Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA

Weiguo Liu, Bertil Schmidt*, Gerrit Voss, Wolfgang Müller-Wittig

School of Computer Engineering, Nanyang Technological University, Singapore 639798

A R T I C L E   I N F O

Article history:
Received 29 January 2008
Received in revised form 9 May 2008
Accepted 27 May 2008
Available online 6 June 2008

Keywords:
Graphics processing unit
Molecular dynamics
Advanced computer architecture

A B S T R A C T

Molecular dynamics is an important computational tool to simulate and understand biochemical processes at the atomic level. However, accurate simulation of processes such as protein folding requires a large number of both atoms and time steps. This in turn leads to huge runtime requirements. Hence, finding fast solutions is of highest importance to research. In this paper we present a new approach to accelerate molecular dynamics simulations with inexpensive commodity graphics hardware. To derive an efficient mapping onto this type of computer architecture, we have used the new Compute Unified Device Architecture programming interface to implement a new parallel algorithm. Our experimental results show that the graphics card based approach allows speedups of up to factor nineteen compared to the corresponding sequential implementation.

1. Introduction

The fast increasing power of the Graphics Processing Unit (GPU) and its streaming architecture opens up a range of new possibilities for a variety of applications. With the enhanced programmability of commodity GPUs, these chips are now capable of performing more than the specific graphics computations they were originally designed for. Recent work shows the design and implementation of algorithms for non-graphics applications. Examples include Quantum Monte Carlo [5], spectroscopic [10] and gravitational simulations [7] in the area computational physics and sequence analysis in the area of computational biology [16,21]. The evolution of GPUs is driven by the computer game market. This leads to a relatively small price per unit and to very rapid developments of next generations. Currently, the peak performance of state-of-the-art GPUs is approximately ten times faster than that of comparable CPUs. Furthermore, the growth rate of the number of transistors used on GPUs is greater than for microprocessors. Consequently, GPUs will become an even more attractive alternative for high performance computing in the near future.

The Compute Unified Device Architecture (CUDA) [2] is a new hardware and software architecture for issuing and managing computations on GPUs. It treats the GPU as a data-parallel computing device without the need of mapping computations to the graphics pipeline. By using the standard C language, CUDA simplifies GPU-based software development for scientific computing compared to previously used graphics-oriented languages such as OpenGL or Cg.

Molecular dynamics (MD) is a computationally intensive method of studying the natural time-evolution of a system of atoms using Newton’s classical equations of motion. In practice, MD has always been limited more by the current available computing power than by investigators’ ingenuity. Researchers in this field have typically focused their efforts on simplifying models and identifying what may be neglected while still obtaining acceptable results. This has led to much skepticism on the ability of MD to be used as a predictive tool for experimental work. High performance computing holds the key to making biologically relevant calculations tractable without compromise. In this paper we show how MD simulations can benefit from the computing power of GPUs. In order to exploit the GPU’s capabilities for high performance MD simulation, we present a new algorithm for non-bonded short-range interactions within the atom system. The algorithm has been implemented using C++ and CUDA and tested on a physical system of up to 131,072 atoms. We show that our new MD algorithm leads to a performance improvement of one order of magnitude on a commodity NVIDIA GeForce 8800 GTX card.

The rest of this paper is organized as follows. In Section 2, we introduce the basic MD simulation algorithm. Previous work on parallelization of this algorithm on different computer architectures is discussed in Section 3. Important features of the CUDA programming model are described in Section 4. Section 5 presents our parallel CUDA-based MD algorithm and its efficient implementation. Performance is evaluated in Section 6. Section 7 concludes the paper with an outlook to further research topics.

* Corresponding author.
E-mail addresses: liuw@ntu.edu.sg (W. Liu), asbschmidt@ntu.edu.sg (B. Schmidt), agerrit@ntu.edu.sg (G. Voss), askwmwittig@ntu.edu.sg (W. Müller-Wittig).
2. Molecular dynamics simulations

Computer simulations play a very important role in scientific research. They act as bridges among microscopic length, time scales and the macroscopic world of the laboratory. In very broad terms, we can identify two categories of computer simulation techniques: MD and Monte Carlo (MC). In contrast with the MC method, MD simulates the time evolution and provides us with the actual trajectory information of the molecular system. In an MD simulation, the time evolution of an atomic system is followed by integrating their equations of motion described by the following classical equations of motion:

\[
\begin{align*}
    F_i &= m_i a_i, \\
    F_i &= -\nabla_v V(r_1, \ldots, r_N).
\end{align*}
\]

