DOI 10.5012/bkcs.2010.31.12.3639

Accelerating Molecular Dynamics Simulation Using Graphics Processing Unit

Hun Joo Myung, Ryuji Sakamaki,[†] Kwang Jin Oh,[*] Tetsu Narumi,[‡] Kenji Yasuoka,[†,*] and Sik Lee

KISTI Supercomputing Center, Daejeon 305-806, Korea. [*]E-mail: koh@kisti.re.kr
[†]Department of Mechanical Engineering, Keio University, Yokohama 223-8522, Japan. [*]E-mail: yasuoka@mech.keio.ac.jp
[‡]Department of Computer Science, University of Electro-Communications, Tokyo, Japan
Received July 16, 2010, Accepted October 6, 2010

We have developed CUDA-enabled version of a general purpose molecular dynamics simulation code for GPU. Implementation details including parallelization scheme and performance optimization are described. Here we have focused on the non-bonded force calculation because it is most time consuming part in molecular dynamics simulation. Timing results using CUDA-enabled and CPU versions were obtained and compared for a biomolecular system containing 23558 atoms. CUDA-enabled versions were found to be faster than CPU version. This suggests that GPU could be a useful hardware for molecular dynamics simulation.

Key Words: Molecular dynamics simulation, Parallel computing, GPU, Biomolecules

Introduction

Molecular dynamics (MD) simulation[1,2] integrates numerically Newton's equations of motion of a system and produces a numerical solution for a given potential. MD simulation has been a useful tool to study molecular systems in various fields like chemistry, physics, chemical engineering, and materials science. Typical applications range from simple molecular systems such as water to more complex systems such as proteins, membranes, and polymers. The phases of interest have ranged from simple phases such as fluids and solids to interfaces and mixtures.

Depending on length and time scales of phenomena and properties of interest, the computational requirement shows a wide spectrum. For example, it is now a routine task to simulate 10^4 - 10^5 particle systems for nanoseconds within one day. However, it is still a difficult task to simulate large macromolecular systems of micrometer dimensions over microsecond timescales using the MD simulation within reasonable wall clock time even with parallel computers.

In order for MD simulation to be more useful, large length and long time scale MD simulation should be feasible by reducing the time-to-solution. There could be two approaches to reduce computational time. One is to develop an efficient algorithm or a coarse-grained model.[3] For example, one could save computational time by using neighbor list algorithm[1,2] for the non-bonded force calculation and smooth particle mesh Ewald method[4] for electrostatic force calculation. Or one could reduce computational load by using a reduced number of particles for interaction with a coarse-grained model. Another example is to use RESPA[5] (reference system propagator algorithm) to integrate the equation of motion with a larger integration time step. A recent study reported that the time step can be increased even up to ~100 fs.[6] The other is to develop an efficient parallel algorithm[7,8,9] for high performance computing. The basic idea of this approach is to reduce computational time by distributing computational load over many processors. Therefore a supercomputer with more processors has been desired for this approach. According to Moore's law,[10] computational power has doubled every 18 months. Now a petascale computer begins to appear in TOP500 list.[11] It usually contains hundreds of thousands of processors. Definitely it would be a good opportunity for us to be able to utilize such a powerful supercomputer.

However, there has been new trend in high performance computing toward hybrid computing due to the fact that power and heat dissipation constraints have prevented microprocessor clock rates from increasing substantially in the last several years. Recently hybrid computing based on graphics processing unit (GPU) has gained much interest as an alternative to homogeneous architecture technologies based only on CPU. This stems from the fact that GPU shows better performance and power consumption than CPU. TSUBAME[11] installed in Tokyo Institute of Technology used NVIDIA GPUs to boost its computational power up to 170 TFLOPS (10^{12} floating point operation per second) and was ranked as 29 th in TOP500 list on September 2008. Another hybrid system, Tianhe-1[11] installed at the National Super Computer Center in China, ranked as 5th position in TOP 500 list announced on September 2009. The theoretical peak of Tianhe-1 was 1.2 PFLOPS (10^{15} floating point operation per second). It is a hybrid supercomputer which consists of Intel Xeon processors and AMD GPUs.

