Accelerating knowledge-based energy evaluation in protein structure modeling with Graphics Processing Units

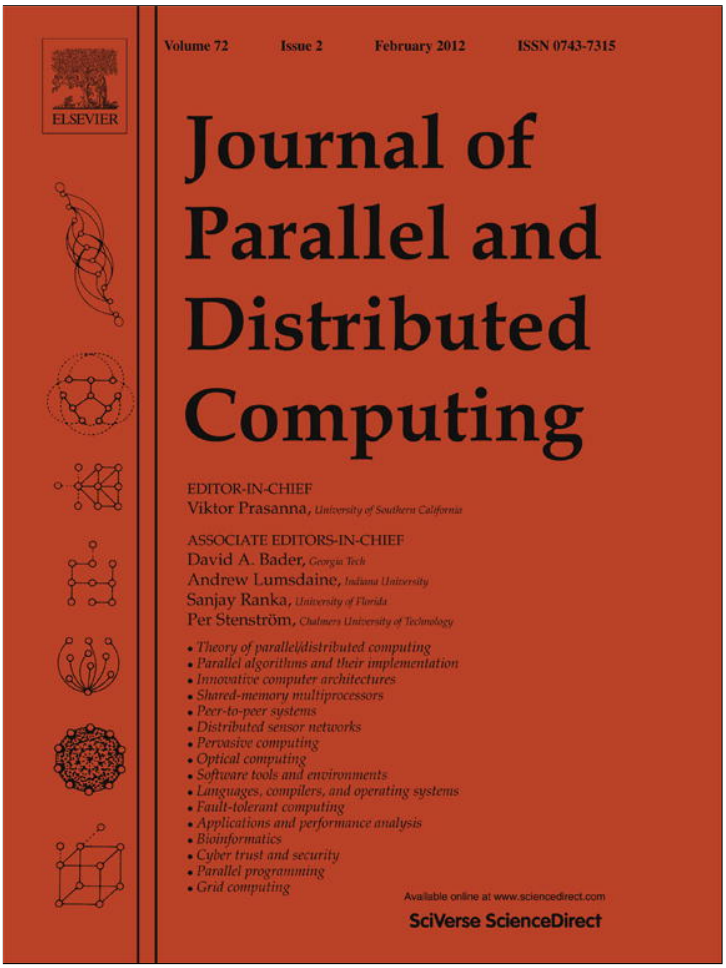
ARTICLE *in* JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING · FEBRUARY 2012

Impact Factor: 1.18 · DOI: 10.1016/j.jpdc.2011.10.005 · Source: DBLP

2 AUTHORS:

Ashraf Yaseen
Old Dominion University
8 PUBLICATIONS **21** CITATIONS
SEE PROFILE

Yaohang Li
Old Dominion University
75 PUBLICATIONS **338** CITATIONS
SEE PROFILE



J. Parallel Distrib. Comput. 72 (2012) 297–307

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

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

Accelerating knowledge-based energy evaluation in protein structure modeling with Graphics Processing Units

Ashraf Yaseen, Yaohang Li *

Department of Computer Science, Old Dominion University, Norfolk, VA 23529, United States

ARTICLE INFO

Article history:
Received 26 June 2011
Received in revised form
10 October 2011
Accepted 17 October 2011
Available online 24 October 2011

Keywords:
Knowledge-based Energy
Protein modeling
GPU
Symmetric *N*-body Problem

ABSTRACT

Evaluating the energy of a protein molecule is one of the most computationally costly operations in many protein structure modeling applications. In this paper, we present an efficient implementation of knowledge-based energy functions by taking advantage of the recent Graphics Processing Unit (GPU) architectures. We use DFIRE, a knowledge-based all-atom potential, as an example to demonstrate our GPU implementations on the latest NVIDIA Fermi architecture. A load balancing workload distribution scheme is designed to assign computations of pair-wise atom interactions to threads to achieve perfect or near-perfect load balancing in the symmetric *N*-body problem in DFIRE. Reorganizing atoms in the protein also improves the cache efficiency in Fermi GPU architecture, which is particularly effective for small proteins. Our DFIRE implementation on GPU (GPU-DFIRE) has exhibited a speedup of up to ~150 on NVIDIA Quadro FX3800M and ~250 on NVIDIA Tesla M2050 compared to the serial DFIRE implementation on CPU. Furthermore, we show that protein structure modeling applications, including a Monte Carlo sampling program and a local optimization program, can benefit from GPU-DFIRE with little programming modification but significant computational performance improvement.

1. Introduction

Modeling structures of protein or protein complex with high resolution is considered one of the most significant grand challenges in computational biology. One of the biggest obstacles in computational protein modeling is its heavy computational cost, particularly for relatively large proteins. It is very common that the computation time of a protein structure modeling program ranges from several hours to several days or even longer. Many protein structure modeling applications can greatly benefit from significant reduction of computation time by enabling one to sample broader conformation space to discover the appropriate structures, execute more simulation steps to obtain models with better accuracy, or carry out simulation on much larger proteins.

The most costly operations, in many protein modeling applications, including homology modeling [25], template-based modeling [42], *ab initio* modeling [4], side chain prediction [17], and loop structure prediction [20], are usually energy evaluations of protein molecules. The protein energy is usually determined by the physics-based energy functions or the knowledge-based energy functions. In this paper, we focus on the knowledge-based energy functions, whose terms are derived from large protein databases by statistical methods and usually occupy large memory space. Reducing the energy evaluation time is the key to accelerate many protein modeling programs. The major part of most protein energy evaluation involves estimating interactions between pair-wise atoms, which is typically an *N*-body problem. For a protein system involving N atoms, conventional evaluation of the potential energy function by estimating every pair-wise interactions requires $O(N^2)$ operations. As a result, for a relatively large protein conformation, calculation of all pairs of interactions requires substantial computational time. By simplifying the long distance interactions, fast *N*-body algorithms, including the Barnes–Hut method [2], the fast multipole method [12], and the particle–mesh methods [14,9], can reduce the computational complexity to $O(N \log N)$ or $O(N)$.

Recent developments in Graphical Processing Units (GPUs) with massively parallel computing mechanism offer attractive opportunities to take advantage of parallelism at the chip level to achieve high performance computing. Being able to simultaneously calculate forces on N particles, GPU has been employed to accelerate the *N*-body simulation in a variety of applications. For example, Nyland et al. [29] developed fast *N*-body simulation with CUDA in astrophysical applications. Stock and Gharakhani [37] introduced an efficient multipole-accelerated treecode method for turbulent flow computations. Anderson et al. [1] carried out general molecular dynamics by simulating N particles in a finite box with periodic boundary conditions. Belleman et al. [3] implemented direct gravitational *N*-body simulation using CUDA programming toolkits [28]. Jetley et al. [15] performed hierarchical *N*-body simulations on GPU-based clusters. Significant speedups have been observed in the above applications.

* Corresponding author.
E-mail addresses: ayaseen@cs.odu.edu (A. Yaseen), yaohang@cs.odu.edu (Y. Li).

0743-7315/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2011.10.005

In this paper, we present an approach to accelerate the calculations of knowledge-based energy functions popularly used in computational protein structure modeling by taking advantage of the GPU architecture. We use an all-atom knowledge-based energy function, DFIRE [43], as an example to illustrate our GPU implementation. Our key contribution is the design of a workload distribution scheme to achieve perfect or nearly perfect load balancing among GPU threads in the symmetric N-body problem. Moreover, unlike many N-body simulations [29,37,1,3,15] where a large number of floating point operations are involved, the evaluation of knowledge-based energy functions exhibits a different computing pattern with few floating-point operations but intensive memory accesses instead. Accordingly, we reorder the protein atom sequence by types to improve cache efficiency in latest NVIDIA Fermi GPU architecture. We name the GPU implementation of DFIRE energy function "GPU–DFIRE" while the original serial CPU version is referred to as "CPU–DFIRE". GPU–DFIRE is implemented on the recent Fermi architecture using CUDA programming environment [28]. We also compare the performance of GPU–DFIRE using pair-wise interactions methods with cell subdivision method and neighbor lists method. A Monte Carlo sampling program and a local optimization program are used to demonstrate the efficiency of GPU–DFIRE in all-atom protein structure modeling.