In Eq. (2.1), the atomic system contains \( N \) atoms. \( m_i \) is the atom mass, \( a_i = \frac{d^2 r_i}{dt^2} \) is its acceleration, and \( F_i \) is the force acting upon it. \( V(r_1, \ldots, r_N) \) is the function of the positions of the atoms. It represents the potential energy of the system. In practice, function \( V \) can be written as a sum of pairwise interactions:

\[
V(r_1, \ldots, r_N) = \sum_{i} \sum_{j \neq i} u_2(r_i, r_j) + \sum_{i} \sum_{j} \sum_{k \neq i} u_3(r_i, r_j, r_k) + \cdots.
\]

In Eq. (2.2), the three body (and higher order) interactions are usually neglected [4], only leaving the pair potential as the concentration of the simulation. In this case, the potential terms in Eq. (2.2) are typically non-linear functions of the distance \( r_{ij} \) between pairs of atoms and may be either bonded or non-bonded in nature. The bonded terms describe energy models caused by covalent bonds within molecules [25]. They are associated with chemical bonds, bond angles, and bond dihedrals. The non-bonded terms are associated with the short-range van der Waals and the long-range electrostatic interactions.

In this paper, we are only concerned with the non-bonded short-range force models because they are used extensively in MD [20]. In practice, the Lennard-Jones (LJ) potential [15] is the most commonly used short-range interaction model. It is given by the following expression:

\[
u(r) = 4\varepsilon \left[ \left( \frac{\sigma}{r} \right)^12 - \left( \frac{\sigma}{r} \right)^6 \right].
\]

where \( r \) is the distance between two interacting atoms, \( \sigma \) is the diameter and \( \varepsilon \) is the well depth. Both \( \varepsilon \) and \( \sigma \) are constants and they are chosen to fit the physical properties of the material.

One of the most time-consuming parts in MD simulations is the computation of interaction forces, which usually takes more than 90% of the total simulation time. From Eqs. (2.2) and (2.3) we can see this is mainly because the force computation requires to calculate the interactions between each atom in the system with every other atom, giving rise to \( O(N^2) \) evaluations of the interaction in each time step. The interaction forces decrease rapidly with increasing distance between atoms. Thus, it is possible to neglect forces between atoms separated by more than a cutoff distance \( r_c \). This means an atom has only interaction forces with atoms that are in a sphere with a radius equal to \( r_c \) [4]. The cutoff method is also called the neighbor list method. It reduces the computational complexity to \( O(N) \). Forces computed using the cutoff method are also called short-range forces.

Fig. 1 illustrates how to reduce computational complexity by using the cutoff method. When the neighbor list is built, all of the nearby atoms within an extended cutoff distance \( r_{\text{cut}} = r_c + \text{skin} \) are stored. At the first step in an MD simulation, the neighbor list is constructed for all the neighbors of each atom. From time to time the list needs to be reconstructed.

3. Previous work on accelerating molecular dynamics

There have been a variety of techniques used to accelerate MD simulations on parallel computer architectures. They range from typical high performance computing (HPC) strategies such as clustering to novel processing architectures. In this section we discuss the various strategies used in accelerating MD. We broadly classify these strategies into two categories: coarse-grained and fine-grained.

The architectures in the coarse-grained category include general-purpose supercomputers, PC clusters and computational grids. Using supercomputers such as Blue Gene [13] for MD can provide tremendous performance at the cost of being overly expensive and inaccessible to most researchers. Therefore, commodity PC clusters [8] and grids [19,24] have been used to provide more accessible high performance at lower cost. However, the cluster approach suffers from scalability issues for a growing number of processors due to the high latencies for communicating between PCs. Grid-based projects such as Folding@Home [19] and Predictor@home [24] have attracted several hundred thousand volunteered PCs across the world. Unfortunately, these volunteered compute resources do not allow for communication between clients but only between server and clients. Hence, this approach is only suitable for MD simulations with a large number of separate trajectories with short time scales.

Computer architectures in the fine-grained category include special-purpose architectures, reconfigurable architectures and GPUs. Special-purpose architectures such as Anton [22], FASTRUN [12], MDGRAPE [18] and MD Engine [25] can provide the fastest means of running a particular MD algorithm with very high arithmetic density. Each arithmetic unit can be specifically designed for the specific force calculation. However, such architectures are limited to one single algorithm, and thus cannot supply the flexibility necessary to run a variety of algorithms required for MD simulations.