Even though GPU was originally developed for computer graphics, the MD simulation community also begins to recognize the usefulness of GPU for MD simulation. Recently several studies have reported GPU acceleration of MD simulation.[12-19] However, most studies considered only relatively simple systems like a system interacting only through Lennard-Jones interaction, which are lack of electrostatic interactions. The purpose of this paper is to implement a general purpose MD simulation code[20] for GPU using NVIDIA's CUDA[21] (Compute Unified Device Architecture).

Implementation

GPU is viewed as a compute device which is implemented as a set of multiprocessors. Each multiprocessor has SIMT (sin-

gle-instruction-multiple-thread) architecture and consists of scalar processor cores. SIMT executes one instruction across many threads. Besides the multiprocessors, there are four types of on-chip memory in GPU. Those are 32-bit register, shared memory, a read-only constant cache, and a read-only texture cache. The register is visible only to the processor, but all others are shared by all the processors.

CUDA decomposes a problem into a grid of thread blocks. The thread block is a group of threads which are executed concurrently on one multiprocessor. When a CUDA program on CPU invokes a kernel (a function in C-like language) grid, the thread blocks of the grid are enumerated and distributed to available multiprocessors. As the thread block terminates, new blocks are launched on the vacated multiprocessors. The multiprocessor SMIT unit creates, manages, schedules, executes threads in groups of 32 parallel threads called warp. Each thread is executed independently on one scalar processor core with its own instruction address and register state.

The threads in a thread block share their data using the shared memory. Threads in different thread blocks cannot communicate or synchronize with each other. On the other hand, the global, constant, and texture memory are available to all threads.

In MD simulation, bottlenecks are the non-bonded force calculation. Therefore we have concentrated on two subroutines responsible for those calculations. We used a general purpose MD simulation code mm_par[18] for GPU implementation using CUDA. The subroutines are v_real() and v_pme(). The former calculates force (electrostatic force + van der Waals force) in real space and the latter calculates the electrostatic force in reciprocal space using smooth particle mesh Ewald method.[4] The force calculation in real space is proportional to N^2 and the force calculation in the reciprocal space is proportional to NlogN where N is the number of particles in the system.

The non-bonded energies in v_real() and v_pme() are respectively given by

$$V_{real} = \sum_i \sum_{j>i} 4\varepsilon\left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right] + \sum_i \sum_{j>i} \frac{1}{4\pi\varepsilon_0}\frac{q_i q_j erfc(\alpha r_{ij})}{r_{ij}} \tag{1}$$

$$V_{reciprocal} = \frac{1}{\varepsilon_0}\sum_{\vec{k}\neq 0}\frac{\exp(-k^2/4\alpha^2)}{k^2}\left|S(\vec{k})\right|^2 . \tag{2}$$

Here ε and σ are Lennard-Jones parameters and α is Ewald parameter. The structure factor $S(\vec{k}) = \sum q_i \exp(i\vec{k}\bullet\vec{r}_i)$ in the smooth particle mesh Ewald method is approximated by

$$S(\vec{k}) \approx b_1(k_1)b_2(k_2)b_3(k_3)F(Q)(k_1,k_2,k_3) \tag{3}$$

where

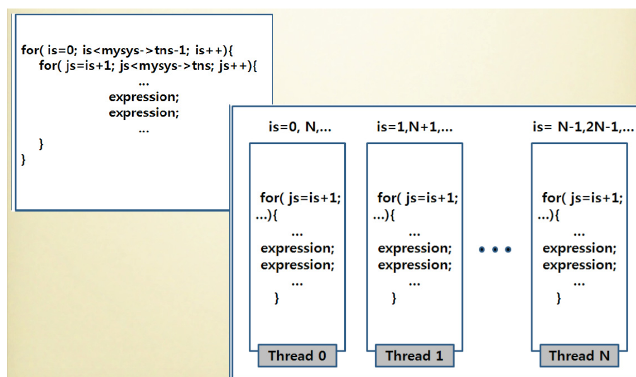

Figure 1. Parallelization scheme for v_real() and v_pme().

