications. For example, Belleman et al. [6] developed a direct gravitational N-body simulation on GPU. Nyland et al. [19] implemented a fast N-body simulation on GPU in astrophysical applications. Stock and Gharakhani [31] introduced a GPU-accelerated multipole-accelerated treecode method for turbulent flow computations. Lashuk et al. [20] proposed a parallel fast multipole method (FMM) on heterogeneous architectures with CPUs and GPUs. Anderson et al. [1] carried out general molecular dynamics on GPU, by simulating N particles in a finite box with periodic boundary conditions. Friedrichs et al. [12] also presented an accelerated molecular dynamic simulation implemented on GPU. Hamada et al. [16] came up with an efficient GPU implementation of Barnes–Hut algorithm on large data sets in astrophysics and turbulence and Jetley et al. [18] provided a scalable implementation on GPU clusters. Grimm and Stadel [15] designed a hybrid symplectic N-body integrator to analysis planet and planetesimal dynamics in the late stage of planet formation with GPU acceleration. Significant speedups have been observed in the above applications.

Although the N-body interactions can be carried out in a straightforward way on a serial processor, efficient parallel implementation to fully take advantage of GPU architectures requires deliberate considerations. Our previous work [35] on symmetric two-body interaction computation on GPU has led to a workload distribution scheme, which is designed to assign computations of pair-wise atom interactions to GPU threads to achieve perfect or near-perfect load balancing in the symmetric N-body problem while facilitating coalesce memory access. In this paper, we extend our previous GPU-based two-body load-balancing workload distribution scheme to explicitly calculations of three-body terms. The effectiveness of our approach is demonstrated in the computation of the Axilrod–Teller potential [3], a physics-based three-body potential function, and the Context-based Secondary Structure Potential (CSSP), a knowledge-based potential energy function with three-body terms to evaluate protein conformation adopting certain secondary structure pattern [21,36].

The rest of the paper is organized as follows. Our proposed GPU workload distribution for computing three-body interactions is presented in Section 2. Section 3 shows our computational results. Finally, Section 4 summarizes our conclusions and future research directions.

2. GPU-based load-balancing scheme for computing three-body interactions

Our load-balancing scheme assumes that the three-body interaction terms are independent of the order of the three particles. In other words, the order permutation of the three particles does not change the potential value. Moreover, for simplicity in illustration, we assume one-thread-per-particle assignment, i.e., each thread keeps one particle information unchanged and then shifts the second or the third particles information at each iteration to obtain all combinations of triplets. A fine-grained assignment other than the one-thread-per-particle assignment to enhance the GPU performance on systems with small number of particles is discussed in Section 2.4.

2.1. Serial implementation

The general serial implementation of three-body interaction computation is straightforward. All one needs to do is to enumerate all triplet combinations of three different particles and then calculate the three-body energy of the triplet. The corresponding pseudo code is illustrated in Algorithm 1. For a molecular system with N particles and assuming that each pair of atoms are interacting, there are totally $N * (N - 1) * (N - 2)/6$ triplet computations and each particle is involved in exactly $(N - 1) * (N - 2)/2$ interaction computations. However, for each particle in the outer loop in Algorithm 1, its number of three-body interaction calculations in the middle and inner loops varies gradually from $(N - 1) * (N - 2)/2$ to 1. Consequently, directly mapping the calculations in the middle and inner loops to GPU threads will lead to a highly unbalanced workload distribution.

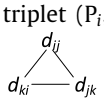
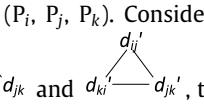
2.2. Rotational symmetry

Our implementation of load-balancing workload distribution scheme for three-body interaction computation on GPU is based on the concept of rotational symmetry. Assuming that the N particles in the system are stored in a cyclic array, we use i, j, and k to denote the indices of the three particles in clockwise order, and d_{ij}, d_{jk}, and d_{ki} to denote the position separations between particles i and j, j and k, and k and i, respectively. Here d_{ij} is calculated as

$$d_{ij} = \begin{cases} j - i, & i < j \\ j - i + N, & i > j \end{cases}$$

d_{jk} and d_{ki} are calculated in a similar way. Clearly, we can have the following two properties

(1) $i \neq j \neq k$, and
(2) $d_{ij} + d_{jk} + d_{ki} = N$.

Then, we study the position separation pattern of  of a triplet (P_i, P_j, P_k). Considering two position separation patterns  and  , they are rotationally symmetric if $d_{ij} = d'_{ij}$ and $d_{jk} = d'_{jk}$ and $d_{ki} = d'_{ki}$ or $d_{ij} = d'_{jk}$ and $d_{jk} = d'_{ki}$ and $d_{ki} = d'_{ij}$ or $d_{ij} = d'_{ki}$ and $d_{jk} = d'_{ij}$ and $d_{ki} = d'_{jk}$. Or equivalently, if  can turn into  via cyclic rotations, then

Algorithm 1. Serial three-body potential calculation for N particles

```
sumE ← 0.0;                                  // initialization
for i ← 1 to N-2 {                           // first particle, outer loop
   particle1 = i;
   for j ← i+1 to N-1 {                      // second particle, middle loop
      particle2 = j;
      for k ← j+1 to N {                     // third particle, inner loop
          particle3 = k;                     // sum overall potential
          sumE ← sumE + TripleE(particle1, particle2, particle3);
      }
   }
}
```

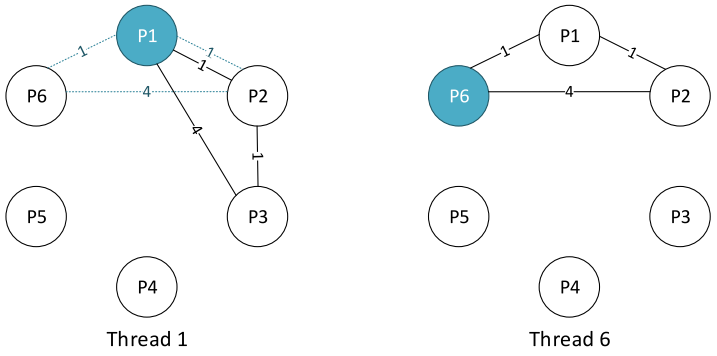


Fig. 1. An example with $N = 6$ with thread 1 starting from P1, and thread 6 from P6. The highlighted particle is the first particle that a thread handles. For separation pattern $4\overset{1}{\triangle}1$, threads 1 and 6 will calculate three-body interactions of triplets (P1, P2, P3) and (P6, P1, P2), respectively. If thread 1 adopts a position separation pattern $1\overset{1}{\triangle}4$, which is rotationally symmetric to $4\overset{1}{\triangle}1$, then thread 1 will compute three-body interaction of triplet (P1, P2, P6), which overlaps the computation of thread 6 with position separation pattern $4\overset{1}{\triangle}1$.