2. Knowledge-based energy

In protein structure modeling, the physics-based energy functions intend to evaluate the protein molecule energy by describing atomic interactions with physics-based energy terms. Popular physics-based energy functions include CHARMM [5], AMBER [24], OPLS [8], GROMOS [16], etc. A typical physics-based energy function consists of bonded terms and non-bonded terms. The bonded terms, including bond energy terms, bond angle energy terms, and torsion energy terms, apply to a set of atoms (2–4) that are linked with covalent bonds. The non-bonded terms consist of van der Waals interactions and electrostatic interactions. Additional terms such as hydrogen bonding, hydrophobic, and solvation are often incorporated in various physics-based energy functions. Physics-based energy functions have been used predominately to simulate protein dynamics.

Alternatively, the knowledge-based (statistical) energy functions are developed to estimate the feasibility of protein conformations instead of the true physical energy. Knowledge-based approaches derive rules from the increasing number of experimentally determined protein conformations by statistical approaches. These rules are converted into "pseudo-potential" energy (scoring) functions based on the inverse Boltzmann law. There exist numerous knowledge-based energy functions based on various aspects or data sets. Examples of knowledge-based energy functions include DFIRE [43], DOPE [30], OPUS [36], I-TASSER [23], and many others. Compared to the physics-based energy functions, the knowledge-based energy functions yield much smoother energy landscape, which can facilitate the search process to locate the most reasonable protein conformations. As a result, the knowledge-based energy functions have been successfully used in a variety of protein structure modeling and protein design [35] applications. For example, in the recent CASP (Critical Assessment of protein Structure Prediction) experiments, knowledge-based potentials have been widely used in top-ranked prediction servers, such as QUACK [39], RAPTOR [32], and pro-sp3-TASSER [44], for conformation sampling, model assessment, or structure refinement.

Unlike the physics-based energy functions computations which require intensive floating-point operations, the knowledge-based energy functions usually demand large memory space instead. Evaluation of a knowledge-based energy function needs to search a large memory space to retrieve the appropriate values, which can be regarded as mainly memory-intensive computation. Table 1 shows the performance comparison between AMBER99 [24] (a physics-based energy) and DFIRE [43] (a knowledge-based energy). One can find that compared to physics-based AMBER99, the knowledge-based DFIRE has lower density of floating point operations but higher frequency of memory access. Moreover, due to frequent searching of a large DFIRE table in the memory, DFIRE has a significantly higher L2 cache miss ratio than AMBER99. In this paper, we focus on accelerating the computation of knowledge-based energy functions by taking advantage of the GPU architectures. Actually, very often, to avoid heavy runtime floating-point computations, some terms in the physics-based energy function are pre-calculated and stored into a table. Hence, the physics-based energy function evaluation may also be memory-intensive [11]. Therefore, in this case, the approaches developed in this paper may also be applicable to some physics-based energy functions.

Table 1
Performance comparison between AMBER99 (a physics-based energy) and DFIRE (a knowledge-based energy) on a set of proteins of various sizes. The benchmark computation is carried out exclusively on a server with a 2.6 GHz Intel Pentium Pro (P6) processor, 4 GB memory, and 256 K L2 cache. The application performance data are obtained by PAPI [6] and Perfsuite [18], which are performance tuning software packages based on hardware performance counters.

	AMBER99	DFIRE
Floating point operations per cycle	0.390	0.133
L2 data cache access per instruction	0.0126	0.0285
L2 cache miss ratio	0.0330	0.176

3. GPU–DFIRE implementation details

In this paper, we use DFIRE [43] potential energy function as an example to illustrate our GPU implementation of memory-intensive knowledge-based energy functions. The GPU implementation of DFIRE (GPU–DFIRE) can be adopted into a variety of protein modeling algorithms, such as Monte Carlo (MC) methods [31], Local Energy Minimization [27], Molecular Dynamics [19], Genetic Algorithms (GA) [10], Evolutionary Computing (EC) [33], etc., where repeatedly assessing the potential energy of protein conformations is required.

3.1. DFIRE potential

DFIRE is an all-atom potential energy function derived from "the structures of single-chain proteins by using a physical state of uniformly distributed points in finite spheres as the zero-interaction reference state" [22]. A large three-dimensional DFIRE array (first atom type, second atom type, and distance bin) is used to store the statistical potential energy values of each possible atom pair based on their Euclidean distance. In this paper, we consider the symmetric version of DFIRE, which has demonstrated better accuracy than the asymmetric one [45]. In symmetric DFIRE, for a protein sequence starting from N-terminal to C-terminal,

$$\mathrm{DFIRE}(\mathrm{ATOMS}[i], \mathrm{ATOMS}[j], d) = \mathrm{DFIRE}(\mathrm{ATOMS}[j], \mathrm{ATOMS}[i], d),$$

for atom pair i and j in distance d and ATOMS[i] denotes the atom information of the ith atom in the protein. The DFIRE program takes the protein PDB file as input and parses it into the ATOMS array, including atom type in DFIRE,[1] sequence number, and spatial coordinate values (XYZ).

[1] DFIRE classifies possible protein atoms into 167 types [45].

```
calcDFIRE(atomi, atomj, d)
{
      if (d < CUTOFF)
            return (DFIRE[atomi.DFIREtype, atomj.DFIREtype, d]);
      else
            return 0.0;
}
```

Fig. 1. DFIRE subroutine for pair-wise atom interaction calculation.

The computation of DFIRE energy is a near N-body calculation. Starting from the first atom in the ATOMS array, for every atom in the protein, the DFIRE program retrieves the energy terms between the current atom and the rest of the atoms not in the same residue. The energy term is obtained by calculating the pair-wise atom distance, converting it into a distance bin, and then looking up the large DFIRE array for the appropriate energy term value. DFIRE calculation has a distance cutoff—if the distance between two atoms is bigger than the cutoff, the interaction between these two atoms is deemed to be small enough to be ignored. Fig. 1 shows the pseudocode of DFIRE subroutine (calcDFIRE) for pair-wise atom interaction calculation. Finally, all energy terms are accumulated to generate the overall DFIRE potential energy value. The major operations in calculating DFIRE energy are memory accesses, i.e., looking up the large DFIRE array for every atom pair.

3.2. Symmetric N-body problem

The serial calculation of symmetric N-body interaction in DFIRE is straightforward, whose pseudo code is shown in Fig. 3. However, unlike the asymmetric N-body problem shown in Fig. 2 where exactly $N-1$ interactions are calculated for each atom, for each atom in the outer loop in the symmetric N-body problem, its number of atom–atom interaction calculations in the inner loop varies gradually from $N-1$ to 1. Consequently, directly mapping the calculations in the inner loop to GPU threads will lead to unbalanced workload distribution.

To balance workload distribution among GPU threads, we design the following novel load assignment scheme. For simplicity in illustration, we assume the one-thread-per-atom assignment. The pseudocode of workload assignment for a thread is shown in Fig. 4. For each thread i, it carries out M atom–atom interaction calculations between atom i and atoms $(i+1) \bmod N$, $(i+2) \bmod N, \ldots, (i+M) \bmod N$. If N is an odd number, M is $(N-1)/2$ and perfect load balancing can be obtained as illustrated in Fig. 5. If N is an even number, for the first $N/2$ threads, M is $N/2$ and for the second $N/2$ threads, M is $N/2-1$. Fig. 6 shows the thread load distribution for an even N, where nearly perfect load balancing can be achieved—the first half of the threads carry out one more atom–atom interaction calculation each than the second half of the threads. In addition to load balancing, the load assignment scheme is particularly suitable for the GPU's Single Instruction Multiple Threads (SIMT) architecture [28]. This is due to the fact that, at each iteration step, each thread reads data from different atoms with the same stride, which can be coalesced into efficient memory transactions.

A straightforward way to handle the symmetric N-body problem is to directly use the asymmetric N-body scheme shown in Fig. 2, i.e., each thread calculates the interactions of one atom with the rest of the atoms and then the half of the overall energy sum gives the total DFIRE potential energy. However, compared to our symmetric scheme, the asymmetric scheme doubles pair-wise interaction calculations. As a result, by taking advantage of the symmetricity of the problem, the computational time using our workload assignment scheme in Tesla 2050M is ∼35% less than that of the asymmetric scheme, consistently in both small and large proteins.