$$b_i(k_i) = \exp(2\pi i(n-1)k_i/K_i)\left[\sum_{j=0}^{n-2} M_n(j+1)\exp(2\pi ik_i j/K_i)\right]^{-1} \times\left[\sum_{j=0}^{n-2} M_n(j+1)\exp(2\pi ik_i j/K_i)\right]^{-1}, \tag{4}$$

and

$$Q(k_1,k_2,k_3) = \sum_i \sum_{n_1,n_2,n_3} q_i M_n(u_{1i}-k_1-n_1K_1) M_n(u_{2i}-k_2-n_2K_2)M_n(u_{3i}-k_3-n_3K_3). \tag{5}$$

Here the scaled fractional coordinates u_i range from 0 to K_i for $i = 1, 2$, and 3. $M_n(u_i)$ is the Cardinal B-spline of order n and $F(Q)$ is the Fourier transform of Q.

As can be expected from eq. (1), the basic control flow of v_real() is to span all particle pairs (which requires double loop over particles) and search for interacting pairs. Once an interacting particle pair is identified, it calculates force and energy. In parallelizing eq. (1) for GPU, we have used a scheme similar to replicated data strategy[22] (see figure 1). That is to say, we assigned force calculation on particle i with other particles to thread i. Similar idea has been applied to v_pme(). Therefore each thread owns N/P atoms, where P is total number of threads.

For performance optimization, we considered a few things in implementing those algorithms for GPU. First is memory access pattern. As is well known, GPU has no cache memory like the one in CPU. Since the access to the global memory is slow, there exist non-negligible overheads for the threads to load data from the global memory. In order to reduce these overheads, it would be better to use array than linked list because each thread can access contiguous global memory address with single memory transaction. Instead of the cache memory, GPU has the shared, constant, and texture memory which are as fast as the register. Therefore we can make best use of those faster memories. In our implementation, we stored positions and charges of particles on the shared memory to reduce global memory reference count.

Constant memory is read-only and small. Therefore we stored frequently-referenced constant numbers such as total number of particles on this memory. Finally, texture memory is also read-only and is optimized for 2D spatial locality. Therefore it is good for storing potential table on this memory.

Second is memory size. Each thread writes data to its local memory and then we assemble all of data from each thread with the reduction method. But, we need a large amount of memory to save number of threads × number of particles × size of float. Global memory might not be enough to store all those. Therefore we performed the reduction in a different way. First we allocated global memory based on thread block instead of thread and stored intermediate results from the calculation on the shared memory. And then we stored final results on the global memory. Finally, the reduction was done on the CPU side.

Third is race condition. The race condition generally occurs when one or more threads access the same shared memory location and this multiple access has not been properly controlled. To avoid the race condition in the kernel, we arranged data arrays on the skew in a thread block as shown in figure 5.

Finally, since a warp executes one common instruction at a time, full efficiency is realized when all 32 threads of a warp execute the same instruction. Conditional branch makes the threads of a warp to diverge so that the threads follow different execution paths. In this case, the warp executes serially each branch path taken until all branch paths are complete and the threads converge back to the same execution path.