Reconfigurable systems are based on programmable logic such as field-programmable gate arrays (FPGAs). They are generally faster than special-purpose MD architectures [3]. They are flexible, but the configuration must be changed for each algorithm, which is generally more complicated than writing code for a programmable architecture.

All these approaches can be seen as accelerators—an approach satisfying the demand for a low cost solution to compute-intensive problems. The main advantage of GPUs compared to the architectures mentioned above is that they are commodity components. In particular, most users have already access to PCs with modern graphics cards. For these users this direction provides a zero-cost solution. Even if a graphics card has to be bought, the installation of such a card is trivial (plug and play). Writing the software
for such a card does still require specialist knowledge, but new high-level programming models such as CUDA [2] offer a simplified C-based programming environment.

Stone et al. [23], Anderson et al. [6], Hamada et al. [14], Bellemann et al. [7] and Yang et al. [26] also implemented MD simulations on GPUs. Stone uses the spatial bin method to accelerate molecular modeling applications with CUDA. By presetting the atoms into bins based on their coordinates, only atoms in neighboring bins need to be loaded onto the GPU. Thus, this method does not require the GPU to build neighbor lists and uses minimal GPU memory and bandwidth. The implementation by Anderson is close to the approach presented in this paper. Anderson models simple Lennard-Jones particles and maps every step of MD on the GPU. Hamada and Bellemann both use CUDA to parallellize gravitational N-body simulations on GPUs, while Yang uses Cg for MD simulations. However, these implementations merely map the computation of pairwise particle interactions onto the GPU. This makes the time-consuming updating of the neighbor lists on the CPU a bottleneck since synchronization and frequent data transfer between CPU and GPU can often be problematic for GPGPU implementations. The solution presented in this paper overcomes this bottleneck by computing both the atomic interactions as well as the neighbor lists on the GPU using a single kernel. Experiments show that the creation and utilization of neighbor lists can be efficiently parallelized on a GPU using our approach.

4. The CUDA programming model

The Compute Unified Device Architecture (CUDA) is a programming model for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API [1]. For now, it is available for NVIDIA 8800 series, NVIDIA Quadro FX 5600/4600, and beyond. From a hardware point of view, CUDA treats the GPU as a set of SIMD multiprocessors. Each multiprocessor is composed of eight processors. The multiprocessor specifications of NVIDIA 8800 series and Quadro FX 5600/4600 are shown in Table 1.

A multiprocessor has on-chip memory of four types:

1. a set of registers per processor,
2. a parallel data cache or shared memory,
3. a read-only constant cache,
4. a read-only texture cache.

These on-chip memories are used to implement fast I/O operations, especially, to speed up read and write accesses to the non-cached device memory (see Fig. 2). Thus, applications can take advantage of them by minimizing over-fetch and round-trips to the low bandwidth device memory. Although the device memory has a low bandwidth, it is big in size and shared by all multiprocessors.

In the CUDA programming model, each multiprocessor is viewed as a multi-core device that is capable of executing a very high number of threads in parallel. These threads are organized as thread blocks. Threads in the same thread block can cooperate together by efficiently sharing data and synchronizing their execution to coordinate memory access with other threads. However, threads in different thread blocks cannot communicate or synchronize with each other. Theoretically, having more active threads per multiprocessor can help hiding memory latency, and can also better fill the instruction pipeline so there are no idle processors. According to [1], the maximum number of threads that can run concurrently on a multiprocessor is 768. In practice, the number of threads is further limited by the shared on-chip memory and hence, the maximal number of threads is application-dependent.

Fig. 3 shows the general structure of the CUDA software stack. It contains a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries of common usage [1]. The data-parallel, compute-intensive portion of a CUDA-based application is isolated into a kernel program and is loaded onto the GPU hardware to execute.

