Towards Large-Scale Molecular Dynamics Simulations on Graphics Processors

Joseph E. Davis, Adnan Ozsoy,
Sandeep Patel, and Michela Taufer

University of Delaware,
Newark, DE, 19716
{jedavis,ozsoy,
sapatel,
taufer}@udel.edu

Abstract. Atomistic molecular dynamics (MD) simulations are a vital tool in chemical research, as they are able to provide a view of chemical systems and processes that is not obtainable through experiment. However, large-scale MD simulations require access to multicore clusters or supercomputers that are not always available to all researchers. Recently, many have begun to explore the power of graphics processing units (GPUs) for various applications, such as MD. We present preliminary results of water simulations carried out on GPUs. We compare the performance gained using a GPU versus the same simulation on a single CPU or multiple CPUs. We also address the use of more accurate double precision arithmetic with the newest GPUs and its cost in performance.

Key words: Molecular dynamics, GPU, CUDA

1 Introduction

For years, graphics processing units (GPUs) have been used extensively in graphics intensive applications. Their development has been driven strongly by the economy, particularly the entertainment industry, in order to meet the ever increasing demand for faster, more detailed three-dimensional (3D) graphics in media such as movies and video games. These devices are designed specifically to alleviate the load of the main CPU by taking over the expensive operations required to render detailed 3D graphics. Most of these operations are inherently data parallel, and since GPUs have been developed with this parallelization in mind, they have the potential to become powerful tools for scientific computing. However, until recently GPU hardware has been restricted to operations specific to graphics processing, limiting their usefulness in other areas. Fortunately, recent efforts have led to the development of general purpose GPUs (GPGPUs) and language libraries such as NVIDIA's Compute Unified Device Architecture (CUDA) [1].

Recent efforts have explored the potential of GPUs for mathematical, scientific, and clinical computing applications. One study has reported coupling GPUs to magnetic resonance imaging (MRI) hardware for medical diagnostics [2]. In the area of molecular modeling, GPUs have been applied to electrostatic potential calculation [3], ion placement [3], and simulations of van der Waals fluids and polymers [4]. Particularly in the areas of molecular modeling and molecular dynamics (MD), accelerating simulations by exploiting parallelism is a major concern. As with many other applications, MD has been adapted for use with massively parallel architectures such as compute clusters and supercomputers. In addition, special purpose hardware has been developed to meet these challenges, such as Protein Explorer systems with its large-scale integration (LSI) 'MDGRAPE-3 chip' [5] and Anton with its 12 identical application-specific integrated circuits (ASICs) designed for MD [6]. However, these architectures target very specific types of calculations, and in that respect are similar to early generations of GPUs. In addition, they are not widely available to most researchers. On the other hand, GPGPUs are cost effective, readily available in recent workstations, flexible, and easy to deploy.

MD simulations are an excellent target for GPU acceleration since most aspects of MD algorithms are easily parallelizable. Enhancing the performance of MD can allow the simulation of both larger time scales and larger length scales. Figure 1 illustrates some types of calculations that are possible at differing length and time scales. Ideally, simulations would be able to attain all-atom resolution on time scales of microseconds to milliseconds. With coarse-grained resolution, even longer time scales approaching seconds may be possible. However, the ultimate goal of any simulation is atomistic resolution of very large length scales over very long time scales, i.e., essentially continuum physics with atomic detail. The utilization of parallel architectures such as GPUs constitutes a step towards this goal.

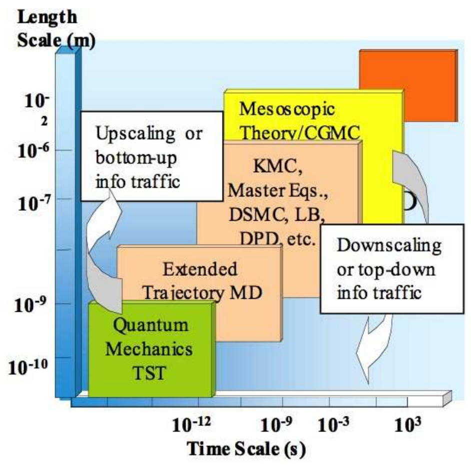

Fig. 1. An illustration of some of the types of theoretical physical calculations that are possible at varying time and length scales. Figure reproduced from Reference [7].