Results and Discussion

For performance test, we used NVIDIA Geforce GTX 285 with 1GB global memory and Intel(R) Core(TM)2 Quad CPU Q9550 of 2.83GHz. Our test system contains a protein (PDB id: 5DHFR) with CHARMM22 force field[23] and TIP3P[24] waters (see figure 2). The system contains a total of 23558 atoms in simulation box having dimension 62.23 Å × 62.23 Å × 62.23 Å. The potential cutoff distance was set to 9 Å for the non-bonded force calculation in real space. For the SPME calculation, K_1, K_2, and K_3 were set to 64 and the interpolation order was set to 4. The Nosé-Hoover chain[5] is coupled to the system globally. Each Nosé-Hoover chain has 5 thermostats. We used 3 and 5 for n_c and n_{ys}, respectively. All bonds including hydrogen atoms were constrained using SHAKE/RATTLE algorithms[25,26] with a tolerance of 10^{-8}.

Figure 3 shows atom index i vs. error ε_i which is given by

$$\varepsilon_i = \frac{\left|\vec{f}_{CPU,i} - \vec{f}_{GPU,i}\right|}{\sum_k \left|\vec{f}_{CPU,k}\right|} \tag{6}$$

where $\vec{f}_{CPU,i}$ is the force on atom i calculated from double precision CPU version and $\vec{f}_{GPU,i}$ is the force on atom i calculated using CUDA-enabled version. As shown in the figure, the order of the error ε_i is 10^{-5} for single precision calculation and even 10^{-13} for double precision calculation. Here we used single precision for the values saved in the texture memory since GTX 285 supports only single precision type for the texture memory. As

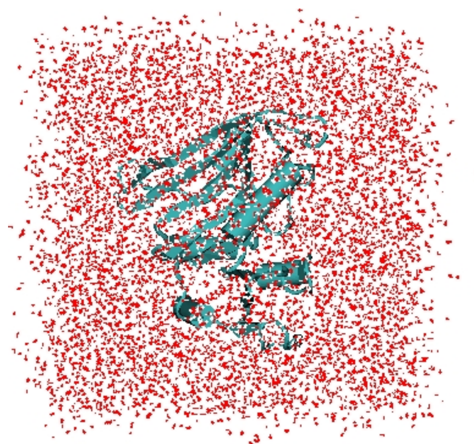

Figure 2. A protein (PDB id: 5DHFR) plus 7023 TIP3P system used for performance test. The system contains total 23558 atoms and the simulation box dimension is 62.23 Å × 62.23 Å × 62.23 Å.

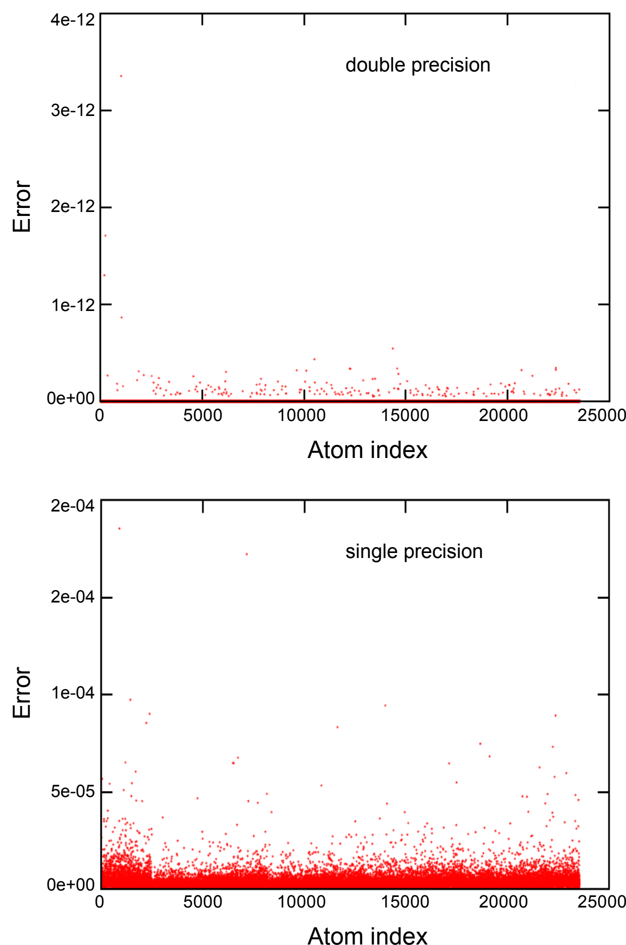

Figure 3. Error *vs.* atom index plot from (a) single precision simulation and (b) double precision simulation on GPU. The error was calculated by using Eq. (6).

