Computer Physics Communications 179 (2008) 634–641

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

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

ARTICLE INFO

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

ABSTRACT

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: liuweiguo@ntu.edu.sg (W. Liu), asbschmidt@ntu.edu.sg (B. Schmidt), asgerrit@ntu.edu.sg (G. Voss), askwmwittig@ntu.edu.sg (W. Müller-Wittig).

0010-4655/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.cpc.2008.05.008

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:

$$\begin{cases} F_i = m_i a_i, \\ F_i = -\nabla_{ri} V(r_1, \ldots, r_N). \end{cases} \tag{2.1}$$

In Eq. (2.1), the atomic system contains N atoms. m_i is the atom mass, $a_i = d^2 r_i/dt^2$ 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:

$$V(r_1, \ldots, r_N) = \sum_i \sum_j u_2(r_i, r_j) + \sum_i \sum_j \sum_k u_3(r_i, r_j, r_k) + \cdots. \tag{2.2}$$

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:

$$u(r) = 4\varepsilon\left[\left(\frac{\delta}{r}\right)^{12} - \left(\frac{\delta}{r}\right)^{6}\right], \tag{2.3}$$

where r is the distance between two interacting atoms, δ is the diameter and ε is the well depth. Both ε and δ 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 $r_{\text{list}} = r_c + skin$ 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.

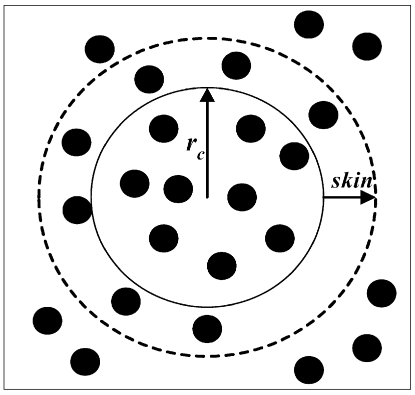


Fig. 1. Make use of r_c and *skin* to construct the neighbor list.

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 slower 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

Table 1
General specifications for NVIDIA CUDA-ready GPUs [1]

	Number of multiprocessors	Clock frequency (GHz)	Amount of device memory (MB)
GeForce 8800 GTX	16	1.35	768
GeForce 8800 GTX	12	1.2	640
Quadro FX 5600	16	1.35	1500
Quadro FX 4600	12	1.2	768

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], Belleman 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 presorting 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 CPU 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 Belleman both use CUDA to parallelize 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.

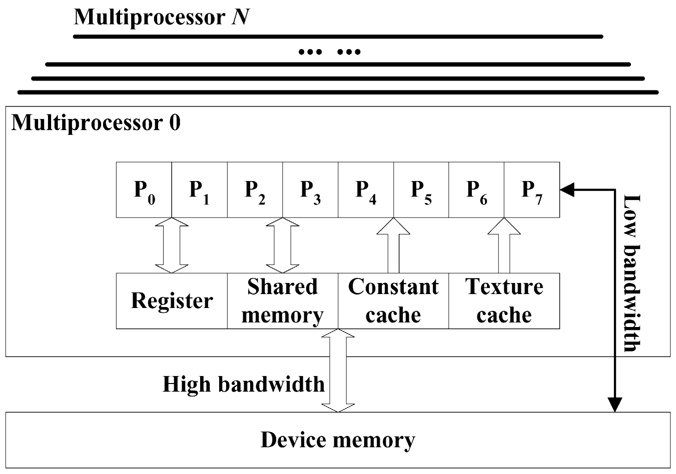


Fig. 2. The hardware model of CUDA-ready GPUs.

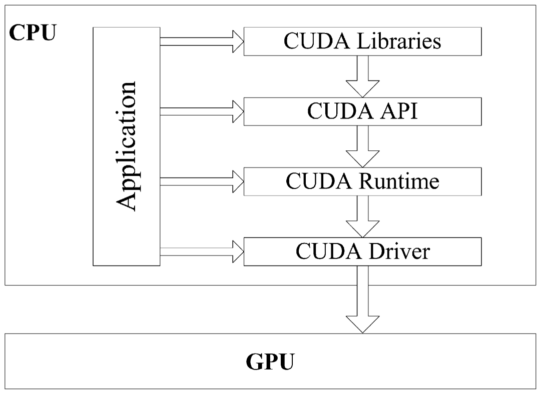


Fig. 3. The CUDA software stack architecture.