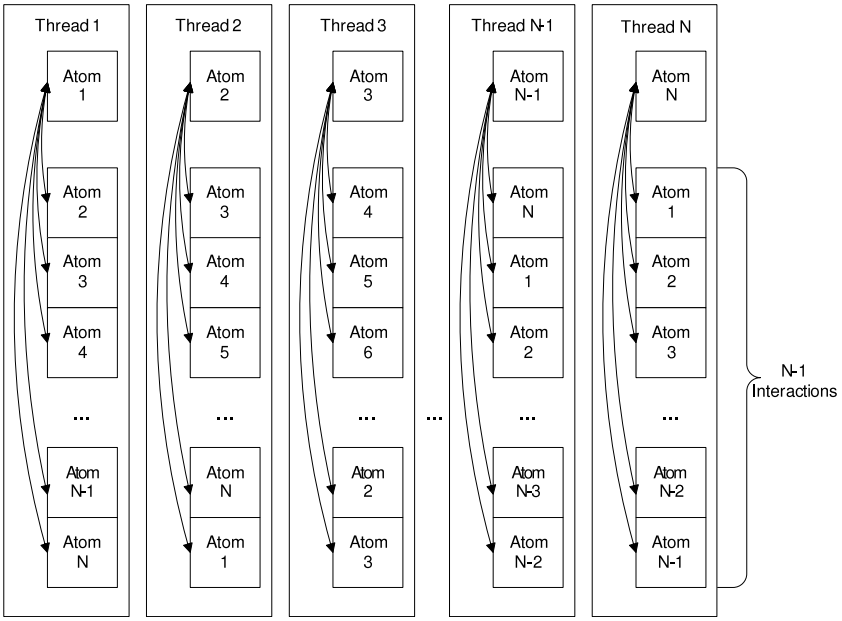


Fig. 2. GPU implementation of asymmetric N-body problem.

```
score ← 0.0;                          // initialization
for i ← 1 to N-1                      // outer loop
   for j ← i+1 to N {                 // inner loop
      d ← dist(ATOMS[i], ATOMS[j]); // distance bin btw. atoms i, j
      score ← score + calcDFIRE(ATOMS[i], ATOMS[j], d);
   }
```

Fig. 3. Pseudocode of symmetric N-body calculation in serial DFIRE.

```
Thread(i)
{
   score ← 0.0;                          // initialization
   if (N mod 2 == 0 AND i > N/2) {       // even number and second half
      count ← N/2 - 1;                   // number of interactions (this thread)
   }
   else {                                // odd number or even number/first half
      count ← N/2;                       // number of interactions (this thread)
   }
   for k ← 1 to count {                  // atom-atom calculation loop
      j ← (i + k) mod N;                 // next atom number
      d ← dist(ATOMS[i], ATOMS[j]);      // distance bin btw. atoms i, j
      score ← score + calcDFIRE(ATOMS[i], ATOMS[j], d);
   }
   local_sum_reduce();                   // sum partial scores in local block
}
```

Fig. 4. Pseudocode of assigning workload to a GPU thread in GPU–DFIRE.

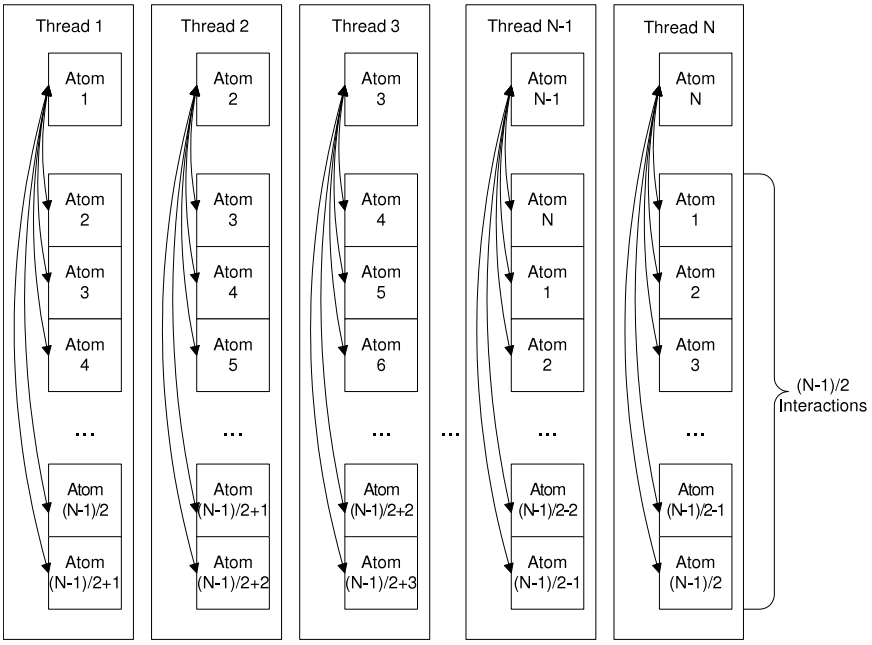


Fig. 5. Perfect balancing when N is an odd number. Each thread carries out $(N-1)/2$ atom–atom iterations.

The pseudocode in Fig. 4 is efficient in calculating the overall DFIRE potential energy of a protein molecule. However, some applications require calculations of all interactions acting on every atom of the protein. For example, a Molecular Dynamics (MD) [19] program needs to calculate the potential energy of an atom at its position in order to evaluate force exerted on it. To calculate DFIRE potential energy of each atom in the protein molecule, the asymmetric scheme shown in Fig. 2 is needed instead.

The above workload distribution is based on the one-thread-per-atom assumption; however, for small proteins, the above load assignment scheme with N threads for N atoms may not produce enough threads to fully utilize the power of the GPU architecture. To address this issue in small proteins, we implement fine-grained threads by continuously dividing the workload originally assigned to one thread to multiple threads so that sufficient threads are spawned to effectively mask the memory access latency. More discussion on fine-grained threads can be found in Section 3.4.

3.3. Cache efficiency

In the latest NVIDIA GPU Fermi architecture [40], the device memory is cached with L1/L2 cache. L1 cache is shared in one multiprocessor while L2 cache is shared among all multiprocessors. Correctly organizing data can take advantage of the cache coherence in GPU.

Generally, atoms in a PDB file are grouped by their residues in the protein. To take advantage of the GPU cache, in our GPU–DFIRE implementation, we reorder atom sequence in a protein according to their types. Atom sequence sorting is a one-time cost in GPU–DFIRE. Fig. 7 shows the sorting of a 6-residue fragment of protein, where atoms of the same types are clustered after reordering. Clustering atoms of the same types together can potentially improve the cache hit rate. Another advantage of reordering the atom sequence by atom types is, in case of cache misses, the requested global memory addresses will have a good chance to fall within fewer cache-lines compared to unsorted atom sequence, which can lead to better bus utilization.

Table 2 compares the GPU–DFIRE performance using sorted and unsorted atom sequences. The performance data is obtained by NVIDIA Compute Visual Profiler 3.2 [34] on an NVIDIA M2050 (Fermi architecture). One can find that when the atom sequences are sorted by the atom types, the number of L1 hits increases while that of L1 misses decreases, which indicates L1 cache efficiency improvement. However, surprisingly, we only observe GPU time decreasing in relatively small proteins with less than 500 residues; for proteins over 500 residues, the GPU times with sorted atom sequence are even higher than those without