This contribution will discuss ongoing work in our laboratory exploring the potential applications of GPUs to atomistic MD simulations. Section 2 discusses some background related to CUDA and some related work with MD using CUDA. Section 3 describes in general our simulations on the CPU and the GPU. In addition, we outline our implementation of MD in CUDA. In Section 4 we provide further detail on the simulation parameters as well as present results from two case studies: pure water simulations and sodium iodide (NaI) solution simulations. Finally, in Section 5 we conclude and discuss future work.

2 Background and Related Work

2.1 Background

In the past, graphics application programming interfaces (APIs) such as OpenGL were the only means of accessing the computing resources of GPUs. In general, these APIs were not easy to use for those who were unfamiliar with graphics processing, as calculations essentially had to be "drawn" and then the resulting "images" interpreted to obtain meaningful results. Recently NVDIA has introduced the CUDA language library. CUDA facilitates the use of GPUs by providing a minimal set of extensions to the C programming language necessary to expose the power of GPUs for general purpose applications.

GPUs are massively parallel multithreaded devices capable of executing a large amount of active threads handled by a hardware thread execution manager that overlaps computation with communication whenever possible. There are multiple streaming multiprocessors, each of which contains multiple scalar processor cores. For example, NVIDIA's G80 GPU architecture contains 16 such multiprocessors, each of which contains 8 cores, for a total of 128 cores which can handle up to 12,288 active threads in parallel. The GPU also has several types of memory, most notably the main device memory (global memory) and on-chip shared memory accessible to all cores on a single multiprocessor. From the perspective of the CUDA programmer, the GPU is treated as a coprocessor to the main CPU. Programs are written in C and linked to the CUDA libraries. A function that executes on the GPU, called a kernel, consists of multiple threads executing code in a single instruction, multiple data (SIMD) fashion. That is, each thread in a kernel executes the same code, but on different data. Further, threads can be grouped into thread blocks. This abstraction takes advantage of the fact that threads executing on the same multiprocessor can share data via on-chip shared memory, allowing some degree of cooperation between threads in the same block.

As described above, GPU architecture is inherently different than a traditional CPU. Therefore, code optimization for the GPU involves different approaches than for the CPU. Such considerations are discussed in more detail elsewhere [1], but here we will briefly address some typical CUDA optimization strategies. First, since threads are independent, it is necessary to maximize independent parallelism, i.e., minimize the need for communication between threads. However, it is also beneficial to take advantage of the per-block shared memory, allowing a small degree of communication between threads in the same block. A balance of these two is desirable for the best performance. Second, since reading from the global memory is relatively expensive, it is often more efficient to redundantly compute a value multiple times rather than compute it once and read it from memory. In other words, maximizing the amount of arithmetic intensive computations allows the latency of communication to be hidden by overlapping with execution. Third, when dealing with global memory, it is most efficient if reads and writes are coalesced, meaning that consecutive threads act on contiguous locations in memory. Therefore it is beneficial to structure the data and algorithms in such a way that this is possible. If this is difficult or not practical, the texture cache can be used to speed up reads from memory locations that are not contiguous. Finally, transfer between the GPU global memory and the main memory of the CPU is extremely slow compared to transfer within the GPU, so this should be done only when absolutely necessary.

2.2 Related Work

In this section we will briefly review some related work involving the implementation of MD on GPUs. One of the first instances reported in the literature was a study by Yang *et al.* [8] in which an MD algorithm was programmed for a GPU using OpenGL, as general purpose GPU programming platforms such as CUDA were not yet readily available. This required translation of the MD algorithm into graphics operations. For example, values such as the atomic coordinates were encoded in the color values of pixels. However, their application to periodic, crystalline solids made neighbor list updates unnecessary, a simplification that is not generalizable to the more fluid, less structurally ordered systems of interest to our work. Overall, this study provides a proof of concept demonstrating that implementing MD on GPUs is feasible.