$d_{ij}\overset{}{\triangle}d_{jk}$ and $d_{ij}'\overset{}{\triangle}d_{jk}'$ are rotationally symmetric. Given an example of a system where $N = 6$, $3\overset{1}{\triangle}2$, $2\overset{3}{\triangle}1$ and $1\overset{2}{\triangle}3$ are rotationally symmetric but $3\overset{1}{\triangle}2$ and $2\overset{1}{\triangle}3$ are not. In our GPU implementation, assuming one-thread-per-particle and all threads share the same computation pattern due to GPU's SIMT architecture, rotationally symmetric position separation patterns indicate that some threads will calculate certain triplets in overlap, which will lead to waste of computational power and, more seriously, erroneous results if not handled correctly. Fig. 1 illustrates an example where $N = 6$. Thread 1 starts from particle P1 and triplet (P1, P2, P3) has position separation pattern $4\overset{1}{\triangle}1$. If a rotationally symmetric position separation pattern of $4\overset{1}{\triangle}1$, for instance, $1\overset{1}{\triangle}4$ is adopted, then thread 1 will carry out three-body interaction calculation on triplet (P1, P2, P6), which is the same as the three-body interaction calculation on triplet (P6, P1, P2) in thread 6 starting at P6 with position separation pattern $4\overset{1}{\triangle}1$. In summary, the fundamental idea of our GPU-based algorithm is to uniquely enumerate all position separation patterns where there are no rotationally symmetric pairs.

To balance workload distribution among GPU threads, we design a novel workload distribution scheme by enumerating all position separation patterns without sharing rotational symmetry. The workload distribution scheme is described in Algorithm 2. For thread i, the index of the first particle is always i specified by the passed parameter and the second and third particles are selected according to the enumerated position separation patterns. The algorithm enumerates all position separation patterns satisfying

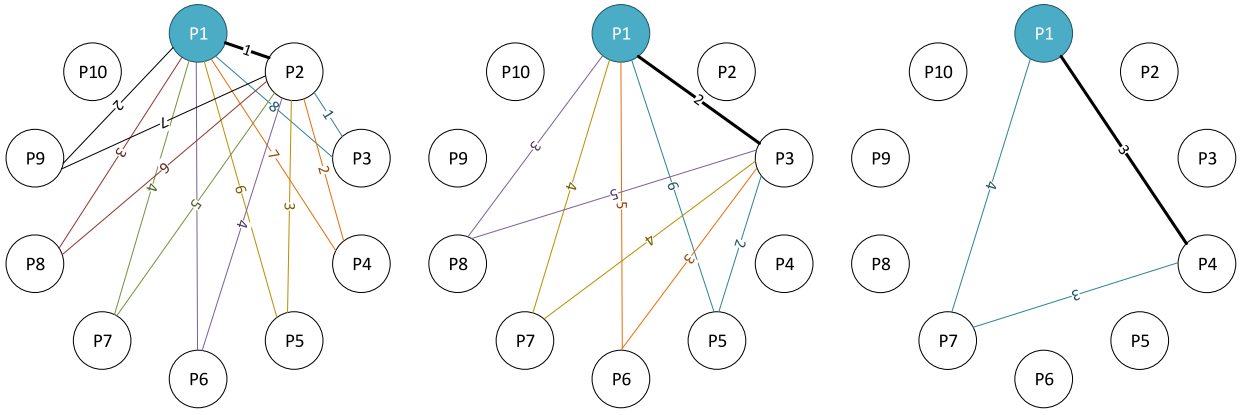
$$d_{ij} \le d_{jk} \quad \text{and} \quad d_{ij} < d_{ki}. \tag{1}$$

It is easy to show that any position separation pattern that does not satisfy the above condition is rotationally symmetric to one of the position separation patterns satisfying this condition, because we can always rotate the smallest position separation to d_{ij}. The above condition also indicates that d_{ij} is bounded by $\lfloor (N-1)/3 \rfloor$. Hence, our algorithm iterates the second particle index from $(i+1)$ mod N to $(i + \lfloor (N-1)/3 \rfloor)$ mod N. Then, the third particle is iterated to satisfy (1).

Fig. 2 illustrates an example of enumerating triplets by the first thread (thread 1) in a system with 10 particles, using Algorithm 2. For thread 1, the first particle is always P1. The second particle iterates from P2 to P4. When the second particle is P2 ($d_{ij} = 1$), three-body interactions of triplets (P1, P2, P3), (P1, P2, P4), (P1, P2, P5), (P1, P2, P6), (P1, P2, P7), (P1, P2, P8), (P1, P2, P9) with position separation patterns $8\overset{1}{\triangle}1$, $7\overset{1}{\triangle}2$, $6\overset{1}{\triangle}3$, $5\overset{1}{\triangle}4$, $4\overset{1}{\triangle}5$, $3\overset{1}{\triangle}6$, and $2\overset{1}{\triangle}7$ are calculated, respectively. When the second particle is P3 ($d_{ij} = 2$), the three-body interactions of triplets (P1, P3, P5), (P1, P3, P6), (P1, P3, P7), (P1, P3, P8) with respective separation patterns $6\overset{2}{\triangle}2$, $5\overset{2}{\triangle}3$, $4\overset{2}{\triangle}4$, and

Algorithm 2. Enumerating all non-rotational symmetric position separation patterns except for the order three symmetric one to calculate three-body interaction of the corresponding triplet particles in a GPU thread.

```
Thread(i, N)                                        // N particles, thread i
{
   localE ← 0.0;                                    // initialization
   particle1 ← i;                                   // particle 1
   range←⌊(N − 1)/3⌋;                               // range of particle 2
   fordij ← 1 to range {
      djk←dij;
      dki←N - djk - dij;
      particle2 ← (i + dij) mod N;                  // particle 2
      while (dki > dij) {                           // range of particle 3
            particle3 ← (i + dij + djk) mod N;      // particle 3
            localE ← localE + TripleE(particle1, particle2, particle3);
            djk←djk + 1;
            dki←dki - 1;
      }
   }
   local_sum_reduce(localE);          // sum partial scores in local block
}
```



(a) $d_{ij} = 1$, the second particle is P2 and three-body interactions of triplets (P1, P2, P3), (P1, P2, P4), (P1, P2, P5), (P1, P2, P6), (P1, P2, P7), (P1, P2, P8), (P1, P2, P9) with position separation patterns 1–8 1, 1–7 2, 1–6 3, 1–5 4, 1–4 5, 1–3 6, and 1–2 7 are calculated, respectively.

(b) $d_{ij} = 2$, the second particle is P3 and three-body interactions of triplets (P1, P3, P5), (P1, P3, P6), (P1, P3, P7), (P1, P3, P8) with separation patterns 2–6 2, 2–5 3, 2–4 4, and 2–3 5 are calculated, respectively.

(c) $d_{ij} = 3$, the second particle and only one three-body interaction of triplet (P1, P4, P7) with separation pattern 3–4 3 is calculated.

Fig. 2. An example of enumerating triplets without sharing rotational symmetry in position separation patterns by the first thread (thread 1) in a system with 10 particles.