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.

```
(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**); $O(N)$
        Update the neighbor list, including all atom pairs that are within
        a distance range (**Neighbor list update**); $O(N^2)$
      End if;
(5)   Make use of the neighbor list to compute forces acted on all atoms
      (**Force calculation**); $O(N)$
(6)   Update velocities of all atoms (**Velocity update**); $O(N)$
(7)   Update the displace list, which contains the displacements of atoms
      (**Displace list update**); $O(N)$
(8)   Accumulate and output target statistics of each time step; $O(N)$
(9) End for
```

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

Table 2
Profiling of the six operations of a sequential MD simulation using a different number of particles on a single Pentium4 3 GHz. The time step is 100

Number of particles	Force calculation	Neighbor list update	Position update	Moved too much	Velocity update	Displace list 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%	52.0%	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%

(2) **Force-decomposition (FD).** In this method, a subset of the force loops inherent in Eq. (2.2) is assigned to each processor. It reduces the expensive communication and memory costs by a factor $\sqrt{P}$ compared with the AD method. However, FD cannot maintain load-balance as easily as AD.

(3) **Spatial-decomposition (SD).** This method corresponds to a geometric decomposition of the physical simulation domain. Each processor computes only the forces on atoms in its sub-domain. As the simulation progresses, processors exchange atoms when they move from one sub-domain to another. SD is more suitable than AD and FD for large-scale MD simulations on coarse-grained architectures, such as clusters [20].

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 particles using the code published in [11]. Our profiling has revealed that more than 97% of the overall runtime is spent on the neighbor list update and force calculation steps (see Table 2). Hence, we have decided to map these two steps onto a GPU.

5.1. Partitioning and kernels

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 is called its *head atom*. If an atom is within distance r_{list} (see Section 2) of the head atom the index this atom is added to the neighbor list of the corresponding head atom.

When designing an efficient parallel neighbor list update algorithm with CUDA, we have to consider that, there is no fast synchronization or communication mechanism between threads in

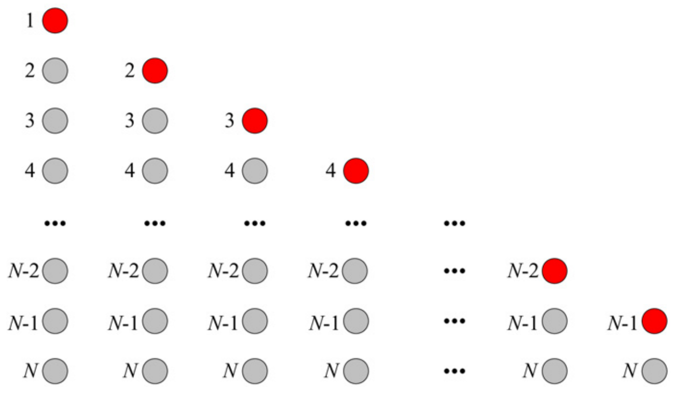


Fig. 5. Illustrations of neighbor list construction step. For the head atom in each column (the red circle), a loop over all other atoms (gray circles) will be done to calculate the pairwise distance. (For interpretation of the references to colour in this figure legend, the reader is referred to the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