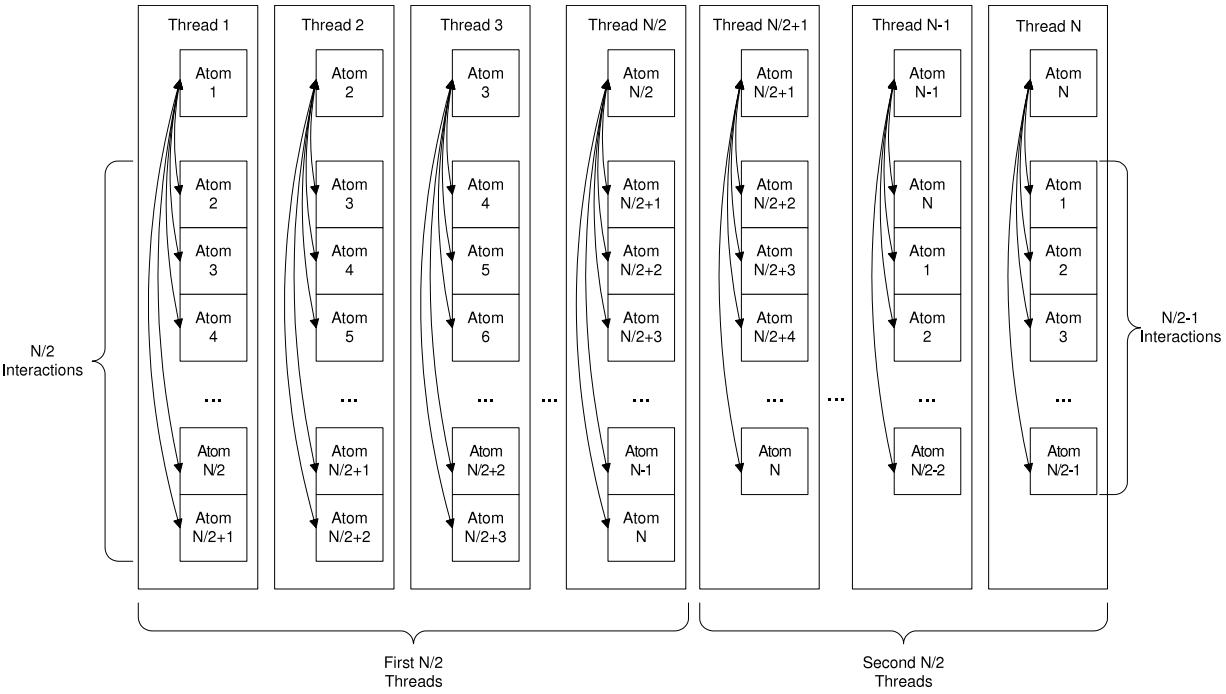


Fig. 6. Nearly perfect balancing when N is an even number. The first $N/2$ threads carry out $N/2$ atom–atom interactions. The second $N/2$ threads carry out $N/2-1$ interactions.

Fig. 7. Reordering atoms in a 6-residue protein fragment according to atom types. Atoms of the same type are clustered after sorting.

sorting. Our further analysis finds that this is mainly caused by the divergent branches in the GPU–DFIRE program. The DFIRE energy evaluation requires testing the atom–atom distance against the DFIRE cutoff—if the atom–atom distance is higher than the DFIRE cutoff, the atom–atom interaction will be ignored, as described in the pseudocode shown in Fig. 1. When the threads handling atom–atom interactions within the DFIRE cutoff as well as those exceeding cutoff co-reside in the same warp, divergent branches may occur in runtime. As shown in Table 2, the number of divergent branches in GPU–DFIRE with sorted atom sequence grows significantly as the protein size increases. In small proteins with less than 500 residues, the numbers of divergent branches in GPU–DFIRE with sorted atom sequences are 1.5–3 times of those without sorting. In contrast, in larger proteins with more than 500 residues, the ratio of divergent branches with sorted and unsorted atom sequences increases to 4–8. This is due to the fact that sorting atom sequences according to atom types increases the chance of divergent branches occurrences, because the clustered together atoms of the same type may be from nearby or faraway residues. In larger protein molecules, the occurrence chance of divergent branch becomes higher after sorting. Consequently, in large proteins, the GPU time gains of the cache efficiency by sorting are counteracted by the increasing number of divergent branches. Nevertheless, in practice, the DFIRE energy function is often used in protein modeling applications on small proteins typically with less than 300 residues. Sorting atom sequences according to atom types are effective in GPU–DFIRE evaluation in small proteins with less than 500 residues due to improved cache efficiency, where the overall GPU times are reduced 11%–45%.

3.4. Other GPU-specific implementations

In addition to asymmetric N-body load balancing and reordering atom sequence according to atom types, we also use the following "standard" CUDA programming techniques in GPU–DFIRE to fully take advantage of the power of the GPU architecture.

(1) Coalesced global memory access

In GPU architecture, global memory access by all threads in the half-warp (non-Fermi) or full-warp (Fermi) of a block can be coalesced into efficient memory transactions [28]. In GPU–DFIRE, we reorganize the data arrays used in the DFIRE program to facilitate coalesced memory access. For example, instead of viewing atoms information as an array of structures, we reconstruct the ATOMS array from "array of structure" to "structure of array". The array reconstruction posts a one-time cost at startup, which has trivial impact to the overall application performance, particularly when the DFIRE potential energy is evaluated many times for different protein conformations. Moreover, threads per block are chosen to be a multiple of warp size.

(2) Fine-grained threads

For an N-body problem with small N, achieving good performance on GPU is difficult, as shown in the literature [29]. In GPU–DFIRE, similar to the technique used in [29], we adopt fine-grained threads by increasing the number of active threads by assigning multiple threads to compute interactions of an atom in a small protein. We firstly calculate the number of threads (T_n) that can be assigned to handle interactions computation of an atom by

Table 2
Performance between sorted and unsorted atom sequences in proteins of various sizes. Performance data are obtained by NVIDIA Compute Visual Profiler 3.2 [7].

PDB	#of res	#of atoms	GPU time			L1 hits		L1 misses		Divergent branches	
			Sorted (μs)	Unsorted (μs)	Sorted/unsorted	Sorted	Unsorted	Sorted	Unsorted	Sorted	Unsorted
1PRB	53	419	49	76	0.65	4,687	3,069	2,305	3,493	160	107
1G6U	96	718	56	85	0.66	8,289	6,007	2,833	4,575	623	291
3DFR	162	1,294	158	242	0.65	17,306	14,581	9,451	12,285	1,962	977
2WJK	210	1,631	220	360	0.61	36,743	24,340	15,980	22,859	1,982	1,018
1YQS	345	2,654	479	729	0.66	111,469	56,012	20,477	27,621	5,226	3,058
3OWH	455	3,619	893	1,011	0.88	139,518	118,291	35,757	55,947	15,247	5,322
2X6Y	597	4,669	1,429	1,416	1.01	220,512	184,191	28,154	48,821	24,133	3,603
2YB9	696	5,595	1,851	1,827	1.01	339,469	301,044	63,379	67,121	24,579	2,147
3NLJ	818	6,642	3,056	2,696	1.13	563,190	422,003	87,131	99,743	27,497	5,432
3GDG	1 068	7,992	3,729	3,534	1.06	710,899	613,904	85,466	248,178	42,726	7,608
3HKY	1 124	8,788	3,985	3,548	1.12	758,835	691,194	79,113	96,515	59,320	8,377
3QHR	1 156	9,284	5,737	4,887	1.17	826,776	729,634	76,225	127,422	67,655	8,110
3NUK	1,314	10,787	7,529	7,131	1.06	1,174,630	1,033,720	66,588	234,798	74,980	15,650
3MBC	1,470	11,222	7,901	7,438	1.06	1,197,540	1,071,820	71,483	184,837	72,870	16,131
2XQY	1,796	13,633	9,504	9,317	1.02	1,863,810	1,650,320	120,925	272,308	79,668	11,929
2WPF	1,955	14,888	12,380	11,799	1.05	2,115,280	1,892,400	108,928	277,929	90,850	14,063
3LU0	3,258	25,407	34,819	33,116	1.05	7,464,650	3,849,770	354,802	381,078	188,759	21,997
2WAQ	3,334	26,454	36,419	34,802	1.05	6,885,780	3,839,500	364,048	666,217	179,784	33,541
3EOO	4,596	34,623	51,778	50,158	1.03	11,196,500	9,431,000	588,343	1,076,800	278,080	38,483
1GYT	6,036	46,152	93,899	91,680	1.02	19,160,200	17,412,200	523,193	1,382,620	364,695	62,972
2PMZ	6,058	48,098	98,554	94,963	1.04	21,747,500	18,818,900	594,047	1,370,870	400,250	50,119
3E3N	6,482	52,706	114,301	111,298	1.03	25,535,300	21,516,200	564,874	1,275,850	367,785	69,461
1UF2	7,434	58,130	151,318	146,181	1.04	31,772,400	27,670,000	743,424	1,647,330	531,273	78,664
3H0G	7,904	62,852	170,996	166,339	1.03	36,193,800	32,331,200	830,885	1,261,420	485,525	74,027
3MQT	9,063	71,547	204,809	200,529	1.02	45,718,400	41,279,800	989,423	2,702,170	642,562	99,490
3KIE	10,400	83,080	283,510	275,717	1.03	63,210,500	57,158,800	1,529,580	2,757,380	744,937	107,584
3HF9	11,907	90,054	342,698	332,235	1.03	72,799,100	70,040,100	1,100,350	2,448,980	636,493	102,063

dividing the maximum number of threads ($T_{\max}$) that a GPU device can launch over the total number of atoms N.

$T_n = T_{\max}/N$.