Since the work of Yang *et al.* there have been several studies of MD using CUDA. Stone *et al.* [3] have integrated CUDA into NAMD, adapting nonbonded force calculations for GPU while retaining the CPU implementations of all other computations. However, minimizing the transfer of data from the GPU will likely enhance performance. This was the approach of Anderson *et al.* [4], who constructed MD code from scratch in which all computations were carried out on the GPU. Their implementation was applied to simple Lennard-Jones liquids and polymer systems with Lennard-Jones and bonded potentials only. A very similar study was carried out by van Meel *et al.* [12], with the most significant difference being the use of a cell-based list structure vs. a neighbor list structure as in Anderson's study. In addition, the study by van Meel compared CUDA with Cg, a less flexible programming language for graphics processing. Our implementation is most similar to that of Anderson *et al.* in that all aspects of the MD algorithm are carried out on the GPU using a neighbor list structure for the nonbonded calculations. Unlike Anderson, we include angle and electrostatic potentials, which are necessary to simulate solvent systems such as water.

3 Methodology

3.1 Molecular Dynamics on CPU

Several molecular modeling packages are available, with CHARMM [13], GROMACS [14], and NAMD [15] among the most popular. CHARMM is one of the oldest modeling packages available, having been developed for over two decades. Today, CHARMM still has an active development community, with new simulation algorithms being incorporated constantly. As a result, CHARMM has a wide variety of algorithms, force fields, and simulation methods available compared to other packages. In addition, CHARMM has been extensively tested and validated in the many literature studies that have used it over the years. For these reasons we use CHARMM both as a model which our GPU implementation emulates and as the reference to validate the results of our algorithm.

3.2 Molecular Dynamics on GPU

Our current CUDA implementation is modeled after CHARMM in terms of the force field functional forms, neighbor list structure, and measurement units. Our water simulations use a modified version of the flexible SPC/Fw water model [9], which will be discussed in more detail below. Nonbonded interactions (Lennard-Jones and electrostatics) are calculated by a single kernel in which each thread iterates through the neighbor list for a single atom i and accumulates the interactions between i and all its neighbor list entries. The texture cache is used for reading the coordinates of the neighbor atoms since they are not contiguous in memory. Shifted force forms are used for the electrostatic and Lennard-Jones potentials so that both energies and forces go smoothly to zero at the cutoff r_{cut}. The Verlet list approach [10] is used to construct the neighbor list. Briefly, a list is constructed for each atom containing all atoms within a cutoff r_{list}, where $r_{list} > r_{cut}$. This way, the list only needs updating whenever an atom has moved more than $\frac{1}{2}(r_{list} - r_{cut})$. The list is constructed on the GPU as follows. Each thread checks the distance between an atom i and all other atoms, and adds to i's neighbor list those atoms that are within r_{list} of i. This process is accelerated by having each block take advantage of shared memory using a previously described tiling approach [11].

Bond and angle potentials are computed using a similar list approach. For the bonds, each thread iterates through all atoms bonded to an atom i and accumulates the total bond forces. For the angles, each thread iterates through the atoms which are involved in an angle with i and calculates the appropriate interactions. Unlike the nonbonded lists, these lists are constructed once on the CPU at the beginning of the simulation, copied to the GPU, and never need to be modified.

4 Computational Experiments

4.1 Platforms

Three different NVIDIA GPUs were used for our simulations: Quadro FX 5600 (1.5GB memory, single precision), GeForce 9800 GX2 (2 GPUs per card, 512MB memory, single precision), and GTX 280 (1GB memory, double precision). For our first case study, the pure solvent systems, the GeForce 9800 GX2 was used. For the second case, the ionic solutions, performance was compared between all GPUs, using double precision arithmetic on the GTX 280 and single precision on the others.

Our reference simulations were run on CHARMM on a Beowulf cluster consisting of compute nodes with Intel Xeon 5150 2.66 GHz (Woodcrest) CPUs. For the pure solvent simulations a single CPU was used, while for the ionic solution performance was compared with 1, 2, 4, and 8 CPUs.

4.2 Solvent Simulation

In our simulations we use a modified version of the flexible SPC/Fw water model developed by Wu *et al.* [9]. Briefly, the intramolecular potential is:

$$V_{intra} = \frac{k_b}{2}\left[\left(r_{OH_1} - r^0_{OH}\right)^2 + \left(r_{OH_2} - r^0_{OH}\right)^2\right] + \frac{k_a}{2}\left(\theta_{\angle HOH} - \theta^0_{\angle HOH}\right)^2 \quad (1)$$

As described in Section 3.2, this potential is evaluated on the GPU using bond and angle lists. The general intermolecular potential consists of a Lennard-Jones potential modeling the van der Waals forces and a Coulomb potential for the electrostatic interactions:

$$V_{inter} = \sum_{i,j}^{pairs}\left(4\epsilon_{ij}\left[\left(\frac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}}\right)^{6}\right] + \frac{q_i q_j}{r_{ij}}\right) \quad (2)$$

As mentioned in Section 3.2, this potential is shifted smoothly to zero at the cutoff and pairs separated by a distance greater than a cutoff r_{cut} are neglected. Furthermore, we have modified the Lennard-Jones parameters slightly to account for the fact that we neglect long-range electrostatics typically computed using an Ewald sum.