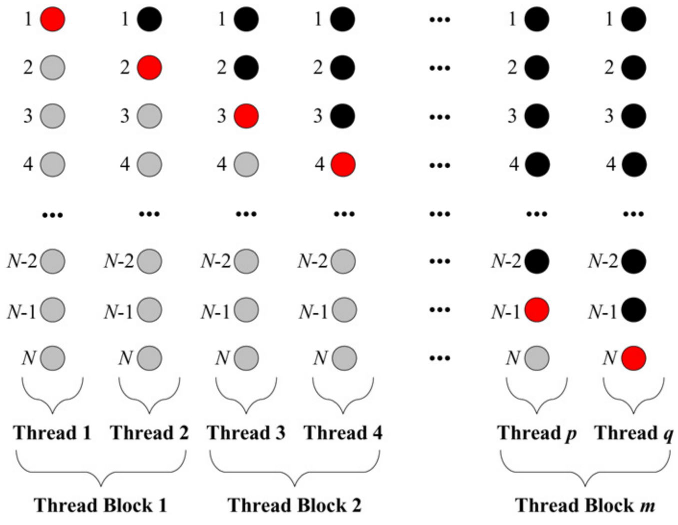


Fig. 6. Example of the partitioning of the distance matrix D for parallel neighbor list update with CUDA. The example uses one thread per column of D and each thread block consists of two threads.

```
(1) For all allocated head atoms do
(2)   Put the coordinates of current head atom into a register;
(3)   For all atoms exclude the current head atom do
(4)     Compute the pairwise distance between the current atom and head atom
          (full distance matrix computation);
(5)     Compare the pairwise distance with $r_{\text{list}}$ and put the indices of eligible
          atoms into the neighbor list in the device memory;
(6)   End for
(7)   Reset the displace list of current head atom with the value 0;
(8) End for
```

Fig. 7. CUDA-based neighbor list update kernel.

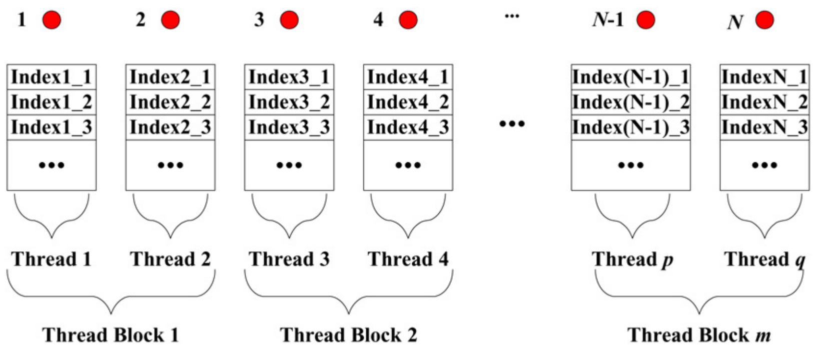


Fig. 8. The neighbor list corresponding to each atom is stored in the device memory and assigned to threads for use in the compute force step.

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).

```
(1)  For all allocated head atoms do
(2)    Put the coordinates of current head atom $i$ into a register;
(3)    Set the value of forces acting on atom $i$ as 0
         ($f_i = 0$, $f_i$ is put into a register);
(4)    For atoms in the current neighbor list do
(5)      Compute the distance $d_{ij}$ between the current atom $j$ and head atom $i$;
(6)      If $d_{ij} < r_c$ do
(7)        Calculate and accumulate the force $f_i$ acting on atom $i$;
(8)      End if
(9)    End for
(10)   Put the value of $f_i$ into on-chip shared memory;
(11) End for
```

Fig. 9. CUDA-based force calculation kernel.

Table 3
Performance comparison between kernel runtime and overhead. The time step is 100

Number of particles	Force calculation kernel (ms)	Force calculation overhead (ms)	Neighbor list update kernel (ms)	Neighbor list update overhead (ms)
8192	480	1115	96	123
16384	630	1199	412	119
32768	2600	3512	1479	146
65536	7950	6724	5946	179
131072	17386	13259	23638	284

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

```
Host program executed on CPU
(1)  Initialize atoms' status and the LJ potential table; set parameters controlling
       the simulation;
(2)  Load data into GPU device memory and launch the kernel (data uploading);
       Kernel program executed on GPU
       (4)  For all time steps do
       (5)    Position update;
       (6)    If Moved too much, do
                Neighbor list update;
              End if;
       (7)    Force calculation;
       (8)    Velocity update;
       (9)    Displace list update;
       (10) End for
(11) Read back results to CPU (data readback);
(12) Output results of each time step;
```

Fig. 10. Integrating all time steps into a single kernel program.

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.

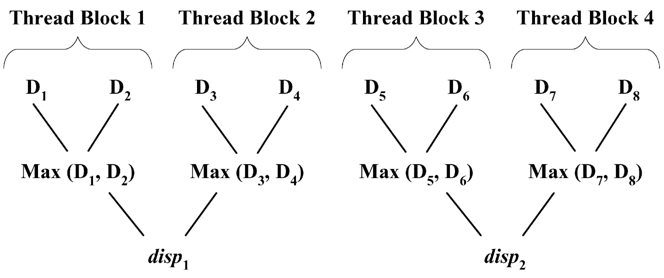


Fig. 12. Tree-based approach used within each thread block.

computation into multiple kernel invocations, thread block communication can be avoided. But this method can not 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 2.5δ to 4.5δ. 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-

