Molecular Dynamics Simulations on Commodity GPUs with CUDA

Weiguo Liu[1], Bertil Schmidt[2], Gerrit Voss[1], and Wolfgang Müller-Wittig[1]

[1] School of Computer Engineering, Nanyang Technological University
Centre for Advanced Media Technology
`{liuweiguo, asgerrit, askwmwittig}@ntu.edu.sg`
[2] Computer Science and Engineering, University of New South Wales
`bertil.schmidt@unsw.edu.au`

Abstract. Molecular dynamics simulations are a common and often repeated task in molecular biology. The need for speeding up this treatment comes from the requirement for large system simulations with many atoms and numerous time steps. In this paper we present a new approach to high performance molecular dynamics simulations on graphics processing units. Using modern graphics processing units for high performance computing is facilitated by their enhanced programmability and motivated by their attractive price/performance ratio and incredible growth in speed. To derive an efficient mapping onto this type of architecture, we have used the Compute Unified Device Architecture (CUDA) to design and implement a new parallel algorithm. This results in an implementation with significant runtime savings on an off-the-shelf computer graphics card.

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 scientific computing [1], image processing [2], computational biology [3,4], and fast Fourier transform [5], just to name a few. 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 [6]. Consequently, GPUs will become an even more attractive alternative for high performance computing in the near future.

However, there are still a number of challenges to be solved in order to enable scientists other than computer graphics specialists to facilitate efficient usage of these resources within their research area. The biggest challenge in order to solve a specific problem using GPUs is reformulating the proposed algorithms and data structures using computer graphics primitives (e.g. triangles, textures, vertices, fragments). Furthermore, restrictions of the underlying streaming architecture have to be taken into account, e.g. random access writes to memory is not supported and no cross fragment data or persistent state is possible (e.g. all the internal registers are flushed before a new fragment is processed).

The Compute Unified Device Architecture (CUDA) [7] 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. CUDA technology gives computationally intensive applications access to the tremendous processing power of GPUs through a revolutionary new programming interface. Providing orders of magnitude more performance and simplifying software development by using the standard C language, CUDA enables developers to create innovative solutions for data-intensive problems.

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 new algorithms based on the CUDA programming model. These algorithms have been implemented using C++ and CUDA and tested on a physical system of 16,384 atoms. We show that our new MD algorithms lead to a significant performance improvement on an NVIDIA GeForce 8800 GTX card.

The rest of this paper is organized as follows. In Section 2, we introduce the basic MD simulation algorithms and highlight previous work on parallelization of these algorithms on different parallel architectures. Important features of the CUDA programming model are described in Section 3. Section 4 presents our new CUDA-based MD algorithms and their efficient GPU implementation. A performance evaluation is given in Section 5. Finally, Section 6 concludes the paper with an outlook to further research topics.

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 [8]. 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 is a deterministic technique. That is, given an initial set of parameters, the subsequent time evolution is in principle completely determined [9]. 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:

$$\begin{cases} F_i = m_i a_i \\ F_i = - \bigtriangledown_{r_i} V(r_1, \ldots, r_N) \end{cases} \tag{1}$$

In Eq.(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:

$$V(r_1, \ldots, r_N) = \sum_i \sum_j u_2(r_i, r_j) + \sum_i \sum_j \sum_k u_3(r_i, r_j, r_k) + \cdots \tag{2}$$

In Eq.(2), the three body (and higher order) interactions are usually neglected [10], only leaving the pair potential as the concentration of the simulation. In practice, the Lennard-Jones (LJ) potential [11] is the most commonly used interaction model. It is given by the following expression:

$$u(r) = 4\epsilon \left[\left(\frac{\delta}{r} \right)^{12} - \left(\frac{\delta}{r} \right)^{6} \right] \tag{3}$$

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 takes more than 90% of the total simulation time [12]. From Eq.(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 [10]. 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.

Figure 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 $r_{list} = r_c + skin$ are stored. At the first step in a MD simulation, the neighbor list is constructed for all the neighbors of each atom. From time to time the list is reconstructed.

Because of their inherent parallelism [13], MD simulations are suitable candidates for mapping onto parallel architectures. In the past twenty years, re-

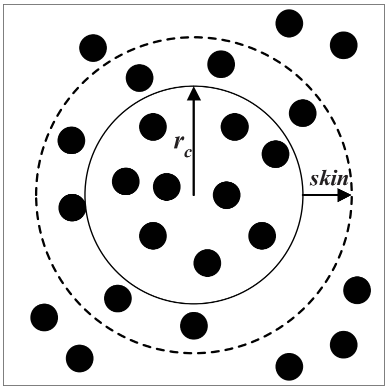

Fig. 1. Make use of r_c and *skin* to construct the neighbor list

searchers have exploited MD's parallelism on various parallel machines. In addition to architectures specifically designed for MD simulations, existing programmable sequential and parallel architectures have been used for solving them.