2–3 5 are accumulated. When the second particle is iterated to P4 ($d_{ij} = 3$), only one triplet(P1, P4, P7) with separation pattern 3–4 3 can satisfy (1). The completion of the algorithm allows thread 1 to carry out three-body interactions of 12 triplets with different position separation patterns that are not rotationally symmetric. Assuming one-particle-per-thread assignment, the total number of three-body interactions is $12 * 10 = 120 = 10 * 9 * 8/6$.

The only position separation patterns that Algorithm 2 cannot iterate are order three rotationally symmetric patterns in case of $d_{ij} = d_{jk} = d_{ki}$, when N is divisible by 3. The order three rotationally symmetric patterns will cause the triplets with equal separation distances to be calculated repeatedly by different threads. Fig. 3 shows an example of a system with 6 particles, where an order three rotationally symmetric pattern 2–2 2 exists. As a result, threads 1, 3, 5 over calculates triplet (P1, P3, P5) while threads 2, 4, 6 over calculates triplet (P2, P4, P6). Furthermore, if N is divisible by 3, $(N - 1) * (N - 2)$ is no longer divisible by 6 and thus the total number of $N * (N - 1) * (N - 2)/6$ interaction computations cannot be equally distributed to N threads. Consequently, order three rotationally symmetric patterns require special handling.

2.3. Load-balancing workload distribution scheme

Algorithm 3 shows the complete workload distribution algorithm with special handling of the case when N is divisible by 3. This algorithm is based on the pseudo-code provided in Algorithm 2. Only the first $N/3$ threads will carry out the three-

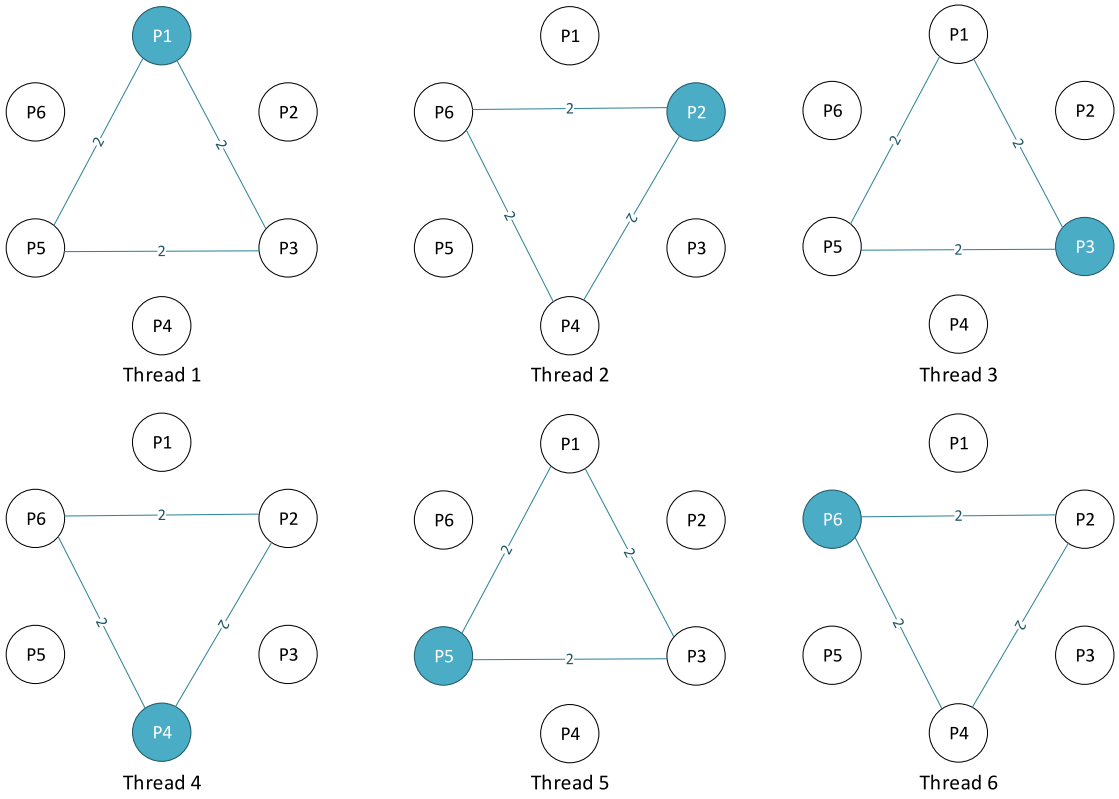


Fig. 3. Order three rotational symmetry in a system with 6 particles. The highlighted particle is the first particle a thread handles. Threads 1, 3, 5 over calculate triplet (P1, P3, P5) while threads 2, 4, 6 over calculate triplet (P2, P4, P6).

body interaction computation of triplets with order three rotationally symmetric position separation patterns to avoid over calculation. When N is not divisible by 3, each thread carries out exactly $(N-1)*(N-2)/6$ three-body interaction operations, where perfect load-balancing is achieved. When N is divisible by 3, the first $N/3$ threads carry out an additional iteration of three-body interaction computation for triplets with order three rotationally symmetric patterns. When N is big enough and thereby a lot of iterations are needed, this additional iteration has little impact to the overall system performance and hence we can claim that nearly perfect load-balancing is obtained.

Fig. 4(a), (b), and (c) show the workload distribution when $N = 7$, 8, and 9, respectively, with perfect load-balancing when $N = 7$ and 8 and nearly perfect load-balancing when $N = 9$. In addition to load balancing, the workload distribution scheme is particularly suitable for the GPU's SIMT architecture [29]. This is due to the fact that, at each iteration step, each thread reads data from different particles with the same stride, which can be coalesced into efficient memory transactions.

2.4. Additional performance improvement implementations on GPU

The above load-balancing workload distribution scheme assumes the one-thread-per-particle on GPU architecture. Nevertheless, for molecular systems with small N value, the one-thread-per-particle scheme with N threads may not produce enough threads to fully utilize all resources in GPU. To address this issue, we implement fine-grained threads by dividing the workload originally assigned to one thread to multiple threads so that sufficient threads are produced when N is small. Firstly, we calculate the number of threads that can be assigned to handle interactions computation of a particle (T_p) by dividing the maximum number of threads that a GPU device can launch ($T_{\max}$) over the total number of particles N.

$$T_p = \lfloor T_{\max}/N \rfloor .$$

Then, we distribute the workload of interaction computation to T_p threads. As a result, large number of threads whose total number is near the maximum number of threads that the GPU can launch are created. In addition to fine-grained treads, other standard CUDA programming techniques, including parallel sum reduction, loop unrolling, and coalesce memory access are implemented in order to fully take advantage of the GPU architecture [29].

3. Computational results

Two three-body potentials, including the Axilrod–Teller potential and the CSSP potential are used to demonstrate the effectiveness of our load-balancing workload distribution scheme on GPU. We name the GPU implementation of Axilrod–Teller and CSSP potential energy functions "GPU-AxT" and "GPU-CSSP," respectively. The load-balancing workload distribution scheme for three-body interactions is adopted in GPU-AxT and GPU-CSSP. As for the two-body interactions in GPU-CSSP, we used our previous approach to balance the workload [35]. Double precision format is used in both functions. Furthermore, the standard CUDA programming techniques for performance improvement are implemented in both potentials.