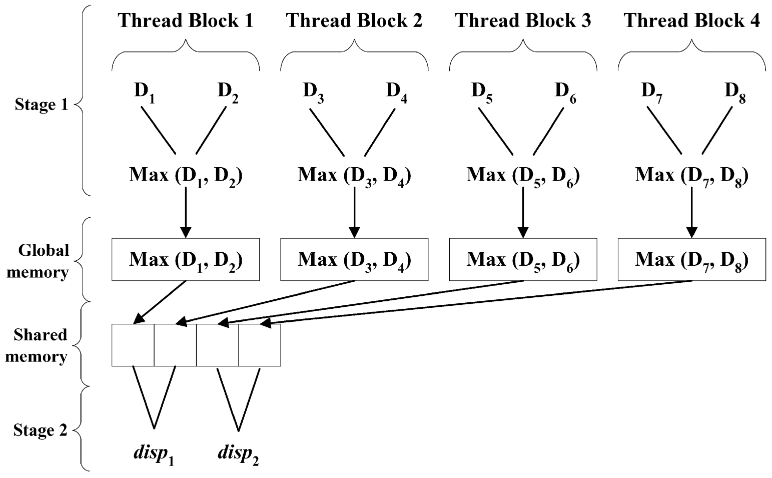


Fig. 13. Two-stage tree-based approach used in our implementation.

Table 4
Comparison of runtimes (in milliseconds) and speedups of MD simulation running on a single Pentium4 3 GHz (MD-CPU) to our GPU-accelerated version running on an AMD Opteron 2210 1.8 GHz with an NVIDIA GeForce 8800 GTX 512 (MD-GPU) for various number of particles and cutoff distances. The total runtime is broken down into F.C. (*force calculation*), N.L.U. (*neighbor list update*), and O.T. (*other time*) for MD-CPU and into K.R. (*kernel runtime*), N.L.U., K.O. (*kernel overhead*) and O.T. for MD-GPU. The time step is 100

#Particles	Cutoff distance	MD-CPU				MD-GPU				Speedup
		F.C.	N.L.U.	O.T.M	Total	K.R.	K.O.	O.T.	Total	
8192	2.5δ	5595	2141	217	7953	418	115	125	658	12.1
	3.0δ	8594	2140	172	10906	625	114	177	916	11.9
	3.5δ	12826	2141	299	15266	1052	117	172	1341	11.4
	4.0δ	17454	2125	233	19812	1510	110	133	1753	11.3
	4.5δ	23360	2156	265	25781	2002	119	141	2262	11.4
16384	2.5δ	12468	8594	375	21437	1022	131	245	1398	15.3
	3.0δ	19140	8594	375	28109	1461	133	281	1875	15.0
	3.5δ	29357	8609	455	38421	2061	123	284	2468	15.6
	4.0δ	40390	8609	422	49421	2667	122	289	3078	16.1
	4.5δ	54920	8625	486	64031	3992	121	277	4390	14.6
32768	2.5δ	29702	33047	751	63500	3044	145	531	3720	17.1
	3.0δ	47515	33078	875	81468	4068	145	649	4862	16.8
	3.5δ	69080	33078	873	103031	5596	143	558	6297	16.4
	4.0δ	93764	33078	1033	127875	7461	146	609	8216	15.6
	4.5δ	126902	33141	925	160968	9605	145	548	10298	15.6
65536	2.5δ	71119	132375	1568	205062	10090	191	1012	11293	18.2
	3.0δ	117533	132344	1638	251515	13185	193	1168	14546	17.3
	3.5δ	172949	132375	1676	307000	17530	194	1244	18968	16.2
	4.0δ	242958	132437	1715	377110	21973	195	1285	23453	16.1
	4.5δ	330299	132562	1623	464484	27839	192	1109	29140	15.9
131072	2.5δ	98263	544375	3424	646062	30373	283	2209	32865	19.7
	3.0δ	152568	543078	3369	699015	34908	403	2298	37609	18.6
	3.5δ	235911	543344	3370	782625	41020	286	2225	43531	18.0
	4.0δ	334189	544594	3342	882125	47346	285	2244	49875	17.7
	4.5δ	459567	544938	3370	1007875	56518	293	2485	59296	17.0

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 Pentium4 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.