Then, we distribute the workload of interaction computation to T_n threads. As a result, large number of threads whose total number is near the maximum number of threads that the GPU hardware can launch are created. With sufficient number of threads, the memory access latency can be effectively masked. Fig. 8 shows the effectiveness of fine-grained threads in small proteins. The speedup is defined by dividing the CPU–DFIRE time (t_{CPU}) by the GPU–DFIRE time (t_{GPU}).

$\text{speedup} = t_{\text{CPU}}/t_{\text{GPU}}$.

More specifically, t_{CPU} is approximately the average time ($t_{\text{CPU-per-interaction}}$) for the CPU to handle an interaction multiplied by the total number of pair-wise interactions. t_{GPU} is the sum of the overhead time and the GPU time for pair-wise interactions calculation. The overhead time includes data transferring time between host and device ($O_{\text{data-transfer}}$), which is related to the size of the protein, as well as time managing GPU threads and kernels (O_{other}). The GPU time for pair-wise interactions is determined by $\frac{1}{2}N(N-1)t_{\text{GPU-per-interaction}}T_{\text{parallel}}/T_{\max}$, where $\frac{1}{2}N(N-1)/T_{\max}$ is the number of interactions a thread handles, $t_{\text{GPU-per-interaction}}$ is the time of an interaction calculation in a thread, and T_{parallel} is the number of threads can actually run parallel in GPU. Then the speedup becomes

$$\text{speedup} \approx \frac{\frac{1}{2}N(N-1)t_{\text{CPU-per-interaction}}}{NO_{\text{data-transfer}} + O_{\text{other}} + 12N(N-1)t_{\text{GPU-per-interaction}}T_{\text{parallel}}/T_{\max}}.$$

When N is small, $\frac{1}{2}N(N-1)/T_{\max}$ is small, i.e., each thread does not have enough work, the speedup can be approximated by

$$\text{speedup} \approx \frac{\frac{1}{2}(N-1)t_{\text{CPU-per-interaction}}}{O_{\text{data-transfer}} + O_{\text{other}}/N},$$

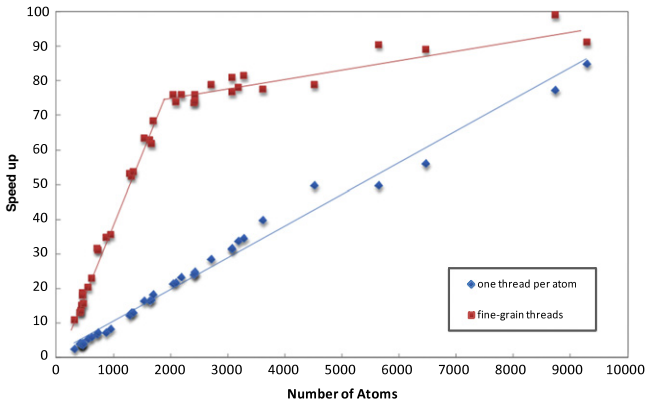


Fig. 8. Effectiveness of fine-grained threads in small proteins (Tesla M2050).

where the speedup almost linearly increases with N. When N is large, each thread has sufficient work and the GPU time for pair-wise interactions calculations dominates the overhead time, the speedup converges to $t_{\text{CPU-per-interaction}}T_{\max}/t_{\text{GPU-per-interaction}}T_{\text{parallel}}$, which is a constant.

Our computational results on Tesla M2050 show that the turning point between linear increasing speedup and constant speedup is when $N = 2000$, as shown in Fig. 8. Fine-grained threads are particularly effective in proteins with less than 2000 atoms. As the number of atoms increases, the speedup curves of fine-grain threads and one-thread-per-atom start to merge because the increasing number of threads in one-thread-per-atom strategy makes more efficient use of the GPU architecture.

(3) Parallel sum reduction

We adopt the parallel sum reduction algorithm [13], a tree-based approach, to compute the overall DFIRE energy from the partial energy sums generated by the GPU threads. The algorithm involves a local sum reduction step and a global sum reduction step. In the local sum reduction step specified in Fig. 4, the tree-

Table 3
Data transferring times in GPU–DFIRE.

Data components	Mode[a]	Time (μs)		
		2ERL (318 atoms)	2X6Y (4669 atoms)	3QHR (9284 atoms)
Residue number array	HtoD	5.66	17.15	36.93
Atom type array	HtoD	4.64	16.16	35.90
DFIRE table	HtoD	840.00	798.59	877.22
Atom coordinate Arrays	HtoD	15.49	59.43	107.70
DFIRE energy	DtoH	1.92	1.95	2.11

[a] HtoD: Host to Device. DtoH: Device to Host.

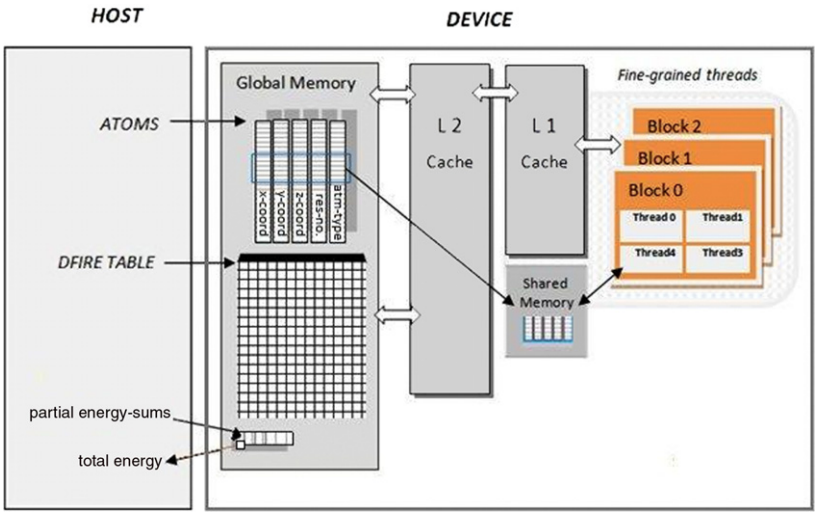


Fig. 9. Implementation of GPU–DFIRE on GPU memory hierarchy (Fermi architecture).

based approach [13] is used to accumulate the partial sums from threads within each block. Then, an additional kernel is launched to add up the partial sums from each block in the previous kernel using the tree-based approach again.

Since fined-grained threads are used for small proteins, the computation time of parallel sum reduction is nearly constant for proteins of various sizes, which is approximately 0.1 ms on Tesla M2050. This is less than the computation time of simply using CPU to sum up the partial sums, which ranges from 0.14 ms (proteins with ∼300 atoms) to 3.0 ms (proteins with ∼10,000 atoms). After all, compare to the overall GPU–DFIRE energy evaluation time, even for the small proteins which typically takes 1.0–3.0 ms, the parallel sum reduction time is relatively small.

(4) Take advantage of GPU memory hierarchy

Our GPU–DFIRE implementation requires transferring the DFIRE table and atom information arrays (ATOMS) including atom type array, residue number array, and atom coordinates arrays, from the host memory to the GPU device memory and retrieving the calculated overall energy from the device memory. Table 3 shows the data transferring time of various components in small (2ERL), medium (2X6Y), and large (3QHR) proteins. One can find that transferring the DFIRE table is most costly; however, it is a one-time cost at GPU–DFIRE startup. Copying residue number array and atom type array can also be one-time transfers if the amino acid sequence of the protein does not change in the protein modeling application. The atoms coordinate arrays need to be updated when the protein 3D conformation changes; however, the transferring time is small compared to the overall GPU–DFIRE energy evaluation time.

We reorganize the data arrays in DFIRE program to different memory locations in the GPU. First of all, we take advantage of the shared memory to reduce the number of accesses to the global memory. In GPU–DFIRE, all threads in a block shares the ATOMS array; hence, the data in the ATOMS array for a thread block can be loaded in the shared memory. Unfortunately, for proteins with large number of atoms, the high-speed shared memory has limited size, which may not be able to accommodate all data in the ATOMS array. We have to break the ATOMS array data into tiles, where a tile represents a fixed dimension of the atom information data. Then, the tile is loaded into the shared memory and used by all threads in a block. Once all threads are done with one tile, the next tile will be loaded into the shared memory and override the previous one. This process is repeated until all computations in the thread block are completed. Moreover, the latest NVIDIA Fermi architecture introduces a cache hierarchy for caching local and global memory accesses. For example, Tesla 2050M GPUs have 64 kB of RAM per streaming multiprocessor, which can be partitioned into shared memory and L1 cache. A CUDA programmer can select combinations of "48 kB shared memory +16 kB L1 cache" or "16 kB shared memory +48 kB L1 cache" according to his/her program needs. Our GPU–DFIRE implementation on Tesla M2050 uses the "48 kB shared memory +16 kB L1 cache" combination, which yields slightly better performance than the other combination. Fig. 9 shows the overall implementation of GPU–DFIRE on GPU memory hierarchy (Fermi).