The GPU-AxT and GPU-CSSP programs are tested on a server with four Tesla C2070 GPUs. The Tesla C2070 GPU (Fermi architecture) has 14 multiprocessors with 32 cores each, 6 GB of global memory, 64 kB of RAM which can be configured between Shared Memory and L1 cache, and 32 kB of registers per multiprocessor.

We benchmark GPU-AxT and GPU-CSSP on a set of systems of various sizes. The GPU time we measured includes the time of transferring the system information (particles) arrays to GPU

Algorithm 3. Load-balancing workload distribution scheme. Perfect load-balancing is achieved when N is not divisible by 3. When N is divisible by 3, an additional iteration is needed for the first N/3 threads.

```
Thread(i, N)                                          // N particles, thread i
{
   localE ← 0.0;                                      // initialization
   particle1 ← i;                                     // particle 1
   range ← ⌊(N − 1)/3⌋;                               // range of particle 2
   fordij ← 1 to range {
      djk ← dij;
      dki ← N - djk - dij;
      particle2 ← (i + dij) mod N;                    // particle 2
      while (dki>dij) {                               // range of particle 3
            particle3 ← (i + dij + djk) mod N;        // particle 3
            localE ← localE + TripleE(particle1, particle2, particle3);
            djk ← djk + 1;
            dki ← dki - 1;
      }
   }
   // Special handling 3-way rotationally symmetric of N is divisible by 3
   if (N mod 3 == 0) {                                // N divisible by 3
      if (i <= N/3) {
            dij = djk = N/3;
            particle2 ← (i + dij) mod N;              // particle 2
            particle3 ← (i + dij + djk) mod N;        // particle 3
            localE ← localE + TripleE(particle1, particle2, particle3);
      }
   }
   local_sum_reduce(localE);           // sum partial scores in local block
}
```

device memories, GPU execution time, and the time of retrieving the calculated results from GPU. We use the nvcc compiler in CUDA 2.0 with "-O3" flag for GPU implementations.

3.1. Computational results of Axilrod–Teller potential

3.1.1. Axilrod–Teller potential

The Axilrod–Teller potential is an intermolecular potential for the interaction of the van der Waals type between three particles [3]. Considering particles i, j, and k, the Axilrod–Teller potential u_{ijk} is calculated as,

$$u_{ijk} = v\left[\frac{1}{r_{ij}^3 r_{ik}^3 r_{jk}^3} + \frac{3(-r_{ij}^2 + r_{ik}^2 + r_{jk}^2)(r_{ij}^2 - r_{ik}^2 + r_{jk}^2)(r_{ij}^2 + r_{ik}^2 - r_{jk}^2)}{8(r_{ij}^5 r_{ik}^5 r_{jk}^5)}\right]$$

where v is a non-additive coefficient and r_{ij}, r_{jk}, and r_{ik} are Euclidean distances between particles i and j, j and k, and k and i, respectively.

3.1.2. GPU-AxT performance

We employ GPU-AxT to calculate the Axilrod–Teller potential energy in a particle system where a simulation box of length L is initialized with N Argon particles (atoms). In order to demonstrate the effectiveness of Axilrod–Teller potential implementation on the GPU (GPU-AxT), we run GPU-AxT on simulation boxes of various sizes (L ranges from 14 for 343 particles to 60 for 27,000 particles). The execution time of the Axilrod–Teller potential evaluation is averaged over 100 runs.

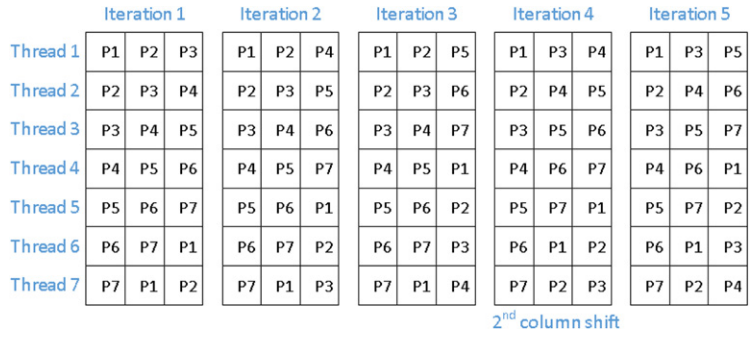
Fig. 5 shows the performance of GPU-AxT implementations on systems of various sizes in terms of double-precision GFLOPS per second on NVIDIA Tesla C2070. For GPU-AxT implementation with one thread per particle, there are certain inefficiencies for systems with less than 2500 particles, due to insufficient number of threads to fully take advantage of the GPU architecture. To improve the performance of systems with small number of particles, we adopt a fine-grained implementation by evenly splitting the workload of three-body interaction computations belonging to one thread to multiple threads so that nearly the maximum number of threads that a GPU device can launch is created. Significant performance improvements are found in systems with smaller number of particles when fine-grained threads are employed, whose floating point throughputs are promoted to be closer to those with a lot of particles. As the number of particles increases, the floating point throughput curves of fine-grain threads and one-thread-per-particle start to merge because the increasing number of threads in one-thread-per-particle strategy makes more and more efficient use of the GPU architecture. For systems with more than 3000 particles, the floating point throughput of GPU-AxT implementation with fine-grained threads is close to but slightly less than that of the two-body potential (262 GFLOPS/s), benchmarked using the "nbody" program provided by Cuda SDK with 100,000 particles. The slight performance loss in three-body potential computation is due to an additional round of memory fetches for triplet computations compared to doublet computations as well as more integer operations to determine the particle indices in each triplet.

Fig. 6 shows the memory-only, arithmetic-only, and full-kernel time that GPU-AxT spends on a system with 3375 particles. One can find that 70.6% of the memory operations are overlapped with the arithmetic operations. After all, with sufficient number of threads, the memory access latency can be effectively masked.

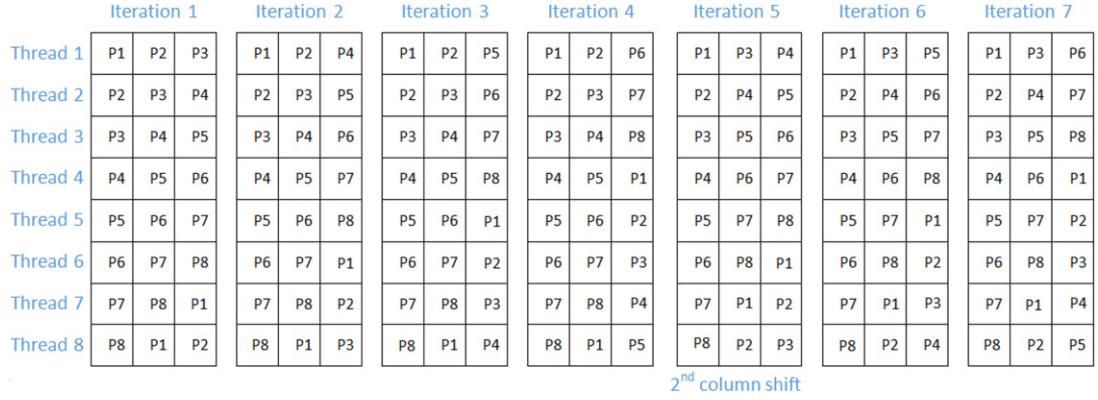
Our GPU-AxT implementation requires transferring the particles coordinates arrays, from the host memory to the GPU device memory and retrieving the calculated overall energy from the device memory. Table 1 shows the data transfer time ($O_{\text{data-transfer}}$) and the calculation time (t_{kernel}) in small, medium, and large systems. The data transfer time is very small compared to the overall GPU-AxT energy evaluation time.