is well known, even summing identical values in a different order produces different results. The force calculation involves tons of arithmetic operations. Therefore the accumulation of many forces from surrounding atoms can lead to different error for

Table 1. Timing results in seconds per MD step of v_real() and v_pme. Total timing results are also given in the last column. dCPU, sGPU, dGPU represent double precision CPU version, single precision GPU version, double precision GPU version, respectively

Run #	v_real()			v_pme()			Total		
	dCPU	sGPU	dGPU	dCPU	sGPU	dGPU	dCPU	sGPU	dGPU
1	5.27	0.17	0.73	0.07	0.020	0.023	5.43	0.24	0.80
2	5.47	0.16	0.74	0.07	0.020	0.023	5.59	0.20	0.78
3	5.48	0.17	0.73	0.07	0.020	0.023	5.6	0.20	0.78
4	5.47	0.16	0.73	0.07	0.020	0.024	5.59	0.20	0.77
5	5.48	0.16	0.73	0.07	0.020	0.024	5.6	0.19	0.77
Average	5.43	0.16	0.73	0.07	0.020	0.023	5.56	0.21	0.78

different atom even in double precision simulation on GPU. The sums of the error ε_i from this study are 0.097 and 3.3×10^{-11} for single and double precision simulations, respectively. A previous study[27] suggested that the error sum of forces of 10^{-3} or larger is not sufficient for a stable MD simulation. This means that total Hamiltonian in our single precision simulation might not be conserved. In our case, we observed a stable MD simulation during 1 million step run. Here we used a smaller time step (0.001 fs) to reduce the numerical error. Note that the stability of MD simulation depends on the numerical error as well as the round-off error.

Table 1 shows timing results in seconds per MD step from five runs. As given in table 1, v_real() in single precision simulation is accelerated by about 34 times while v_pme() is accelerated only by about 4 times. Since one has more dramatic improvement in performance for more time-consuming part from parallelization, one can expect larger speedup in v_real() than v_pme() from the GPU implementation.

Overall timing results from MD simulations for 100 steps are shown in figure 4. According to NVIDIA, the single precision performance of GeForce GTX285 is 1062 GFLOPS. On the other hand, the double precision performance is only 85 GFLOPS. The theoretical peak of single precision performance is about 10 times higher than that of double precision performance. For our case, the sustained performance of double precision version is about 4 times slower than that of single precision as shown in figure 8. This is due to the fact that there is usually gap between the theoretical peak performance and the sustained performance originating mainly from hardware architecture and non-floating point operations in the code, and sometimes from poorly optimized code. In any case, using double precision reduces the performance.

Single and double precision CUDA-enabled versions on GPU are 26 and 7 times faster than CPU version, respectively. In the CUDA-enabled version, we implemented simple $O(N^2)$ algorithm for the non-bonded force calculation in real space. It just spans all pairs and searches for interacting pairs, that is to say, pairs within the potential cutoff distance, and then calculates and accumulates forces for the interacting pairs. The main advantages of this algorithm for GPU are that it is easier to implement, minimizes global memory access, avoids thread synchronization, and takes advantage of symmetry. On the other hand, conventional bookkeeping techniques[1,2] such as Verlet neighbor list method, cell-linked list method, and combined method of Verlet neighbor list and cell-linked list might not be