Table 5
Comparison of runtimes (in milliseconds) and speedups of LAMMPS running on a Pentium4 3 GHz to our CUDA implementation running on an AMD Opteron 2210 1.8 GHz with an NVIDIA GeForce 8800 GTX 512 for various number of particles and cutoff distances. The time step is 100

#Particles	Cutoff distance	LAMMPS	MD-GPU	Speedup
8192	2.5δ	5120	658	7.8
	3.0δ	6664	916	7.3
	3.5δ	8971	1341	6.7
	4.0δ	11272	1753	6.4
	4.5δ	14205	2262	6.3
16384	2.5δ	12508	1398	8.9
	3.0δ	17573	1875	9.4
	3.5δ	24132	2468	9.8
	4.0δ	29428	3078	9.6
	4.5δ	38644	4390	8.8
32768	2.5δ	32529	3720	8.7
	3.0δ	47941	4862	9.9
	3.5δ	55383	6297	8.8
	4.0δ	65101	8216	7.9
	4.5δ	81242	10298	7.9
65536	2.5δ	96491	11293	8.5
	3.0δ	122192	14546	8.4
	3.5δ	150497	18968	7.9
	4.0δ	173146	23453	7.4
	4.5δ	214890	29140	7.4
131072	2.5δ	343931	32865	10.5
	3.0δ	411997	37609	11.0
	3.5δ	493818	43531	11.3
	4.0δ	538077	49875	10.8
	4.5δ	651413	59296	11.0

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 since GPU performance grows faster than Moore's law as it applies to CPUs.

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].

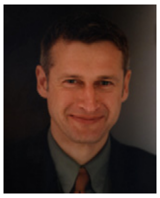
References

[1] NVIDIA CUDA Compute Unified Device Architecture-Programming Guide (V1.1), http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf, Nov. 2007.
[2] NVIDIA CUDA Homepage http://developer.nvidia.com/object/cuda.html.
[3] S. Alam, P. Agarwal, M. Smith, J. Vetter, D. Caliga, Using FPGA devices to accelerate biomolecular simulations, Computer 40 (3) (2007) 66–73.
[4] M.P. Allen, Introduction to molecular dynamics simulation, in: Computational Soft Matter—From Synthetic Polymers to Proteins, NIC Series, vol. 23, John von Neumann Institute for Computing, 2004, pp. 1–28, http://www.fzj.helmholtz.de/nic-series/volume23/allen.pdf.
[5] A.G. Anderson, W.A. Goddard, P. Schroder, Quantum Monte Carlo on graphical processing units, Computer Physics Communications 177 (2007) 298–306.
[6] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, Journal of Computational Physics 227 (2008) 5342–5359.
[7] R.G. Belleman, J. Bédorf, S.F. Portegies Zwart, High performance direct gravitational *N*-body simulations on graphics processing units II: An implementation in CUDA, New Astronomy 13 (2008) 103–112.
[8] K.J. Bowers, et al., Scalable Algorithms for molecular dynamics simulations on commodity clusters, in: Proc. ACM/IEEE Conf. on Supercomputing (SC06) Tampa, FL, 2006.
[9] H.J.C. Berendsen, D. Van Der Spoel, R. Van Drunen, GROMACS: A message-passing parallel molecular dynamics implementation, Computer Physics Communications 91 (1995) 43–56.
[10] S. Collange, M. Daumas, D. Defour, Line-by-line spectroscopy simulations on graphics processing units, Computer Physics Communications 178 (2008) 135–143.
[11] F. Ercolessi, A molecular dynamics primer, http://www.fisica.uniud.it/~ercolessi/md/, 1997.
[12] R. Fine, G. Dimmler, C. Levinthal, FASTRUN: A special purpose hardwired computer for molecular simulation, Proteins 11 (1991) 242–253.
[13] B.G. Fitch, et al., Blue matter: Strong scaling of molecular dynamics on blue Gene/L, in: Proc. International Conf. on Computational Science (ICCS 2006), in: LNCS, vol. 3992, Springer-Verlag, 2006, pp. 846–854.
[14] T. Hamada, T. Iitaka, The chamomile scheme: An optimized algorithm for *N*-body simulations on programmable graphics processing units. 2007, ArXiv Astrophysics e-prints, astro-ph/0703100, 2007.
[15] J.E. Lennard-Jones, Cohesion, in: Proceedings of Physical Society, 1931, pp. 461–482.
[16] W. Liu, B. Schmidt, G. Voss, W. Müller-Wittig, Streaming algorithms for biological sequence alignment on GPUs, IEEE Transactions on Parallel and Distributed Systems 18 (9) (2007) 1270–1281.
[17] G.M. Morris, D.S. Goodsell, R.S. Halliday, R. Huey, W.E. Hart, R.K. Belew, A.J. Olson, Automated docking using a Lamarckian genetic algorithm and empirical binding free energy function, J. Computational Chemistry 19 (1998) 1639–1662.
[18] T. Narumi, Y. Ohno, N. Okimoto, A. Suenaga, R. Yanai, M. Taiji, A high-speed special-purpose computer for molecular dynamics simulations: MDGRAPE-3, in: NIC Workshop 2006, NIC Series, vol. 34, John von Neumann Institute for Computing, 2006, pp. 29–36, http://www.fzj.helmholtz.de/nic-series/volume34/narumi.pdf.
[19] V.S. Pande, et al., Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing, Biopolymers 68 (1) (2003) 91–109.
[20] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, Journal of Computational Physics 117 (1995) 1–19.
[21] M.C. Schatz, C. Trapnell, A.L. Delcher, A. Varshney, High-throughput sequence alignment using graphics processing units, BMC Bioinformatics 8 (474) (2007).
[22] D.E. Shaw, et al., Anton: A special-purpose machine for molecular dynamics simulation, in: Proceedings of the 34th Annual International Symposium on Computer Architecture, San Diego, California, 2007.
[23] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, Journal of Computational Chemistry 28 (2007) 2618–2640.
[24] M. Taufer, C. An, A. Kerstens, C.L. Brooks III, Predictor@Home: A protein structure prediction supercomputer based on global computing, IEEE Transactions on Parallel and Distributed Systems 17 (8) (2006) 786–796.
[25] S. Toyoda, Development of MD engine: High-speed accelerator with parallel processor design for molecular dynamics simulations, Journal Computational. Chemistry 20 (1999) 185–199.
[26] J. Yang, Y. Wang, Y.C. Chen, GPU accelerated molecular dynamics simulation of thermal conductivities, Journal of Computational Physics 221 (2) (2007) 799–804.