3.1.3. Load balancing scheme vs. Direct mapping scheme

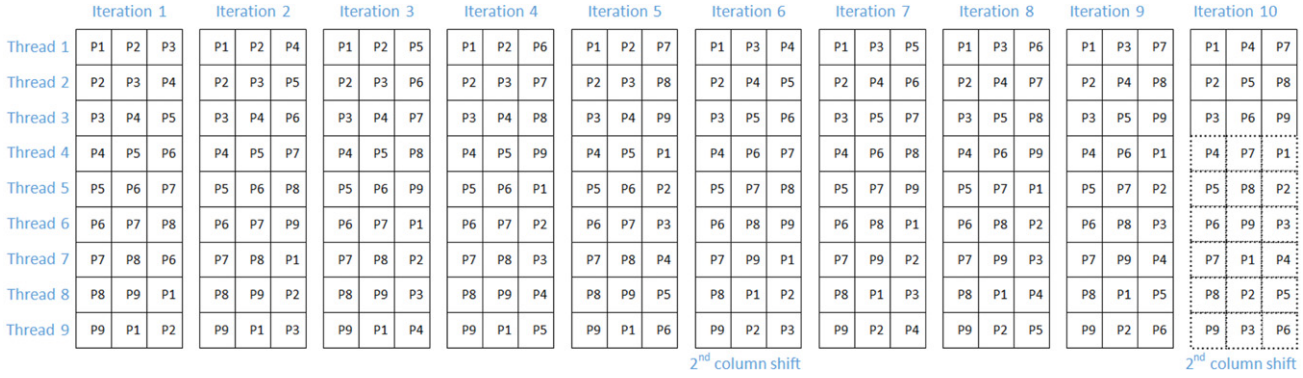
Theoretically, assuming every three particles are interacting with each other, if the serial algorithm is directly mapped to GPU implementation, the longest thread needs to carry out $(N-1)*(N-2)/2$ three-body interaction calculations. When the load-balancing scheme is used, each thread handles at most $(N-1)*(N-2)/6+1$ three-body interactions. Therefore,



(a) Perfect balancing ($N = 7$).



(b) Perfect balancing ($N = 8$).



(c) $N = 9$, due to order three rotational symmetry, only the first three threads handles triplets (1, 4, 7), (2, 5, 8), and (3, 6, 9), respectively, at iteration 10.

Fig. 4. Workload distribution scheme when $N = 7, 8, 9$. Perfect load balancing is achieved when $N = 7$ and 8. Near-perfect load balancing is obtained when $N = 9$.

Table 1
Data transferring times and computation time in GPU-AxT.

Component	Mode	Time (μs)		
		Small-system (3375 particles)	Medium-system (15,625 particles)	Large-system (42,875 particles)
Particle coordinate arrays	Host to device	0.07	0.23	0.62
Axilrod–Teller energy	Device to host	0.07	0.15	0.32
Axilrod–Teller kernel	On device	243.35	25,004.82	455,049.73

the theoretical speedup of the load-balancing scheme over direct mapping scheme is approximately 3. Fig. 7 shows that, when no distance cutoff is applied, the GPU-AxT implementation using the load-balancing scheme is around 3 times faster than that of direct mapping. This agrees well with our theoretical analysis. Particularly in practical large-scale simulations that takes hours to days, the three times speedup of load-balancing scheme over direct mapping is much appreciated.

Nevertheless, in practice, when interactions between particles separated over a certain distance are weak enough to be ignored in computation, truncation by distance cutoff is often applied. Provided that the accumulated errors have negligible impact to simulation accuracy and do not affect the stability of the system, using the distance cutoff can significantly reduce the overall three-body interaction computation. Fig. 7 also shows that the speedup of the GPU-AxT implementation using the load-balancing scheme when half of box width is used as cutoff distance over that of direct mapping is reduced to approximately 1.8.

The performance reduction of the balanced GPU-AxT with distance cutoff is mainly caused by the divergent branches in the

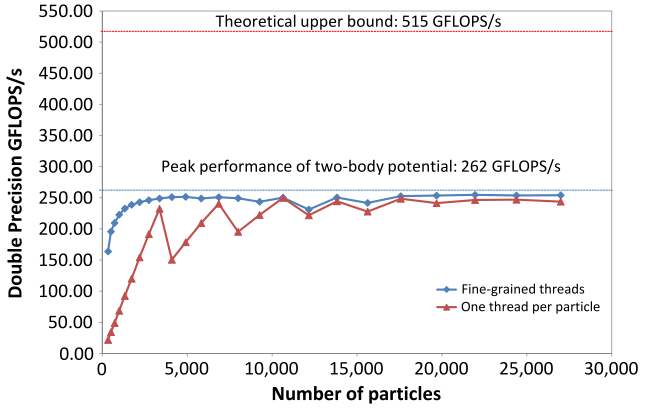


Fig. 5. Performance comparison of GPU-AxT implementations on systems of various sizes in terms of double-precision floating point throughput on NVIDIA Tesla C2070.

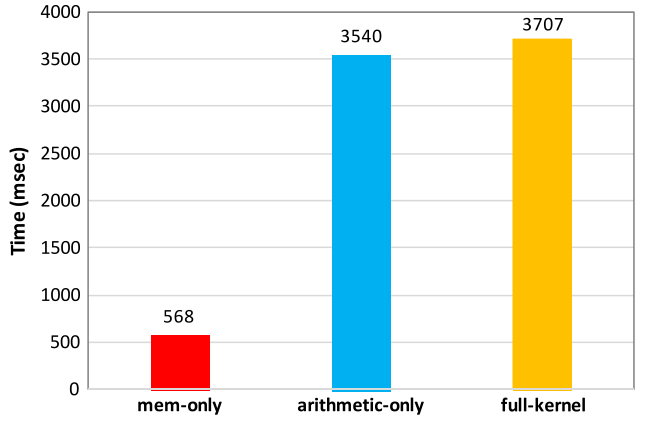


Fig. 6. Memory-only, arithmetic-only, and full-kernel time in GPU-AxT on a system with 3375 particles. 70.6% of memory operations are overlapped with arithmetic operations.