(5) Loops unrolling

Loop unrolling is an optimization technique that can be applied to GPU programming by replacing the body of a loop with multiple copies of itself [29]. We unroll the atom–atom calculation loop in the pseudocode shown in Fig. 4. Fig. 10 compares the GPU–DFIRE performance of 2 times loop unrolling and 4 times loop unrolling by showing their average percentages relative to the computation time of GPU–DFIRE without loop unrolling. One can find that replicating the loop body for 2 times yields best performance in GPU–DFIRE. Loop unrolling is more effective in large proteins with averagely ∼4% computation time reduction because they contain more loop iterations.

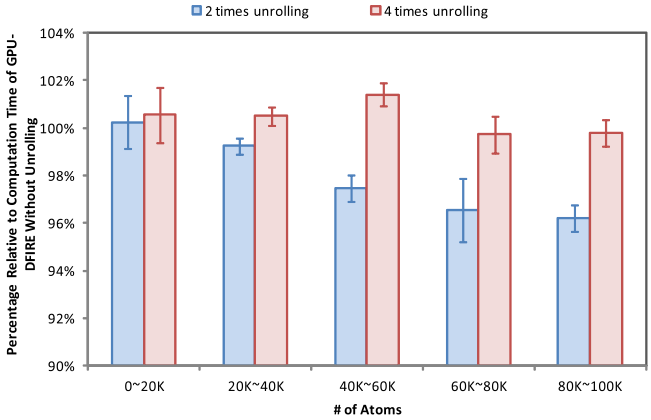


Fig. 10. Effect of loop unrolling in GPU–DFIRE performance (Tesla M2050) on a set of 84 proteins ranges from 318 atoms to 99 595 atoms. The computation time measures are based on the average of 10 runs.

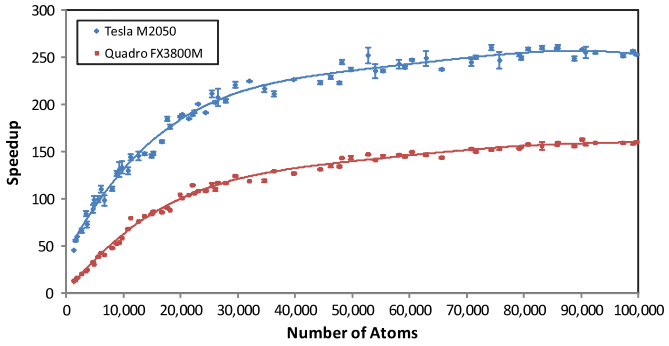


Fig. 11. Overall speedup of GPU–DFIRE on Tesla M2050 and Quadro FX3800M on a set of proteins with various sizes with respect to CPU–DFIRE. The computation time measures are based on the average of 10 runs for each protein.

4. Computational results

The GPU–DFIRE programs are tested on a server with Quadro FX3800M as well as a server with Tesla M2050 GPU. The Tesla M2050 GPU (Fermi architecture) has 14 multiprocessors with 32 cores each, 3 G of global memory, 64 kB of RAM which can be configured between Shared Memory and L1 cache and 32 kB of registers per multiprocessor. The Quadro FX3800M GPU (non-Fermi architecture) has 16 multiprocessors with 8 cores each, 1 G of global memory and 16 K of RAM. The CPU version of DFIRE (CPU–DFIRE) runs on a server with an Intel i7 CPU 920 @ 2.67 GHz, 8 MB cache, and 6 G memory. We firstly benchmark GPU–DFIRE on a set of proteins of various sizes. The GPU time we measured includes the time of transferring the protein information (ATOMS) arrays to GPU device memories, GPU execution time, and the time of retrieving the calculated overall DFIRE energy from GPU. Then, we apply GPU–DIRE to a Monte Carlo program for protein conformation sampling and a program for protein energy local minimization. We use the gcc compiler with the default "-O3" optimization flag specified in DFIRE package for CPU–DFIRE. For GPU–DFIRE, nvcc compiler in CUDA 2.0 is used with "-O3" flag.

4.1. Overall speedup

Fig. 11 shows the overall speedup of GPU–DFIRE using pair-wise interaction on NVIDIA Tesla M2050 and Quadro FX3800M on proteins of various sizes with respect to CPU–DFIRE. Due to not having enough computations in each thread, there are certain inefficiencies in GPU–DFIRE for proteins with less than 20,000 atoms. However, the performance is consistently high for proteins with more than 20,000 atoms. For very large proteins with more than 50,000 atoms, the maximum speedups of GPU–DFIRE can converge to ∼150 and ∼250 using Quadro FX3800M and Tesla M2050, respectively. Moreover, it is important to notice that Tesla M2050 yields significant higher speedups than Quadro FX3800M in small proteins with less than 5000 atoms. This is due to the fact that Quadro FX3800M is non-Fermi architecture without L1 cache, where the strategy of sorting atom sequence by atom types to achieve cache efficiency described in Section 3.3 cannot take effect.

It is also important to note that DFIRE energy is usually used in protein modeling applications with relatively small proteins. In this paper, our goal of using GPU–DFIRE to evaluate DFIRE energy of very large proteins is simply to explore its ultimate speedup on GPU.

4.2. Comparison with cell subdivision method and neighbor lists method

The cell subdivision method and neighbor lists method are popularly used approaches in molecular simulations to avoid unnecessary long distance interaction calculations. When only considering the interactions of neighboring atom pairs, the CPU computation complexity of N-body interactions can be reduced from $O(N^2)$ to $O(N)$ in average case [41,26]. The cell subdivision method divides the overall simulation domain into cells with edges equal to or greater than the interaction cutoff radius so that only the evaluation of interactions between atoms in the same and neighboring cells are needed. The neighbor lists method uses the cell subdivision method to construct a list of neighboring atom pairs within interaction cutoff radius. Anderson et al. [1] implemented the GPU-enabled cell subdivision method and neighbor lists methods for general molecular dynamics.

Fig. 12 compares the performance of GPU–DFIRE implementations using pair-wise interactions, cell subdivision, and neighbor lists methods. One can find that GPU–DFIRE using pair-wise interactions yields better performance for proteins with less than 8000 atoms compared to the other two methods. This is due to the fact that DFIRE has a much larger interaction cutoff radius (15 A) than the typical ones (6–7 A) used in MD. As a result, for proteins with less than 8000 atoms, most atoms reside in the neighboring cells and therefore, the cell subdivision method does not significantly reduce interaction calculations but imposes overhead for computing cell division. On the other hand, GPU–DFIRE using cell subdivision method is effective for large protein with more than 8000 atoms when its reduction in interaction calculation surpasses its overhead. Although the neighbor lists method can further reduce the number of interacting pairs, creating and maintaining neighboring list requires additional access to GPU memory and causes costly overhead, which leads to more overall computation time than the cell subdivision method. The neighbor lists method also outperforms the pair-wise interaction method in very large proteins with more than 13,000 atoms.

4.3. Applications in protein structure modeling

GPU–DFIRE can be adapted to a variety of protein structure modeling algorithms requiring assessing protein molecule energy or feasibility. We use GPU–DFIRE in a Monte Carlo protein conformation sampling program and a local structure optimization program (MINIROT) where DFIRE is the target energy function. The measured computation time is the application execution time, which includes CPU time, GPU time, and the data transferring time between host memory and device memory.

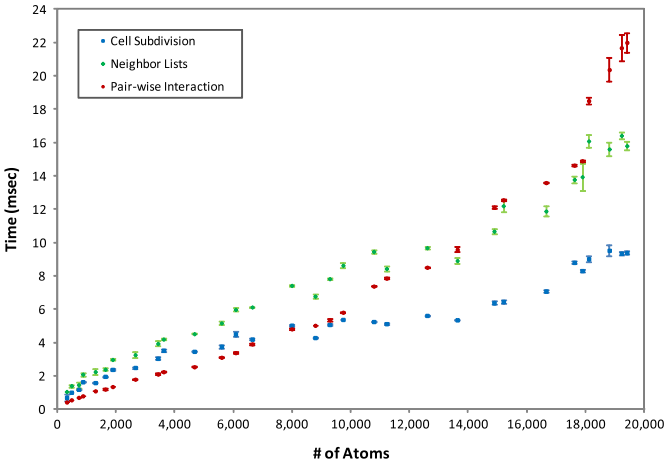


Fig. 12. Performance comparison of GPU–DFIRE implementations using the pair-wise interaction method, the cell subdivision method, and the neighbor lists method.