Starting configurations for our simulations were pre-equilibrated with CHARMM using the NVT (constant number of particles, constant volume, constant temperature) ensemble. Simulations were then run from these starting configurations using our CUDA MD code for GPUs using the NVE (constant number of particles, constant volume, constant energy) ensemble. A Verlet integrator with 1 femtosecond (fs) timestep was used both on the GPU and in CHARMM to integrate the equations of motion. Pure solvent systems consisted of cubic periodic boxes equilibrated to a density of 1.012 g/mL, the calculated equilibrium density for the SPC/Fw model [9]. Several different system sizes were used to determine the effect of system size on performance. Figure 2 shows one of the cubic water boxes used. To measure the performance, we simulated 100,000 MD steps using both CHARMM (on a single processor) and our GPU code. For the GPU simulations, only one type of GPU, the GeForce 9800 GX2, was used. Five system sizes were used, consisting of 233, 826, 1981, 3921, and 6845 water molecules. From the total execution time for each simulation, we determined the number of MD steps per second. This data is shown in Table 1. On average, our GPU

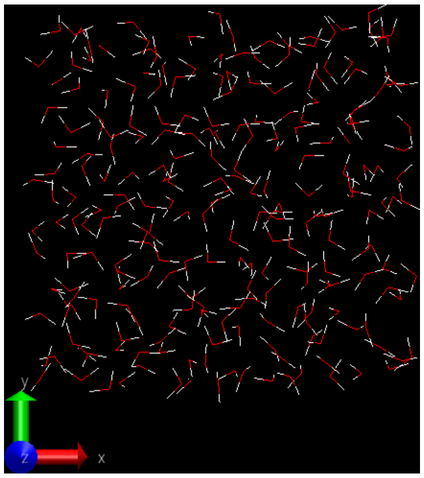

Fig. 2. A snapshot of the cubic water system containing 233 water molecules.

Table 1. Performance of our GPU code for the pure water systems compared to CHARMM on a single CPU. GPU simulations were run on the GeForce 9800 GX2. The total execution time for 100,000 MD steps was used to obtain this data.

# waters	# atoms	steps/sec (CHARMM)	steps/sec (GPU)
233	699	165.34	609.09
826	2478	43.49	271.2
1981	5943	17.25	159.12
3921	11763	8.52	72.32
6845	20535	4.73	32.47

code is ∼7x faster than CHARMM on a single CPU.

To ensure that our simulation results are accurate, we ran longer simulations (several million MD steps for a simulation time of several nanoseconds) with the 233 water system on both CHARMM and our GPU code. Figure 3 plots the temperature (a function of kinetic energy) and total potential energy over the simulation time. Both quantities fluctuate, as expected, around the same average value, indicating that our GPU implementation does in fact produce results consistent with CHARMM.

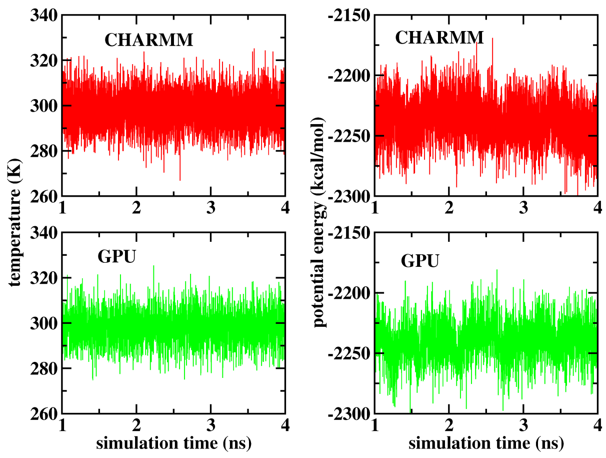

Fig. 3. Energy and temperature profiles over time for the 233 water system simulated using CHARMM and our CUDA code on a GPU.