5. The CUDA-based MD simulation algorithm

Many parallel algorithms for MD simulations have been proposed and implemented by different researchers. The details of these algorithms vary widely since there are numerous application-dependent and architecture-dependent characteristics to consider. Generally, from the point of view of data decomposition, they can be categorized into three types:

1. Atom-decomposition (AD). Each processor is assigned a subset of $N/P$ ($N$ is the number atoms; $P$ is the number of processors) atoms at the beginning of the simulation. As each processor must keep identical copies of atom information, it is also called replicated-data method [20]. The AD method has been widely used especially on shared memory architectures.
In this section we describe how MD simulations can be efficiently mapped onto a GPU using CUDA. We take advantage of the inherent parallelism of MD simulations and design parallel algorithms using the AD method. The main reasons we choose the AD method to design our algorithms are

- good load balancing and scalability can be easily achieved,
- according to the CUDA model as described in Section 3, the GPU hardware is viewed as a shared memory multiprocessor system, the AD method can give good performance in such a system.

The algorithm outline in Fig. 4 illustrates how a sequential MD simulation works. Fig. 4 also states the computational complexity of each operation. It can be seen that there are six main operations (shown in bold characters) in a sequential MD simulation. We have profiled these six operations for different numbers of partitions (shown in bold characters) in a sequential MD simulation. It can be seen that there are six main operations listed, where

\[
\begin{array}{cccccc}
\text{Number of operations} & \text{Force calculation} & \text{Neighbor list update} & \text{Position update} & \text{Moved too much} & \text{Velocity update} \\
8192 & 70.4\% & 26.9\% & 0.7\% & 0.5\% & 1.1\% & 0.4\% \\
16384 & 58.2\% & 40.1\% & 0.5\% & 0.5\% & 0.5\% & 0.2\% \\
32768 & 46.8\% & 62.6\% & 0.7\% & 0.2\% & 0.1\% & 0.2\% \\
65536 & 34.7\% & 64.6\% & 0.2\% & 0.2\% & 0.2\% & 0.1\% \\
131072 & 15.2\% & 84.3\% & 0.2% & 0.12% & 0.1% & 0.08% \\
\end{array}
\]

The neighbor list update step (Step (4) in Fig. 4) constructs a list of all neighbors for each atom. This requires a large number of pairwise calculations: each atom has to loop over all other atoms to compute the pairwise distance between them. This corresponds to computing distance matrix \(D = [r_{ij}]\) of size \(N \times N\). Since \(r_{ij} = r_{ji}\), this matrix is symmetric and therefore the amount of calculation can be reduced by half. Fig. 5 illustrates the pairwise distance calculation for each column of \(D\). The red atom in each column (the red circle), a loop over all other atoms (gray circles) will be done to calculate the pairwise distance. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Algorithm outline of a sequential MD simulation (with the computation complexity listed, where \(N\) denotes the number of atoms).

1. Initialize atoms' status and the LJ potential table; set parameters controlling the simulation; \(O(N)\)
2. For all time steps do
3. Update positions of all atoms (Position update); \(O(N)\)
4. If there are atoms that have moved too much, do (Moved too much); \(O(N)\)
5. Update the neighbor list, including all atom pairs that are within a distance \(r_{\text{cut}}\) (Neighbor list update); \(O(N^2)\)
6. End if;
7. Make use of the neighbor list to compute forces acted on all atoms (Force calculation); \(O(N)\)
8. Update velocities of all atoms (Velocity update); \(O(N)\)
9. Update the displace list, which contains the displacements of atoms (Displace list update); \(O(N)\)
10. Accumulate and output target statistics of each time step; \(O(N)\)
11. End for

Fig. 4. Algorithm outline of a sequential MD simulation (with the computation complexity listed, where \(N\) denotes the number of atoms).

Fig. 5. Illustrations of neighbor list construction step. For the head atom in each column (the red circle), a loop over all other atoms (gray circles) will be done to calculate the pairwise distance. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Fig. 6. Example of the partitioning of the distance matrix \(D\) for parallel neighbor list update with CUDA. The example uses one thread per column of \(D\) and each thread block consists of two threads.

Fig. 7. CUDA-based neighbor list update kernel.
different thread blocks. Therefore, we have decided to use a partitioning of \( D \) into threads and thread blocks as illustrated in Fig. 6: each column is assigned to a single thread and there are two threads in a thread block. Note that in Fig. 6 we not only compute the lower triangular matrix of \( D \) but the complete matrix. If we would only calculate the lower triangular matrix, then all threads, except for Thread 1, need to communicate and synchronize with other threads to complete their local neighbor list. For instance, Thread 4 would not know whether atoms 1, 2 and 3 are in the local neighbor list. In order get this information Thread 4 would need to access the local neighbor lists of atoms 1, 2 and 3. In CUDA, this access would require costly communication and synchronization between threads of different thread blocks. In order to solve this problem efficiently with CUDA, we let each thread loop over all other atoms for current head atom. That is, in Fig. 6 both the lower triangular and upper triangular matrices are calculated. Fig. 7 shows the corresponding kernel program for neighbor list update using CUDA. The coordinates of the head atom of each thread are stored in registers in order to minimize access time.