Weiguo Liu received his Bachelor and Master degree from the Xi'an JiaoTong University, China in 1998 and 2002, and the Ph.D. degree from the Nanyang Technological University (NTU), Singapore, in 2006. He is currently a Research Fellow with the Center for Advanced Media Technology at NTU. His research interests include computational biology, parallel algorithms and architectures, high-performance computing, and data mining.

Bertil Schmidt is an Associate Professor at the School of Computer Engineering at NTU, Singapore. He received his Master degree in Computer Science from the Kiel University, Germany, in 1995 and his Ph.D. degree from the Loughborough University, UK, in 1999. He has worked with the company ISATEC in the area of embedded parallel systems. His research interests include parallel algorithms and architectures, high-performance computing, and bioinformatics.

Gerrit Voss is a senior staff member of the Centre for Advanced Media Technology, NTU Singapore since 2001. Prior to this, he worked in the Department. Visualization and Virtual Reality. at Fraunhofer-IGD Darmstadt, Germany. He is a member of the core development team of OpenSG (www.opensg.org). Mr. Voss received his diploma degree in Computer Science from the Darmstadt Technical University (Germany) in 1997. He is author of several publications in the area of real time rendering, Virtual Reality, Augmented Reality and GPGPU computation.



Wolfgang Müller-Wittig is the Director of the Centre for Advanced Media Technology (CAMTech) since January 2001. CAMTech is a joint venture between the Fraunhofer Institute for Computer Graphics (Fraunhofer-IGD), Darmstadt (Germany), and Nanyang Technological University (NTU), Singapore. Furthermore, he is an Associate Professor at the NTU School of Computer Engineering. Prior to joining CAMTech, he worked as a scientist in the "Visualization & Virtual Reality" Department at Fraunhofer-IGD, and then as the head of the "Visualization Group". Current research foci include highly interactive three-dimensional computer graphics for the manufacturing, engineering, edutainment, cultural heritage, and biomedical sciences. He received his university degree (Dipl.-Inform.) as well as his doctoral degree (Dr.-Ing.) in Computer Science from the Darmstadt University of Technology (Germany).