Special-purpose architectures can provide the fastest means of running a particular algorithm with very high processing element (PE) density. Each PE is specifically designed for the pariwise 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. GRAPE [14] is a series of application specific processor designs, which is specially built to accelerate the MD simulations. More recent examples, better tuned to the needs of MD simulations, include ATOMS [15], FASTRUN [16], and MDGRAPE-3 [17].

Considerable effort has been spent by researchers to implement MD simulation algorithms on vector supercomputers [18]. Several other approaches are based on SIMD or MIMD parallel machines with a few dozens of processors [19,20]. SIMD and MIMD architectures are programmable and can be used for a wider range of applications. Since these architectures contain more general-purpose parallel processors, their PE density is less than the density of special-purpose architectures. Nevertheless, these solutions can still achieve significant runtime savings. However, the costs involved in designing and producing SIMD architectures are quite high. As a consequence, none of the above solutions has a successor generation, making upgrading impossible.

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 [7] offer a simplified programming environment.

3 CUDA Programming Model for Computing on GPUs

Compute Unified Device Architecture (CUDA) is a novel hardware and 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 [21]. For now, it is available for NVIDIA 8800 series, NVIDIA Quadro FX 5600/4600, and beyond.

From the 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.

Table 1. General specifications for NVIDIA CUDA-ready GPUs [21]

	Number of Multiprocessors	**Clock frequency (GHz)**	**Amount of device memory (MB)**
GeForce 8800 GTX	16	1.35	768
GeForce 8600 GTX	12	1.2	640
Quadro FX 5600	16	1.35	1500
Quadro FX 4600	12	1.2	768

A multiprocessor has on-chip memory of four types:

(1) one 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 access to the non-cached device memory. 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 hide memory latency, and can also better fill the instruction pipeline so there are no idle processors. According to [21], 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.

4 CUDA-Based MD Simulation Algorithms

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 characters 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 [13]. The AD method has been widely used especially on shared memory architectures.
(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 very well suited to large-scale MD simulations. It achieves optimal $O(N/P)$ scaling and achieves better performance on Coarse-grained architectures, such as Clusters, than AD and FD [13].

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 is: (1) good load balancing and scalability can be easily achieved, (2) according to the CUDA model 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 outline in Figure 2 illustrates how a sequential MD simulation works. In Figure 2, the computational complexity of each operation is listed on the end of them. In practice, the neighbor list update and force computation are the most time-consuming operations in each time step.

In the neighbor list update step (step (4) in Figure 2), a list is constructed for all neighbors of each atom. There are a large number of pairwise calculations in this step: each atom will loop over all other atoms to compute the pairwise distance between them. This corresponds to compute an $N \times N$ distance matrix D. As $r_{ij} == r_{ji}$, only the lower triangle matrix has to be computed, thus the calculation is half reduced. If the pairwise distance with the head atom of current column is within r_{list} (see Section 2), the index of current atom is added into the neighbor list array of current head atom.

There are two problems we should consider when design our CUDA-based neighbor list update algorithm. First, as mentioned in Section 2, in CUDA, thread blocks cannot communicate or synchronize with each other. This

```
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; O(N)
4.      If there are atoms who have moved too much, do
            Update the neighbor list, including all atom pairs that are within a
            distance range (half distance matrix computation); $O(N^2)$
        End if
5.      Make use of the neighbor list to compute forces acted on all atoms; O(N)
6.      Update velocities of all atoms; O(N)
7.      Update the displace list, which contains the displacements of atoms; O(N)
8.      Accumulate and output target statistics of each time step; O(N)
9. End for
```

Fig. 2. The outline of a sequential MD simulation (with the computation complexity listed, N is the number of atoms)

limitation will make the full computation of the distance matrix D necessary. For example, assume each column of the distance matrix D is assigned to a single thread and there are two threads in a thread block (see Figure 3).

In Figure 3, if we only calculate the lower triangle matrix then except for Thread 1, all other threads cannot keep the whole information of local neighbor list. For instance, as to Thread 4, the current head atom 4 will not know whether atoms 1, 2 and 3 are in the local neighbor list or not. In order get this information, Thread 4 must access the local neighbor lists of atoms 1, 2 and 3. In CUDA, this access is very expensive because it has to be done in the low bandwidth device memory. In order to solve this problem, we let each thread loop over all other atoms for current head atom. That is, in Figure 3 both the lower triangle and upper triangle matrices are calculated. Figure 4 shows our algorithm for neighbor list update using CUDA. Because the coordinates of head atoms will be reused many times in the inner loop over all other atoms in order to calculate pairwise distances, we put them into a register before the inner loop so as to speedup access for them.