After the neighbor list update step, the indices of all eligible atoms will be stored in a neighbor list array in the device memory for later usage. During the compute force step, each thread will loop over the local neighbor lists to do force calculations. This is illustrated in Fig. 8 using one neighbor list per thread.

Fig. 9 shows our CUDA-based kernel program for the force computation. The coordinates of head atoms and the forces acting on them are frequently used in the inner loop and therefore stored in registers in order to optimize their access efficiency. The results of force computations \( f_i \) are written to dynamically allocated shared memory, since they will be used by other operations, such as the position and velocity update operations (steps (3) and (6) in Fig. 4).

5.2. Kernel integration

In the previous subsection we have presented our CUDA kernels for neighbor list update and force calculation. Since there are multiple time steps in an MD simulation, these two kernels need to be executed repeatedly. In practice, runtime overhead such as kernel program initialization, data transfer to GPU, and results read back are introduced by kernel launches. According to our experiments these overhead cause a great performance loss.

Table 3 shows the overhead introduced by kernel launches. In order to eliminate these overhead, we have integrated all time steps into a single kernel. Thus, there is only a single kernel launch, which significantly reduces the overhead. This method is illustrated in Fig. 10. As the kernel cannot output results directly, all results have to be read back to CPU for further processing and outputting.

In Fig. 10, the six main operations of an MD simulation have been shown in bold characters. Among them, the *Moved too much* step needs special attention and is further described in Fig. 11. In order to integrate this step into a CUDA program with a single kernel, we first need to partition the displace list onto multiple thread blocks. In order to find the largest two displacements, we use a tree-based approach (see Fig. 12). In Fig. 12 communication between thread blocks is necessary. However, as mentioned before, in CUDA thread blocks cannot communicate or synchronize with each other efficiently. In [2], Mark Harris proposes a kernel decomposition method to try to solve this problem. By decomposing...
(1) For all atoms do
(2) Scan the displace list to find the magnitude of the two largest displacements $disp_1$ and $disp_2$ in the system;
(3) End for
(4) If $disp_1 + disp_2$ is larger than $skin$ then returns TRUE, otherwise returns FALSE;

Fig. 11. The Moved too much step.

<table>
<thead>
<tr>
<th>Thread Block 1</th>
<th>Thread Block 2</th>
<th>Thread Block 3</th>
<th>Thread Block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>$D_2$</td>
<td>$D_3$</td>
<td>$D_4$</td>
</tr>
<tr>
<td>$D_5$</td>
<td>$D_6$</td>
<td>$D_7$</td>
<td>$D_8$</td>
</tr>
<tr>
<td>$D_9$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 12. Tree-based approach used within each thread block.

displacement into multiple kernel invocations, thread block communication can be avoided. But this method cannot be used in our implementation since there is only one kernel in our case.

In order to implement the Moved too much step efficiently in a single kernel, we have therefore designed a two-stage tree-based method (see Fig. 13). In the first stage all thread blocks run in parallel to find the local maximum displacement value. They then write all the values into a global memory array in parallel. Next all values in the global memory array are fetched into a shared memory array in the first thread block. At last the first thread block will execute the tree-based approach locally to find the two largest displacement values $disp_1$ and $disp_2$.

6. Performance evaluation

We have implemented the proposed algorithm using CUDA Toolkit 1.1 [2] and evaluated it on the following setup:

- **Nvidia GeForce 8800 GTX**: 1.35 GHz engine clock speed, 16 multiprocessors, 768 MB device memory, 16 KB shared memory/multiprocessor.
- Tests have been conducted with this card installed in a PC with an AMD Opteron 2210 1.8 GHz, 2 GB RAM running Windows XP.

A set of performance evaluation tests have been conducted using different cutoff distances and particles to evaluate the processing time of the GPU implementation versus that of the sequential MD simulation on a PC. The MD simulation program is benchmarked on an Intel Pentium4 3 GHz processor with 1 GB RAM. We have modified the MD code (md3.f90) from Ercolessi ([11], available online at http://www.fisica.uniud.it/~ercolessi/md/f90/) into a 32 bit C++ version for our evaluation. Optimization techniques such as explicitly arranged arrays and unrolled loops have been used on the C++ code to allow the use of SSE instructions for peak performance.