4.3 Ionic Solutions

For our second case study we simulated NaI solutions with a liquid-vapor interface. Only nonbonded interactions are applicable to the ions. Lennard-Jones parameters for sodium and iodide were transferred from the CHARMM force field and modified slightly to better reproduce *ab initio* water interaction energies [16] with our modified SPC/Fw water model. NaI solution systems consisted of a non-cubic periodic box (larger z-dimension to create a vapor interface), with simulation parameters otherwise identical to those of the pure water simulations Here we compared performance among different GPUs and different numbers of CPUs for a single system size, which consisted of 988 water molecules, 18 Na^+ ions and 18 I^- ions, for a total of 3000 atoms. Figure 4 shows a snapshot of this system. As with the previous case study we measure performance by the number of MD steps per second.

First, we compare the performance of three NVIDIA GPUs: Quadro FX 5600, GeForce 9800 GX2, and GTX 280. This data is shown in Table 2. Several observations are apparent from the data in Table 2. Both single precision GPUs perform about equally well, with the GeForce 9800 GX2 achieving ∼6% better performance. As with the pure water simulations, the speedup of the single precision GPUs over CHARMM on a single CPU is ∼7x. Regarding the double precision simulation run on the GTX 280, performance drops by a factor of

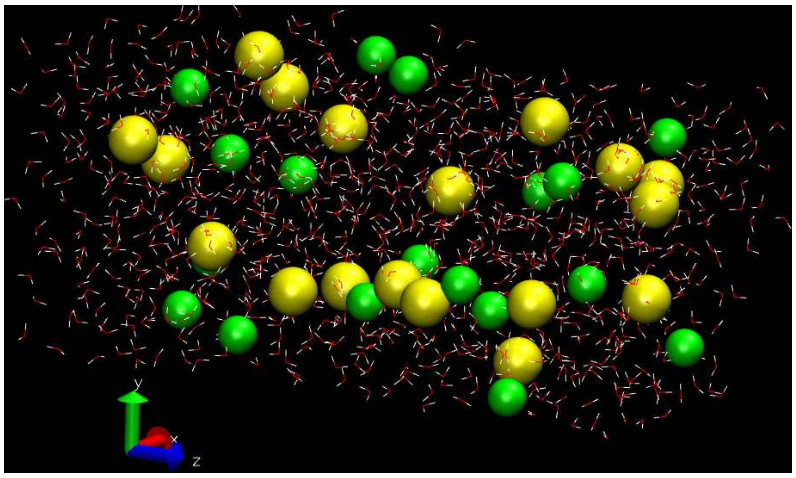

Fig. 4. Snapshot of the NaI solution system containing 988 waters, 18 Na^+, and 18 I^-.

Table 2. Performance of our GPU code for the NaI solution system compared among different GPUs and CHARMM on 1, 2, 4, and 8 CPUs. The total execution time for 10 million MD steps (10 nanoseconds simulation time) was used to obtain this data.

	steps/sec
GeForce 9800 GX2	260.88
Quadro FX 5600	246.37
GTX 280	31.79
CHARMM (1 CPU)	34.34
CHARMM (2 CPUs)	64.95
CHARMM (4 CPUs)	116.62
CHARMM (8 CPUs)	186.05

∼8 compared to the single precision. This performance is approximately equal to that of CHARMM running on a single CPU. We note, however, that not all of the optimizations we implemented with single precision were possible with double precision. Specifically, double precision does not support texture memory in the same way that single precision does. Reading coordinates from the texture memory accounts for a speedup of ∼2–3x in our single precision code. Therefore, the performance cost of double precision relative to single precision is closer to ∼3–4x. Even with this consideration, the performance cost of using double precision is still significant. Finally, we note that the single precision GPUs perform better than CHARMM even on 8 CPUs. By linear extrapolation, we estimate that the performance of the single precision GPUs is approximately equivalent to that of CHARMM running on 12 CPUs.

5 Conclusions

We have presented results of all-atom MD simulations implemented on NVIDIA GPUs using CUDA. Despite the fact that our implementation is relatively naïve and straightforward, we have achieved promising results in terms of performance. Our GPU implementation performs ∼7x faster than CHARMM on a single CPU, which is a speedup approximately equivalent to CHARMM on 12 CPUs. Looking to the future, we plan to expand the MD options possible by adding potential energy terms such as dihedrals as well as additional algorithms such as Ewald summation to account for long-range electrostatics. Furthermore, we anticipate that more code optimizations are possible, leading to even more efficient performance. Finally, our ultimate goal is to integrate some GPU acceleration directly into CHARMM, which will be made possible with the upcoming FORTRAN compiler and libraries for CUDA, towards the study of very large systems for long simulation times on the order of 100 nanoseconds.