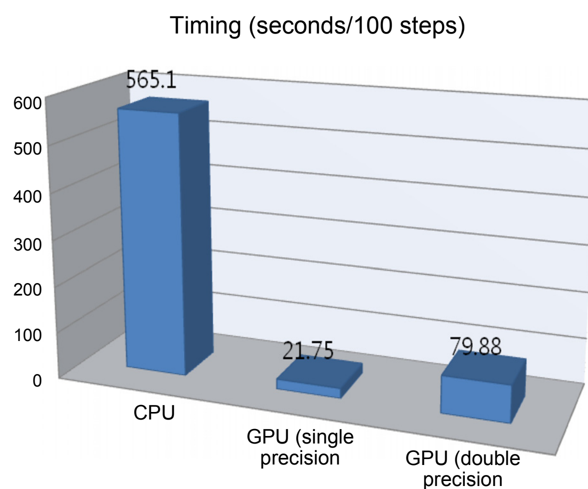

Figure 4. Performance results for 100 MD steps from double precision simulation on CPU, single precision simulation on GPU, and double precision simulation on GPU.

efficient for GPU due to the fact that those involve out of order memory access and inner loop contains non-coalesced memory access even though those are known to be faster algorithms for calculating the non-bonded force calculation on CPU. But note that the particle number dependence of the computational time of $O(N^2)$ algorithm would outpace the performance gain by GPU for larger systems.[28] In this case, an efficient implementation of a book-keeping algorithm on GPU or other kind of efficient parallel algorithm might be required.

Conclusion

We have developed CUDA-enabled version of a general purpose molecular dynamics simulation code for GPU. Since the non-bonded force calculation is most time-consuming part in MD simulation, we have focused on parallelizing the subroutines responsible for the non-bonded force calculation. Timing results using CUDA-enabled and CPU versions were obtained and compared for a medium-sized biomolecular system containing 23558 atoms. This test system involves both van der Waals interaction and electrostatic interaction. From the timing results using CPU and CUDA-enabled versions, it is found that single precision CUDA-enabled version is about 26 times faster than its double precision CPU version. However our double precision

GPU implementations were only times faster than Verlet neighbor list algorithm on CPU. In any case, the CUDA-enabled version enables us to simulate about 1.5 ns per day. Even though the code is not fully parallelized on GPU, this study demonstrates that GPU could be a useful hardware accelerator for MD simulation.

However, there are a few issues for further studies to improve parallel performance. First, for larger system, $O(N^2)$ algorithm for the non-bonded force calculation in real space is not sufficient for achieving high performance on GPU due to its computational dependency on the particle number. Therefore an efficient parallel implementation for a book-keeping technique or any other good parallel algorithm for GPU might be required. Second, single precision might be lack of enough accuracy for longer time scale simulations. Even though a systematic study has not been reported yet for the effects of the precision on MD simulation, a mixed precision method[28] could be a good candidate to start for achieving both high performance and high accuracy. Third, the present version is implemented on single CPU/single GPU system. For larger scales and longer time simulations, for example, protein folding simulations of large proteins in solution, a code for multi-CPU/multi-GPU system would be required. In this case, a novel parallelization scheme should be developed to make best use of multi-CPU/multi GPU architecture and the overhead from data transfer between CPU and GPU should be minimized to increase the scalability.