Table 4
Percentages of energy evaluation times in Monte Carlo sampling program and MINIROT using the CPU-only implementations and the heterogeneous CPU–GPU implementations with energy function evaluation on GPU. The energy evaluation times in heterogeneous CPU–GPU implementations include DFIRE energy calculation time as well as data transferring time between CPU and GPU.

	Percentage of energy evaluation time (CPU-only implementation) (%)	Percentage of energy evaluation time (heterogeneous CPU–GPU implementation with energy evaluation on GPU) (%)
MC	99.9	89.2
MINIROT	75.8	24.1

In this paper, we only consider how the acceleration in energy function evaluation using DFIRE can affect the overall performance of the protein structure modeling program. However, it is important to notice that if more parallel computations, e.g., randomly proposing new atom positions in Monte Carlo algorithm and the matrix operations in MINIROT, are moved to the GPU, more aggressive performance improvements may be obtained. After all, the key advantage of using GPU–DFIRE is that almost no programming modification of the original protein structure modeling program is necessary since GPU–DFIRE can provide the same programming interface as CPU–DFIRE.

4.3.1. Monte Carlo sampling

We use GPU–DFIRE in a Monte Carlo sampling program provided by the TINKER package [38]. DFIRE is used as the target potential energy function. The protein conformational search is carried out by using Cartesian all atoms move where the position of every atom in the protein molecule is changed by a small random perturbation during every Monte Carlo trial. The Monte Carlo sampling program uses the Metropolis algorithm [27] and the DFIRE energy is evaluated in every iteration step to determine the acceptance of the proposed new conformation.

Adopting GPU–DFIRE, the Monte Carlo sampling algorithm is implemented as a heterogeneous CPU–GPU program. The evaluation of the DFIRE energy is carried out on the GPU while the rest of the Monte Carlo computations are executed on the CPU. We carry out the Monte Carlo optimization program using GPU–DFIRE on Tesla M2050 on a protein 3GDG with 7992 atoms. Our computational results show that the number of Monte Carlo steps per second is increased from 2.57 in CPU–DFIRE to 212.67 in GPU–DFIRE. The acceleration in the energy function evaluation significantly improve the performance of the Monte Carlo computation with an average speedup of 82.67, where the original computation time for 10^5 iterations is reduced from more than an hour to less than one minute.

The Cartesian all atom move in this Monte Carlo example requires evaluation of interactions between every atom pair. However, for Monte Carlo methods employing local conformational moves by changing a few torsion angles or positions of small number of atoms in a Monte Carlo trial, energy re-evaluations are only necessary for the atom pairs with relative position changes and thereby the computation times of both CPU–DFIRE and GPU–DFIRE may be further reduced.

4.3.2. Local structure optimization

We adopt GPU-DFIRE in the MINIROT program [21] provided by the TINKER package [38] where DFIRE is the target energy function. The MINIROT program performs local energy minimization of an initial protein structure over dihedral angle space using a limited memory Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm. DFIRE energy is evaluated at every iteration step to determine the descending gradient until convergence is reached. Similar to the implementation of the Monte Carlo program, the evaluation of the DFIRE energy is on the GPU and the rest computations are on CPU. Because evaluation of the descending gradient on each atom is needed in MINIROT, we use the asymmetric scheme shown in Fig. 2 instead of the symmetric scheme in GPU–DFIRE. Table 4 compares the percentages of energy function evaluation times in the Monte Carlo sampling program and the MINIROT program using the CPU-only implementation and the heterogeneous CPU–GPU implementation with energy calculations on GPU. Compared to the Monte Carlo program where 99.9% of its computation time is in energy evaluation, the MINIROT program has significant more computations on CPU due to its matrix operations in linear search. Therefore, the energy evaluation occupies 75.8% of the overall computation time in the MINIROT program, which is still in majority but is much less than that of the Monte Carlo sampling program. Table 5 shows the performance of the MINIROT program on 6 initial structures of protein 2ERL using GPU–DFIRE and CPU–DFIRE. Using GPU–DFIRE by migrating the energy evaluation computation to the GPU, the percentage of computational time spent in energy evaluation in the overall MINIROT program is reduced from 75.8% to 24.1%. As a result, although not as significant as that of the Monte Carlo program using GPU–DFIRE, MINIROT with GPU–DFIRE can achieve an average speedup of ∼4.5.

Table 5
Performance comparison of MINIROT program on 6 initial structures of 2ERL (319 atoms) using GPU–DFIRE and CPU–DFIRE. GPU–DFIRE is carried out on Tesla M2050 server.

	RMSD (A)	Num. of steps	Execution time (s)	
			CPU–DFIRE	GPU–DFIRE (Tesla M2050)
Initial 1	3.19	338	64.76	14.19
Initial 2	2.06	225	43.39	9.73
Initial 3	2.24	480	91.70	19.89
Initial 4	3.19	222	43.23	10.02
Initial 5	2.85	692	130.35	26.81
Initial 6	3.62	434	82.21	17.27

5. Summary and future research directions

In this paper, we investigate the approaches of using GPU to accelerate calculation of memory-intensive, knowledge-based energy functions for protein structure modeling applications. We develop GPU–DFIRE—a GPU implementation for the all-atom, knowledge-based DFIRE potential energy function. A workload distribution scheme is designed to achieve perfect or near-perfect load balancing in the symmetric N-body problem in DFIRE. Reordering atoms in the protein according atom types also improves the GPU cache efficiency, which is effective for small proteins but is harmful for large ones due to increasing divergent branches as protein size grows. Other standard GPU programming techniques such as coalesced global memory access, parallel sum reduction, fine-grained threads, and loop unrolling, are adopted to take advantage of the GPU architecture to its fullest. Speedups of up to 150 and 250 are observed in GPU–DFIRE on Tesla M2050 and Quadro FX3800M, respectively, relative to the DFIRE computation on the CPU. Because evaluating the energy function is the most costly operation in many protein structure modeling applications, significant performance improvements are also found in a Monte Carlo program for protein conformation sampling and a protein structure local optimization program when GPU–DFIRE is used to replace CPU–DFIRE. Compared to cell subdivision method or neighbor lists method, the GPU–DFIRE implementation using pair-wise interactions is more effective for proteins with less than 8000 atoms because of the large radius cutoff in DFIRE potential.

The techniques developed in GPU–DFIRE can also be applied to other knowledge-based energy functions. Our future research directions will include investigating the development of a general GPU-accelerated framework that can be easily employed to accelerate other knowledge-based energy functions. We are also interested in extending our work to physics-based energy functions to determine the best combination of tabulated and calculated terms that can yield optimal performance in GPU architectures. Moreover, we plan to integrate GPU–DFIRE into other protein modeling algorithms such as molecular dynamics and genetic algorithms.

Acknowledgments

We would like to thank Dr. Yaoqi Zhou for his valuable comments on the manuscript and providing the DFIRE package. We also want to thank the reviewers for their suggestions that greatly helped us to improve the contents of the manuscript. Yaohang Li acknowledges support from NSF grant 1066471 and ODU 2011 SEECR grant.