After the neighbor list update step, the indices of all eligible atoms will be stored in the neighbor list array in the device memory for later usage. This is mainly because the size of the neighbor list may be very large and there is no enough on-chip memory to store it. During the compute force step, each thread will loop over the local neighbor lists to do force calculations.

Figure 5 gives our CUDA-based algorithm for the force computation. Because the coordinates of head atoms and the forces acted on them are reused many times in the inner loop over all atoms in the neighbor list, we put them into registers before the inner loop so as to increase the access efficiency for them. The results of force computation f_i will be used by other operations, such as the position and velocity update operations (step (3) and (6) in Figure 2), so we put them into a dynamically allocated shared memory to speedup access to them.

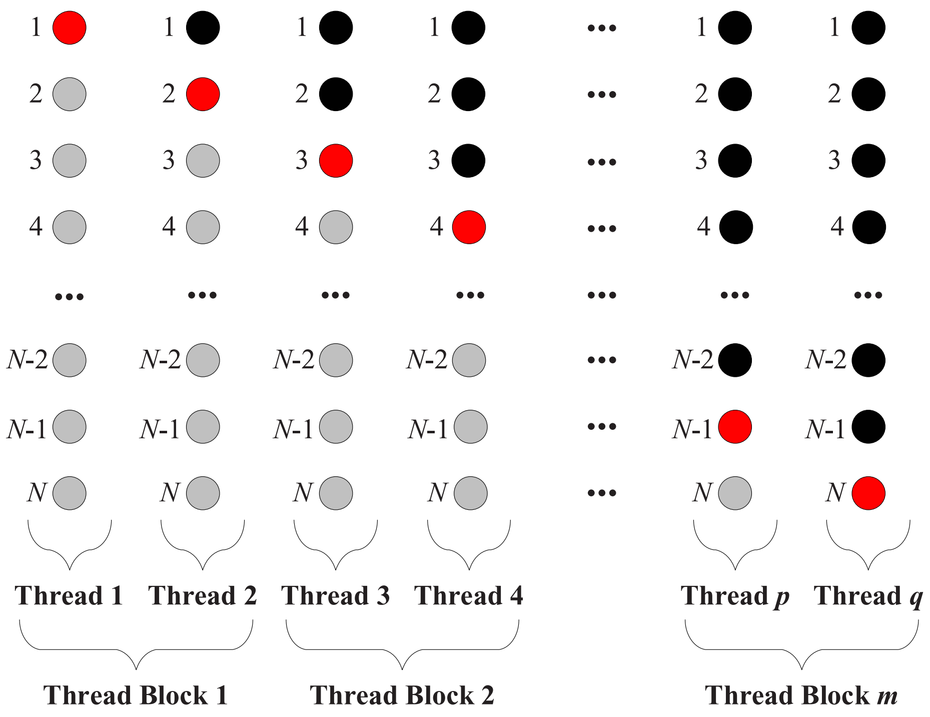

Fig. 3. Parallel neighbor list update illustration (Red circles denote head atoms). Assume each thread is allocated one column of the distance matrix and each thread block consists of two threads.

```
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 $r_{list}$ 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. 4. CUDA-based neighbor list update algorithm

Figure 6 shows our CUDA-based MD simulation algorithm. In order to eliminate the overhead for launch multiple kernels, we have put all time step loops into a single kernel. As the kernel cannot output results directly, all statistics have to be read back to CPU for further processing and outputting.

```
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 acted 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 $f_i$ acted 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. 5. CUDA-based force computation algorithm

```
/*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;
        /*Kernel program executed on GPU*/
        3.  For all time steps do
        4.      Update positions of all atoms;
        5.      If there are atoms who have moved too much, do
                    CUDA-based neighbor list update algorithm;
                End if
        6.      CUDA-based force computation algorithm;
        7.      Update velocities of all atoms;
        8.      Update the displace list;
        9.      Put statistics of each time step into the device memory;
       10.  End for
11. Read back statistics to CPU;
12. For all time steps do
13.     Output statistics of each time step;
14. End for
```

Fig. 6. CUDA-based MD simulation algorithm

5 Performance Evaluation

We have implemented the proposed algorithm using CUDA Toolkit 0.8 [7] and evaluated it on the following graphics card:

- *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 1.8GHz, 2GB RAM running Windows XP.