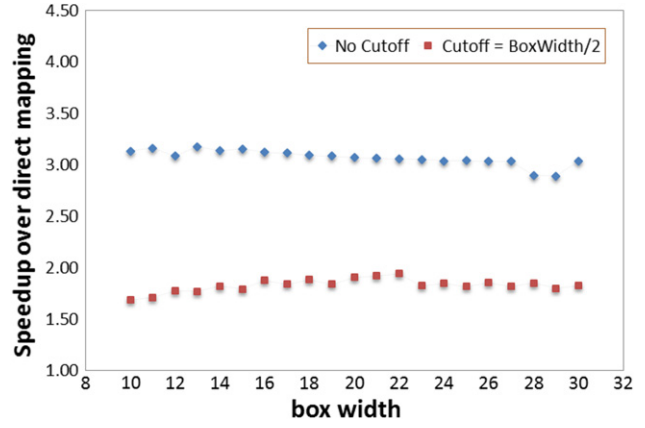


Fig. 7. Speedup of GPU-AxT Implementation using load-balancing scheme over that of direct mapping. When no cutoff distance is applied, speedup is approximately 3, agreeing well with the theoretical analysis. When half box width cutoff is adopted, speedup is reduced to ∼1.8.

program. Evaluation of the Axilrod–Teller potential with distance cutoff requires testing pairwise particle distances—if any one of the pairwise particle distances is higher than the cutoff distance, the three-body interaction computation will not be carried out. When the threads handling three-body interactions within the distance cutoff as well as those exceeding cutoff co-reside in the same GPU warp, divergent branches will occur in runtime.

Table 2
Comparison of the number of divergent branches in GPU-AxT with half box cutoff distance and GPU-AxT without cutoff.

Box width	# of branch instruction		# of divergent branch	
	No cutoff	w. cutoff	No cutoff	w. cutoff
10	2,628	6,722	0	1,319
12	7,817	16,954	0	1,338
14	39,338	91,987	0	11,766
16	87,558	207,909	0	27,614
18	266,098	640,687	0	89,429
20	500,511	1,182,504	0	152,506
22	1,181,940	2,858,410	0	435,072
24	2,489,780	5,809,260	0	730,359
26	4,829,030	11,573,100	0	1,635,830
28	8,787,680	20,750,600	0	2,717,490
30	15,192,000	36,138,900	0	5,058,260

Table 2 compares the number of divergent branches in GPU-AxT with half box width cutoff with GPU-AxT without distance cutoff. The performance data is obtained by NVIDIA Compute Visual Profiler 3.2 [8]. When no distance cutoff is adopted, GPU-AxT does not suffer from branch divergence because pairwise particle distances are not necessarily checked against cutoff distance. In contrast, when distance cutoff is applied, branch instructions (if statements) are inserted to compare the pairwise particle distances with the cutoff distance, which potentially leads to divergent branches. When half box width cutoff is used, the divergent branches in GPU-AxT are approximately 15% of the total number of branch instructions, which results in speedup reduction.

3.1.4. Applications in Monte Carlo and molecular dynamics simulation

We apply GPU-AxT to a Monte Carlo sampling program and a molecular dynamics program for simulating Argon gas systems to demonstrate the effectiveness of GPU-accelerated three-body calculations. The measured computation time is the overall application execution time, which includes the CPU time, the GPU time, and the data transferring time between host memory and device memory.

The Monte Carlo simulation is carried out by using Cartesian all atoms move where the position of every particle in the system is changed by a small random perturbation during a Monte Carlo trial. The Monte Carlo sampling program employs the Metropolis algorithm [28] and the Axilrod–Teller potential energy is evaluated in every iteration step to determine the acceptance of the proposed new conformation. Adopting GPU-AxT, the Monte Carlo sampling program is implemented as a heterogeneous CPU–GPU program. The evaluation of the Axilrod–Teller potential energy is carried out on the GPU while the rest of the Monte Carlo computations are executed on the CPU. We execute the Monte Carlo program using GPU-AxT on a Tesla C2070 simulating a system with 3375 particles. The computation time is averaged over 1000 Monte Carlo iteration steps. Compared to that of the direct-mapping implementation (10.94 s), the computation time per step in the Monte Carlo program using load-balancing GPU-AxT is reduced to 3.68 s—a 2.97 speedup, when no distance cutoff is applied. When half box width distance cutoff is used, computation time per step in the Monte Carlo program using load-balancing GPU-AxT is 0.89 s, which is 1.81 times faster than the direct-mapping implementation (1.62 s).

The implementation of the molecular dynamics program is based on the pseudocode described in [23], which is tuned and yields 10 times faster than normal implementation according to [23]. The Axilrod–Teller potential is used to estimate the van der Waals interactions between three particles. The main difference compared to the Monte Carlo program, where only the overall potential energy value is needed, is that the forces acting on all particles need to be calculated at each iteration step. In our

GPU-AxT implementation for molecular dynamics simulation, we maintain arrays of velocities and accelerations for particles in the system. In each three-body interaction calculation in GPU-AxT, the accelerations of the three participating particles are calculated as the derivative of their partial potential energy with respect to their positions and are integrated accordingly. Once all interaction calculations are completed in GPU-AxT, the new velocity and position of each particle are then evaluated according to its acceleration.

The molecular dynamic simulation program is also implemented as a heterogeneous CPU–GPU program, where the particle interaction calculations are carried out on the GPU while the rest of the program is executed on the CPU. We carried out the program on a system with 3375 particles and the computation time is recorded for 1000 iterations. When no distance cutoff is applied, the overall computation time of the molecular dynamic program using load-balancing GPU-AxT is 11.57 h, which is about 2.9 times faster than the direct-mapping implementation (33.06 h). When half box width distance cutoff is used, the overall computation time using load-balancing GPU-AxT becomes 1.91 h, which is a ∼1.5 speedup over the direct-mapping implementation (2.77 h).

The computational results of the Monte Carlo and molecular dynamics programs are consistent with the analysis in Section 3.1.3. It is important to note that in this work we only consider how the GPU acceleration in three-body interaction calculation using Axilrod–Teller potential can affect the overall performance of the simulation programs. In fact, if more parallel computation components, e.g., the proposal function of generating new particle positions in Monte Carlo or calculations of the new particle velocities and positions in molecular dynamics, are moved to the GPU, more significant performance improvements are possible.

3.2. Computational results of context-based secondary structure potential (CSSP)

3.2.1. Context-based secondary structure potential (CSSP)

CSSP is a statistical potential integrating inter-residue interaction potentials for assessing the quality of predicted protein secondary structures [21]. Considering a protein chain with L amino acid residues and a fragment of size S where $S < L$, the CSSP potential of a protein molecule is calculated as,

$$U_{protein} = \sum_{i=0}^{L-S+1} \left(\sum_{i}^{S} U(R_i) + \sum_{i\neq j}^{S} U\left(R_i, R_j\right) + \sum_{i\neq j\neq k}^{S} U\left(R_i, R_j, R_k\right) \right) - \sum_{i=1}^{L-S-1} \left(\sum_{i}^{S-1} U\left(R_i\right) + \sum_{i\neq j}^{S-1} U\left(R_i, R_j\right) + \sum_{i\neq j\neq k}^{S-1} U\left(R_i, R_j, R_k\right) \right),$$

where R_i denotes residue i in the protein chain and $U(R_i)$, $U\left(R_i, R_j\right)$, and $U\left(R_i, R_j, R_k\right)$ are singlet, doublet, and triplet potential terms, respectively.