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 (2008) 5342–5359.
[2] J. Barnes, P. Hut, A hierarchical $O(n \log n)$ force calculation algorithm, Nature 324 (1986) 446–449.
[3] 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 (2008) 103–112.
[4] R. Bonneau, J. Tsai, I. Ruczinski, D. Chivian, C. Rohl, C.E. Strauss, D. Baker, Rosetta in CASP4: progress in *ab initio* protein structure prediction, Proteins 5 (2001) 119–126.
[5] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, M. Karplus, CHARMM: a program for macromolecular energy, minimization and dynamics calculations, J. Comput. Chem. 4 (1983) 187–217.
[6] S. Browne, C. Deane, G. Ho, P. Mucci, PAPI: a portable interface to hardware performance counters, in: Proceedings of Department of Defense HPCMP Users Group Conference, 1999.
[7] Compute visual profiler 3.2, 2011. developer.download.nvidia.com.
[8] W.D. Cornell, P. Cieplak, C.I. Bayly, I.R. Gould, K.M. Merz, D.M. Ferguson, D.C. Spellmeyer, T. Fox, J.W. Caldwell, P.A. Kollman, A second generation force-field for the simulation of proteins, nucleic-acids, and organic-molecules, J. Am. Chem. Soc. 117 (1995) 5179–5197.
[9] T. Darden, D. York, L. Pederson, Particle mesh Ewald: an $N \log(N)$ method for Ewald sums in large systems, J. Chem. Phys. 98 (12) (1993) 10089–10092.
[10] E. Faraggi, Y. Yang, S. Zhang, Y. Zhou, Predicting continuous local structure and the effect of its substitution for secondary structure in fragment-free protein structure prediction, Structure 17 (2009) 1515–1527.
[11] D.B. Gordon, S.A. Marshall, S.L. Mayo, Energy functions for protein design, Curr. Opin. Struct. Biol. 9 (1999) 509–513.
[12] L. Greengard, The Rapid Evaluation of Potential Fields in Particle Systems, ACM Press, 1987.
[13] M. Harris, Optimizing parallel reduction in CUDA, NVIDIA Developer Technology, 2008.
[14] R. Hockney, J. Eastwood, Computer Simulation Using Particles, McGraw-Hill, 1981.
[15] P. Jetley, L. Wesolowski, F. Gioachin, L.V. Kale, T.R. Quinn, Scaling hierarchical N-body simulations on GPU clusters, in: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, SC10, 2010.
[16] W.L. Jorgensen, D.S. Maxwell, J. Tirado-Rives, Development and testing of the OPLS all-atom force field on conformational energetics and properties of organic liquids, J. Am. Chem. Soc. 118 (1996) 11225–11236.
[17] G.G. Krivov, M.V. Shapovalov, R.L. Dunbrack, Improved prediction of protein side-chain conformations with SCWRL4, Proteins 77 (4) (2009) 778–795.
[18] R. Kufrin, PerfSuite: an accessible, open source performance analysis environment for Linux, in: 6th International Conference on Linux Clusters: The HPC Revolution, 2005.
[19] M.S. Lee, M.A. Olson, Assessment of detection and refinement strategies for de novo protein structures using force field and statistical potentials, J. Chem. Theory Comput. 3 (2007) 312–324.
[20] Y. Li, I. Rata, S. Chiu, E. Jakobsson, Improving predicted protein loop structure ranking using a Pareto-optimality consensus method, BMC Struct. Biol. 10 (2010) 22.
[21] Z. Li, H.A. Scheraga, Monte Carlo-minimization approach to the multiple-minima problem in protein folding, Proc. Natl. Acad. Sci. 84 (1987) 6611–6615.
[22] Y. Li, D. Wardell, V. Freeh, A resource-efficient computing paradigm for computational protein modeling applications, in: Proceedings of the 8th IEEE International Workshop on High Performance Computational Biology, HiCOMB09, 2009.
[23] M. Lu, A.D. Dousis, J. Ma, OPUS–PSP: an orientation-dependent statistical all-atom potential derived from side-chain packing, J. Mol. Biol. 376 (1) (2008) 288–301.
[24] A.D. MacKerell, D. Bashford, M. Bellott, R.L. Dunbrack, J.D. Evanseck, M.J. Field, S. Fischer, J. Gao, H. Guo, S. Ha, D. Joseph-McCarthy, L. Kuchnir, K. Kuczera, F.T.K. Lau, C. Mattos, S. Michnick, T. Ngo, D.T. Nguyen, B. Prodhom, W.E. Reiher, B. Roux, M. Schlenkrich, J.C. Smith, R. Stote, J. Straub, M. Watanabe, J. Wiorkiewicz-Kuczera, D. Yin, D.M. Karplus, All-atom empirical potential for molecular modeling and dynamics studies of proteins, J. Phys. Chem. B 102 (1998) 3586–3616.
[25] M.A. Marti-Renom, A.C. Stuart, A. Fiser, R. Sanchez, F. Melo, A. Sali, Comparative protein structure modeling of genes and genomes, Annu. Rev. Biophys. Biomol. Struct. 29 (2000) 291–325.
[26] T. Maximova, C. Keasar, A novel algorithm for non-bonded-list updating in molecular simulations, J. Comput. Biol. 13 (2006) 1041–1048.
[27] 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 (1953) 1087–1092.

[28] NVIDIA, 2010. CUDA programming guide version 3.1. Available at: http://www.nvidia.com/object/cuda_get.html.
[29] L. Nyland, M. Harris, J. Prins, Fast *N*-body simulation with CUDA, GPU Gems (2007) 3.
[30] C. Oostenbrink, A. Villa, A.E. Mark, W.F. Van Gunsteren, A biomolecular force field based on the free enthalpy of hydration and solvation: the GROMOS force-field parameter sets 53A5 and 53A6, J. Comput. Chem. 25 (2004) 1656–1676.
[31] D. Patterson, The top 10 innovations in the new NVIDIA fermi architecture, and the top 3 next challenges, 2009. www.NVIDIA.com.
[32] J. Peng, J. Xu, RaptorX: exploiting structure information for protein alignment by statistical inference, Proteins (2011) doi:10.1002/prot.23175.
[33] A. Piccolboni, G. Mauri, Application of evolutionary algorithms to protein folding, prediction, in: Proceedings of ICONIP97, 1998.
[34] J.W. Ponder, F.M. Richards, An efficient Newton-like method for molecular mechanics energy minimization of large molecules, J. Comput. Chem. 8 (1987) 1016–1026.
[35] A. Roy, A. Kucukural, Y. Zhang, I-TASSER: a unified platform for automated protein structure and function prediction, Nat. Protoc. 5 (2010) 725–738.
[36] M.Y. Shen, A. Sali, Statistical potential for assessment and prediction of protein structures, Protein Sci. 15 (2006) 2507–2524.
[37] M.J. Stock, A. Gharakhani, Toward efficient GPU-accelerated *N*-body simulations, in: Proceedings of 46th AIAA Aerospace Sciences Meeting and Exhibit, 2008.
[38] Tinker package, 2011. http://dasher.wustl.edu/tinker/.
[39] D. Xu, J. Zhang, A. Roy, Y. Zhang, Automated protein structure modeling in CASP9 by I-TASSER pipeline combined with QUARK-based *ab initio* folding and FG–MD-based structure refinement, Proteins (2011) doi:10.1002/prot.23111.
[40] Y. Yang, Y. Zhou, *Ab initio* folding of terminal segments with secondary structures reveals the fine difference between two closely-related all-atom statistical energy functions, Protein Sci. 17 (2008) 1212–1219.
[41] Z. Yao, J. Wang, G. Liu, M. Cheng, Improved neighbor list algorithm in molecular simulations using cell decomposition and data sorting method, Comput. Phys. Comm. 161 (2004) 27–35.
[42] Y. Zhang, Template-based modeling and free modeling by I-TASSER in CASP7, Proteins (Suppl. 8) (2007) 108–117.
[43] C. Zhang, S. Liu, Y. Zhou, Accurate and efficient loop selections by the DFIRE-based all-atom statistical potential, Protein Sci. 13 (2) (2004) 391–399.
[44] H. Zhou, J. Skolnick, Protein structure prediction by pro-Sp3-TASSER, Biophys. J. 96 (6) (2009) 2110–2127.
[45] H. Zhou, Y. Zhou, Distance-scaled, finite ideal-gas reference state improves structure-derived potentials of mean force for structure selection and stability prediction, Protein Sci. 11 (2002) 2714–2726.

Ashraf Yaseen received the B.S. degree in CS from Jordan University of Science and Technology in 2002 and the M.S. degree in CS from New York Institute of technology in 2003. He is currently pursuing his Ph.D. degree in Computer Science at Old Dominion University. His research interests include Computational Biology and High Performance Computing.

Yaohang Li is an Associate Professor in Computer Science at Old Dominion University. He received his B.S. in Computer Science from South China University of Technology in 1997 and M.S. and Ph.D. degrees from the Department of Computer Science, Florida State University in 2000 and 2003, respectively. After graduation, he worked as a research associate in the Computer Science and Mathematics Division at Oak Ridge National Laboratory, TN. His research interest is in Computational Biology, Monte Carlo Methods, and High Performance Computing.