References

1. Allen, M. P.; Tildesley, D. J. *Computer Simulation of Liquids*; Clarendon: Oxford, 1987.
2. Smit, B.; Frenkel, D. *Understanding Molecular Simulation*; Academic Press: Orlando, 2001.
3. Shelley, J. C.; Shelley, M. Y.; Reeder, R. C.; Bandyopadhyay, S.; Klein, M. L. *J. Phys. Chem. B* **2001**, *105*, 4464.
4. Essmann, U.; Perera, L.; Berkowitz, M. L.; Darden, T.; Lee, H.; Pedersen, L. G. *J. Chem. Phys.* **1995**, *103*, 8577.
5. Martyna, G. J.; Tuckerman, M. E.; Tobias, D. J.; Klein, M. L. *Mol. Phys.* **1996**, *87*, 1117.
6. Minary, P.; Tuckerman, M. E.; Martyna, G. J. *Phys. Rev. Lett.* **2004**, *93*, 150201.
7. Plimpton, S. *J. Comp. Phys.* **1995**, *117*, 1.
8. Oh, K. J.; Deng, Y. *Comp. Phys. Comm.* **2007**, *177*, 426.
9. Oh, K. J.; Klein, M. L. *Comp. Phys. Comm.* **2006**, *174*, 263.
10. Moore, G. E. *Electronics Magazine* **1965**, *38*, 4.
11. http://www.top500.org.
12. Anderson, J. A.; Lorenz, C. D.; Travesset, A. *J. Comp. Phys.* **2008**, *227*, 5342.
13. Stone, J. E.; Phillips, J. C.; Freddolino, P. L.; Hardy, D. J.; Trabuco, L. G.; Schulten, K. *J. Comp. Chem.* **2007**, *28*, 2618.
14. Yang, J.; Wang, Y.; Chen, Y. *J. Comp. Phys.* **2007**, *221*, 799.
15. van Meel, J. A.; Arnold, A.; Frenkel, D.; Portegies Zwart, S. F.; Belleman, R. G. *Mol. Sim.* **2008**, *34*, 259.
16. Hardy, D. J.; Stone, J. E.; Schulten, K. *Parallel Computing* **2009**, *35*, 164.
17. Liu, W.; Schmidt, B.; Voss, G.; Muller-Wittig, W. *Comp. Phys. Comm.* **2008**, *179*, 634.
18. Taufer, M.; Saponaro, P.; Padron, O.; Patel, S. *Improving Numerical Reproducibility and Stability in Large-Scale Numerical Simulations on GPUs*; 24th IEEE International Parallel and Distributed Processing Symposium, 2009.
19. Friedrichs, M. S.; Eastman, P.; Vaidyanathan, V.; Houston, M.; Legrand, S.; Beberg, A. L.; Ensign, D. L.; Bruns, C. M.; Pande, V. S. *J. Comp. Chem.* **2009**, *30*, 864.
20. Oh, K. J.; Klen, M. L. *Comp. Phys. Comm.* **2006**, *174*, 560.
21. http://www.nvidia.com.
22. Smith, W. *Comp. Phys. Comm.* **1991**, *62*, 229.
23. MacKerell, A., Jr.; Bashford, D.; Bellott, M.; Dunbrack, R. L., Jr.; Evanseck, J. D.; Field, M. J.; Fischer, S.; Gao, J.; Guo, H.; Ha, S.; Joseph-McCarthy, D.; Kuchnir, L.; Kuczera, K.; Lau, F. T. K.; Mattos, C.; Michnick, S.; Ngo, T.; Nguyen, D. T.; Prodhom, B.; Reiher W. E., III.; Roux, B.; Schlenkrich, M.; Smith, J. C.; Stote, R.; Straub, J.; Watanabe, M.; Wiorkiewicz-Kuczera, J.; Yin, D.; Karplus, M. *J. Phys. Chem. B* **1998**, *102*, 3586.
24. Jorgensen, W. L.; Chandrasekhar, J.; Madura, J. D.; Impey, R. W.; Klein, M. L. *J. Chem. Phys.* **1983**, *79*, 926.
25. Ryckaert, J.-P.; Ciccotti, G.; Berendsen, H. J. C. *J. Comp. Phys.* **1977**, *23*, 327.
26. Andersen, H. C. *J. Comp. Phys.* **1983**, *52*, 24.
27. Narumi, T.; Kameoka, S.; Taiji, M.; Yasuoka, K. *SIAM J. Sci. Comp.* **2008**, *30*, 3108.
28. Narumi, T.; Hamada, T.; Nitadori, K.; Sakamaki, R.; Kameoka, S.; Yasuoka, K. *High-Performance Quasi Double-Precision Method using Single-Precision Hardware for Molecular Dynamics Simulations with GPUs*; 10th International Conference on High Performance Computing in Asia-Pacific Region, 2009.