3.2.2. Performance of load-balancing scheme

Unlike the Axilrod–Teller potential, which is a pure three-body potential, the CSSP potential includes three-body terms together with two- and single-body terms. In GPU-CSSP implementation, the single-body terms are calculated using direct mapping and the two-body terms are calculated using the pairwise load-balancing scheme described in [35], which has a theoretical speedup of ∼2.0 over direct mapping scheme. The three-body terms are calculated using the load-balancing workload distribution scheme described in this paper with theoretical speedup of ∼3.0 over direct mapping.

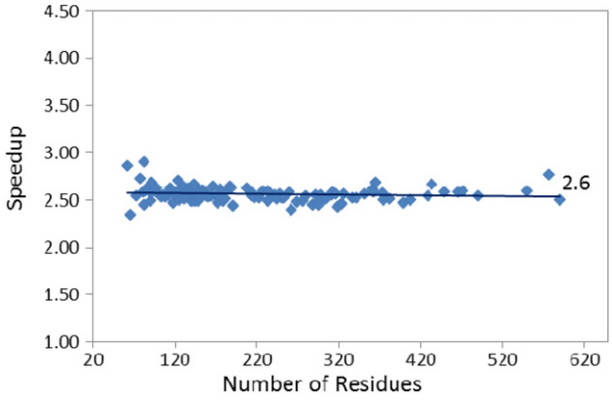


Fig. 8. Performance of the balanced GPU-CSSP with respect to the unbalanced GPU-CSSP on Tesla C2070.

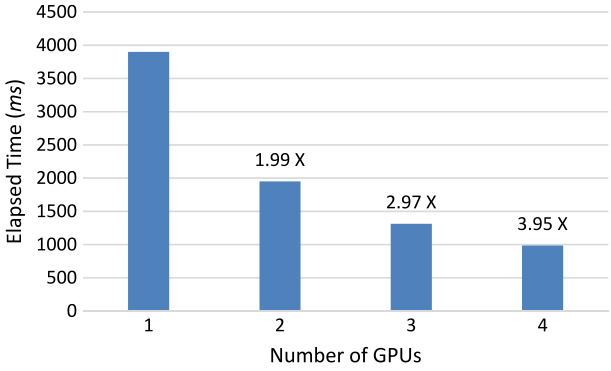


Fig. 9. Computational time and speedup of GPU-AxT implementation using multiple Tesla C2070 GPUs on a molecular system with 8000 particles. The elapsed time is the average iteration time over 100 runs.

Fig. 8 shows the speedup of the load-balancing GPU-CSSP over that of direct-mapping on a set of proteins ranged from tens to hundreds of residues, where an average speedup of 2.6 is obtained. This speedup is consistent for small and large proteins.

3.3. Scalability of the load balancing scheme across multiple GPUs

The load-balancing workload scheme can be smoothly scaled across multiple GPUs. Based on the fact that the three-body interaction computations of triplets in each thread are independent, we divide the computation work of each thread into approximately even n_{GPU} partitions, where n_{GPU} is the number of GPUs available, so that each GPU can carry out one partition simultaneously. Once the computations on multiple GPUs are completed, partial sums of the results from each GPU are collected and accumulated. For large-scale three-body computations, the GPU-AxT implementation is compute bound, as memory transaction cost is negligible compared to that of the GPU computations for three-body interaction.

Fig. 9 shows that the elapsed time of executing GPU-AxT on a system with 8000 particles using multiple Tesla C2070 GPUs. One can find that the GPU-AxT implementation can achieve nearly linear speedup by using multiple GPUs over the case with a single GPU.

4. Conclusions

In this paper, we investigate the approaches of using GPU to accelerate calculation of general three-body potential energy functions for molecular simulation applications. A workload distribution scheme is developed to achieve perfect load-balancing when N is indivisible by 3 and nearly perfect load-balancing

otherwise. The load-balancing workload distribution scheme has demonstrated its effectiveness in the GPU-implementations of the Axilrod–Teller potential and the CSSP potential where three-body terms are incorporated.

The new features of latest GPU architectures have the potential to further improve the performance of our three-body energy potential implementations on GPU. For example, the new shuffle (SHFL) instruction of the Kepler GPU architecture allows parallel threads to access data from other threads directly, which enable triplets to be shuffled among threads in a warp and thus enhance parallel efficiency. This will be one of the research directions in our future work.

When a distance cutoff is used, the emergence of potential divergent branches hurts the efficiency of the load-balancing scheme for three-body interaction computation, as shown in Section 3.1.3. Data structures such as quadtree in 2D [27] or octree in 3D [5], may be used to reduce the number of divergent branches and thus enhance computational efficiency, due to the fact that three-body interactions are only needed to be computed in the pre-computed space subdivision. With the cost of auxiliary memory [13], the three-body interaction computation may be accelerated significantly. We will also pursue this direction in our future implementations.

Unfortunately, extending this load-balancing workload distribution scheme to four-body potential calculation on GPUs is not straightforward. The position separation patterns in four-body term calculation are more complicated than those of three-body due to order two rotational symmetry in position separation patterns when four particles are involved. Order two rotational symmetry does not exist in three-body term calculation and requires deliberate analysis. Nonetheless, in general, the computational cost of four-body or higher order terms are much higher than that of three-body ones while the effectiveness of four-body or higher order terms still need theoretical justifications. Therefore, at this moment, there is no immediate urgency in extending the load-balancing workload distribution scheme for general four-body or higher order terms calculations on GPUs.

Acknowledgments

We would like to thank the reviewers for their suggestions that greatly help us improve the contents of the manuscript. Yaohang Li acknowledges support from NSF grant 1066471. Hao Ji acknowledges support from ODU Modeling and Simulation Fellowship.