Table 4 reports the performance of the sequential MD and our CUDA implementation for the number of particles ranging from 8192 to 131,072. The sequential MD code is compiled with the Intel C/C++ Compiler (ICC) professional version 10.1 for Windows. The CUDA code is compiled with the CUDA Toolkit 1.1 for Windows. Both of these tests were performed with the Full Optimization (/Ox) enabled for the corresponding compiler. All measurements have used 100 time steps and $skin = 0.5\delta$. The cutoff distances $r_c$ ranges from 2.5$\delta$ to 4.5$\delta$. From Table 4 we can make two observations:

1. the speedup improves with a larger number of particles.
2. the speedup drops slightly for larger cutoff distance values.

There are two reasons for Observation 1: Firstly, there is higher arithmetic intensity for a larger number of particles. Secondly, the relative influence of the kernel overhead is smaller for larger particle systems. A larger cutoff distance on the other hand can increase the variation in neighbor lists sizes and therefore can introduce additional load imbalance between threads, which explains Observation 2.

In our experiments, most systems do the neighbor list update step only once despite different cutoff distances used. This is why for the system with particle number 8192, 16384, 32768, 65536 or 131072 it takes similar runtime for the N.L.U. step in Table 4.

Data transfer between the CPU and GPU is a known bottleneck for many GPGPU applications and therefore should be minimized. This bottleneck is caused by the relatively low PCI Express bus bandwidth as well as the overhead associated with initializing each transfer. From Fig. 10 we can see that in our method we only need to do the data uploading (step (2) in Fig. 10) and data readback (step (11) in Fig. 10) once. In the data uploading step, system information data such as particles’ coordinates and velocities are loaded onto the GPU. This data is small in size and can be transferred in a few of milliseconds. During the data readback step, the intermedi-
ate results such as the kinetic energy and potential energy of each time step are read back to the CPU. The maximum intermediate results to be read back are 150 MB in size (for the physical system with 131,072 particles) and this transfer can be done in less than 50 ms (given a maximum data transfer rate of 4 GB/s). Data transfer, together with the kernel program initialization, kernel launch and release constitute the kernel overhead. From Table 4 we can see the kernel overhead is greatly minimized in practice.

In order to compare our GPU version to a well-optimized sequential code, we have also compared our CUDA implementation to LAMMPS ([20], available online at http://lammps.sandia.gov/). The LAMMPS code is compiled with GNU GCC 4.1.1 with the Full Optimization (-O3) enabled. The optimized Lennard-Jones routines from style_opt are used in order to get the best possible performance out of LAMMPS. The LAMMPS applications were benchmarked on an Intel Pentium 4 3 GHz processor with 1 GB RAM. Table 5 shows the performance comparison results. As can be seen, our implementation achieves speedups of up to factor 11 compared to LAMMPS.

Currently, the utilized Nvidia GeForce 8800 GTX card implements single-precision IEEE-754 floating point arithmetic, but with some deviations [1]. In practice this will cause some differences in MD computing results between GPU and CPU. The output of our MD simulation program includes static values for the temperature, kinetic energy, potential energy and pressure of the physical system. According to our experiments in Table 4, the differences between the output values of our sequential C++ version and our CUDA version are less than 0.5%. Higher precision results can be achieved with the availability of the new generation GPUs that support native double-precision calculations in the near future.

7. Conclusions and future work

In this paper we have introduced a parallel CUDA-based MD simulation algorithm that can be efficiently implemented on modern graphics hardware. We have made use of the fast on-chip memory to design and implement this algorithm. All key components of our algorithm have been mapped onto the GPU for execution in a single kernel, which reduces overheads significantly.
The evaluation of our implementation on a mass-produced graphics card shows speedups of up to factor 11 compared to LAMMPS on a Pentium IV 3.0 GHz. Our results are especially encouraging. The evaluation of our implementation on a mass-produced graphics card shows speedups of up to factor 11 compared to LAMMPS on a Pentium IV 3.0 GHz. Our results are especially encouraging.

The presented implementation of the MD simulation algorithm using CUDA is quite generic. Therefore, it would be interesting to see if this algorithm or its extensions can be integrated into widely used MD-based tools such as Gromacs [9] and AutoDock [17].

References