Acknowledgments This work was supported by the National Science Foundation, grant #SCI-0802650, DAPLDS - a Dynamically Adaptive Protein-Ligand Docking System based on multi-scale modeling, and the NVIDIA University Professor Partnership Program. The authors also wish to thank Joshua Anderson and Scott LeGrand for helpful advice.

References

1. NVIDIA CUDA Programming Guide 2.0, NVIDIA, 2008.
2. Stone, S. S., Haldar, J. P., Tsao, S. C., Hwu, W. W., Liang, Z, Sutton, B. P.: Accelerating advanced MRI reconstructions on GPUs. Proc. of 2008 Computing Frontiers Conference, 7-9 May 2008.
3. Stone, J. E., Phillips, J. C., Freddolino, P. L., Hardy, D. J., Trabuco, L. G., Schulten, K.: Accelerating molecular modeling applications with graphics processors. J. Comput. Chem. **28** (2007) 2618.

4. Anderson, J. A., Lorenz, C. D., Travesset, A.: General purpose molecular dynamics simulations fully implemented on graphics processing units. J. Comput. Phys. **227** (2008) 5342.
5. Taiji, M., Narumi, T., Ohno, Y., Futatsugi, N., Suenaga, A., Takada, N., Konagaya, A.: Protein Explorer: A petaflops special-purpose computer system for molecular dynamics simulations. Proc. of 2003 ACM/IEEE Supercomputing Conference, 15-21 Nov 2003.
6. Shaw, D. E., Deneroff, M. M., Dror, R. O., Kuskin, J. S., Larson, R. H., Salmon, J. K., Young, C., Batson, B., Bowers, K. J., Chao, J. C., Eastwood, M. P., Gagliardo, J., Grossman, J. P., Ho, C. R., Ierardi, D. J., Kolossváry, I., Klepeis, J. L., Layman, T., McLeavey, C., Moraes, M. A., Mueller, R., Priest, E. C., Shan, Y., Spengler, J., Theobald, M., Towles, B., Wang, S. C.: Anton: A special-purpose machine for molecular dynamics simulation. Proc. of the 34th Annual International Symposium on Computer Architecture, 9-13 June 2007.
7. Vlachos, D. G.: A review of multiscale analysis: Examples from systems biology, materials engineering, and other fluid-surface interacting systems. Adv. Chem. Eng. **30**, 1–61, invited (2005).
8. Yang, J., Wang, Y., Chen, Y.: GPU accelerated molecular dynamics simulation of thermal conductivities. J. Comput. Phys. **221** (2006) 799.
9. Wu, Y., Tepper, H. L., Voth, G.: Flexible simple point-charge water model with improved liquid-state properties. J. Chem. Phys. **124** (2006) 024503.
10. Allen, M. P., Tildesley, D. J.: Computer Simulation of Liquids, Oxford: Clarendon Press, 1987.
11. Nguyen, Herbert (ed.): GPU Gems 3, Boston: Addison-Wesley, 2008, chapter 31.
12. van Meel, J. A., Arnold, A., Frenkel, D., Zwart, S. F. P., Belleman, R. G.: Harvesting graphics power for MD simulations. Mol. Sim. **34** (2008) 259.
13. Brooks, B. R., Bruccoleri, R. E., Olafson, B. D., States, D. J., Swaminathan, S., Karplus, M.: CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. J. Comput. Chem. **4** (1983) 187.
14. Lindahl, E., Hess, B., van der Spoel, D.: GROMACS 3.0: A package for molecular simulaton and trajectory analysis. J. Mol. Model. **7** (2001) 306.
15. Phillips, J. C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R. D., Kalé, L., Schulten, K.: Scalable molecular dynamics with NAMD. J. Comput. Chem. **26** (2005) 1781.
16. Lamoureux, G., Roux, B.: Absolute hydration free energy scale for alkali and halide ions established from simulations with a polarizable force field. J. Phys. Chem. B. **110** (2006) 3308.