References

[1] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Comput. Phys. 227 (10) (2008) 5342–5359.
[2] J.A. Anta, E. Lomba, M. Lombardero, Exploring the influence of three-body classical dispersion forces on phase equilibria of simple fluids: An integral-equation approach, Phys. Rev. E (3) 49 (1) (1994) 402–409.
[3] B.M. Axilrod, E. Teller, Interaction of the van der Waals type between three atoms, J. Chem. Phys. 11 (6) (1943) 299–300.
[4] J. Barnes, P. Hut, A hierarchical O(N-Log-N) force-calculation algorithm, Nature 324 (6096) (1986) 446–449.
[5] J. Bedorf, E. Gaburov, S.P. Zwart, A sparse octree gravitational Nbody code that runs entirely on the GPU processor, J. Comput. Phys. 231 (7) (2012) 2825–2839.
[6] R.G. Belleman, J. Bedorf, S.F.P. Zwart, High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA, New Astron. 13 (2) (2008) 103–112.
[7] R.B. Best, J. Mittal, M. Feig, A.D. MacKerell Jr., Inclusion of many-body effects in the additive CHARMM protein CMAP potential results in enhanced cooperativity of alpha-helix and beta-hairpin formation, Biophys. J. 103 (5) (2012) 1045–1051.
[8] Compute Visual Profiler 3.2 [http://www.developer.download.nvidia.com].
[9] T. Darden, D. York, L. Pedersen, Particle mesh Ewald—an N. Log(N) method for Ewald sums in large systems, J. Chem. Phys. 98 (12) (1993) 10089–10092.
[10] M.J. Elrod, R.J. Saykally, Many-body effects in intermolecular forces, Chem. Rev. 94 (7) (1994) 1975–1997.
[11] Y. Feng, A. Kloczkowski, R.L. Jernigan, Four-body contact potentials derived from two protein datasets to discriminate native structures from decoys, Proteins 68 (1) (2007) 57–66.
[12] M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, C.M. Bruns, V.S. Pande, Accelerating molecular dynamic simulation on graphics processing units, J. Comput. Chem. 30 (6) (2009) 864–872.
[13] I. Gargantini, An effective way to represent quadtrees, Commun. ACM 25 (12) (1982) 905–910.
[14] L. Greengard, The rapid evaluation of potential fields in particle systems (Thesis (doctoral)) MIT Press, Yale University, Cambridge, Mass, 1988.
[15] S.L. Grimm, J.G. Stadel, The genga code: Gravitational encounters in N-body simulations with GPU acceleration, Astrophys. J. 796 (1) (2014).
[16] T. Hamada, R. Yokota, K. Nitadori, T. Narumi, K. Yasuoka, M. Taiji, 42 TFlops hierarchical N-body simulations on GPUs with applications in both astrophysics and turbulence, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009.
[17] R.W. Hockney, J.W. Eastwood, Computer Simulation Using Particles, A. Hilger, Bristol, England, Philadelphia, 1988, Special student edn..
[18] P. Jetley, L. Wesolowski, F. Gioachin, L.V. Kale, T.R. Quinn, Scaling hierarchical N-body simulations on GPU clusters, in: High Performance Computing, Networking, Storage and Analysis SC, 2010 International Conference for: 13–19 November 2010, Vol. 2010, pp. 1–11.
[19] M.H. Lars Nyland, Jan Prins, Fast N-Body simulation with CUDA, in: GPU Gems 3, NVIDIA, 2007.
[20] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L.X. Ying, D. Zorin, G. Biros, A massively parallel adaptive fast multipole method on heterogeneous architectures, Commun. ACM 55 (5) (2012) 101–109.
[21] Y. Li, H. Liu, I. Rata, E. Jakobsson, Building a knowledge-based statistical potential by capturing high-order inter-residue interactions and its applications in protein secondary structure assessment, J. Chem. Inf. Model. 53 (2) (2013) 500–508.
[22] Y. Li, W. Zhu, GPU-accelerated multi-scoring functions protein loop structure modeling, in: 9th IEEE International Workshop on High Performance Computational Biology, 2010.
[23] G. Marcelli, The role of three-body interactions on the equilibrium and non-equilibrium properties of fluids from molecular simulation. Swinburne University of Technology Dissertation, 2001.
[24] G. Marcelli, R.J. Sadus, Molecular simulation of the phase behavior of noble gases using accurate two-body and three-body intermolecular potentials, J. Chem. Phys. 111 (4) (1999) 1533–1540.
[25] G. Marcelli, R.J. Sadus, A link between the two-body and three-body interaction energies of fluids from molecular simulation, J. Chem. Phys. 112 (14) (2000) 6382–6385.
[26] G. Marcelli, B.D. Todd, R.J. Sadus, Beyond traditional effective intermolecular potentials and pairwise interactions in molecular simulation, in: Computational Science-Iccs 2002, Pt III, Proceedings, Vol. 2331, 2002, pp. 932–941.
[27] P. Mazumder, Planar decomposition for quadtree data structure, Comput. Vis. Graph. Image Process. 38 (1987) 258–274.
[28] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines, J. Chem. Phys. 21 (6) (1953) 1087–1092.
[29] NVIDIA: CUDA programming guide version 3.1. In.; 2010.
[30] NVIDIA [http://www.nvidia.com/page/home.html].
[31] M.J. Stock, A. Gharakhani, Toward efficient GPU-accelerated N-body simulations, in: 46th AIAA Aerospace Sciences Meeting and Exhibit, 2008.
[32] L. Wang, R.J. Sadus, Effect of three-body interactions on the vapor–liquid phase equilibria of binary fluid mixtures, J. Chem. Phys. 125 (7) (2006) 074503.
[33] L. Wang, R.J. Sadus, Influence of two-body and three-body interatomic forces on gas, liquid, and solid phases, Phys. Rev. E (3) 74 (2 Pt 1) (2006) 021202.
[34] L. Wang, R.J. Sadus, Three-body interactions and solid–liquid phase equilibria: application of a molecular dynamics algorithm, Phys. Rev. E (3) 74 (3 Pt 1) (2006) 031203.
[35] A. Yaseen, Y. Li, Accelerating knowledge-based energy evaluation in protein structure modeling with graphics processing units, J. Parallel Distrib. Comput. 72 (2) (2012) 297–307.
[36] A. Yaseen, Y. Li, Context-based features enhance protein secondary structure prediction accuracy, J. Chem. Inf. Model. 54 (3) (2014) 992–1002.
[37] G. Zhao, M.B. Carson, H. Lu, Prediction of specific protein–DNA recognition by knowledge-based two-body and three-body interaction potentials. in: Conf. Proc. IEEE Eng. Med. Biol. Soc., Vol. 2007, 2007, pp. 5017–5020.
[38] W. Zhu, A. Yaseen, Y. Li, DEMCMC-GPU: An efficient multi-objective optimization method with GPU acceleration on the fermi architecture, New Gener. Comput. 29 (2) (2011) 163–184.

Dr. Ashraf Yaseen is an Assistant Professor in the Department of Electrical Engineering and Computer Science at Texas A&M University at Kingsville. He received his B.S. degree in CS from Jordan University of Science and Technology in 2002, his M.S. degree in CS from New York Institute of Technology in 2003, and his Ph.D. degree in CS from Old Dominion University in 2014. His research interests include Computational Biology and High Performance Computing.

Hao Ji is a Ph.D. student in the Department of Computer Science at Old Dominion University, Norfolk, VA, USA. He received the B.S. degree in Applied Mathematics and M.S. degree in Computer Science from Hefei University of Technology, Hefei, China, in 2007 and 2010, respectively. His research interest include Monte Carlo Methods for Big Data Analysis, Large-Scale Linear Algebra, and High Performance Scientific Computing.

Dr. Yaohang Li is an Associate Professor in the Department of Computer Science at Old Dominion University. His research interests are in Computational Biology, Monte Carlo Methods, and Scientific Computing. He received the Ph.D. and M.S. degrees in Computer Science from the Florida State University in 2003 and 2000, respectively. After graduation, he worked at Oak Ridge National Laboratory as a research associate for a short period of time. Before joining ODU, he was an associate professor in the Computer Science Department at North Carolina A&T State University.