Table 2. Comparison of runtimes (in milliseconds) and speedups of MD simulation running on a single Pentium4 3GHz to our GPU-accelerated version running on an AMD Opteron 2210 1.8GHz with an NVIDIA GeForce 8800 GTX 512 for various time steps. The cutoff distance is fixed at 2.5δ.

Time steps		**100**	**200**	**300**	**400**	**500**
Indices in the neighbor list		1096077	1096077	1096077	1096077	1096077
MD-CPU	**Overall(ms)**	22468	36063	49703	63406	78078
MD-GPU (8800GTX)	**Kernel(ms)**	1168	1914	2656	3399	4149
	Overall(ms)	1468	2265	3078	3875	4671
Speedup	**Overall**	15.3	15.9	16.1	16.4	16.7

A set of performance evaluation tests have been conducted using different numbers of time steps and cutoff distances, to evaluate the processing time of the GPU implementation versus that of the original MD simulation on the PC. The MD simulation program is benchmarked on an Intel Pentium IV 3GHz processor with 1GB RAM. We have modified the MD code (md3.f90) from Ercolessi ([9], available online at http://www.fisica.uniud.it/ ercolessi/md/f90/) into a 32bit version for our evaluation. This is because for now, there is only a 32bit version of CUDA. In our experiments, there are 16,384 atoms in the simulated physical system.

Table 3. Comparison of runtimes (in milliseconds) and speedups of MD simulation running on a single Pentium4 3GHz to our GPU-accelerated version running on an AMD Opteron 2210 1.8GHz with an NVIDIA GeForce 8800 GTX 512 for various cutoff distances. The time step is fixed at 100.

r_c		$\mathbf{2.5\delta}$	$\mathbf{3.0\delta}$	$\mathbf{3.5\delta}$	$\mathbf{4.0\delta}$	$\mathbf{4.5\delta}$
Indices in the neighbor list		1096077	1597512	2549064	3243673	4607456
MD-CPU	**Overall(ms)**	22468	29984	41234	53984	69765
MD-GPU (8800GTX)	**Kernel(ms)**	1168	1634	2309	3111	4446
	Overall(ms)	1468	1968	2641	3437	4796
Speedup	**Overall**	15.3	15.2	15.6	15.7	14.5

Table 2 reports the performance of the sequential MD and our CUDA implementation for different time steps. In Table 2, we set the cutoff distance r_c 2.5δ and skin 0.5δ. Table 3 shows the performance of the sequential MD and our CUDA implementation for different cutoff distances. In Table 3, we make both programs run 100 time steps while $skin = 0.5\delta$. From Table 2 and Table 3 we can see, our GPU implementation achieves speedups of almost seventeen compared to the sequential MD simulation runtime.

6 Conclusions and Future Work

In this paper we have introduced CUDA-based MD simulation algorithms that can be efficiently implemented on modern graphics hardware. We have made use of the fast on-chip memory in CUDA to design and implement our algorithms. All key components of our algorithms have been mapped onto the GPU for execution. The evaluation of our implementation on a high-end graphics card shows a speedup of almost seventeen compared to a Pentium IV 3.0GHz. The results are especially encouraging and to our knowledge this is the first reported implementation of MD simulations on graphics hardware using CUDA.

Our implementation of the MD simulation algorithm using CUDA is quite generic. Our future work will include the extension and integration of this implementation into Gromacs [22] and Autodock [23].

Acknowledgment

This work was supported in part by the A*Star BMRC grant 04/1/22/19/375. Weiguo Liu is currently supported by the Lee Kuan Yew Postdoctoral Fellowship.

References

1. Krüger, J., Westermann, R.: Linear algebra operators for gpu implementation of numerical algorithms. ACM Trans. Graph. 22, 908–916 (2003)
2. Xu, F., Müller, K.: Ultra-fast 3d filtered backprojection on commodity graphics hardware. In: IEEE International Symposium on Biomedical Imaging 2004 (2004)
3. Liu, W., Schmidt, B., Voss, G., Müller-Wittig, W.: Streaming algorithms for biological sequence alignment on gpus. IEEE Transactions on Parallel and Distributed Systems 18(9), 1270–1281 (2007)
4. Horn, D., Houston, M., Hanrahan, P.: Clawhmmer: a streaming hmmer-search implementation. In: Proceedings of Supercomputing 2005 (2005)
5. Moreland, K., Angel, E.: The fft on a gpu. In: Proceedings SIGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 112–119 (2003)
6. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A., Purcell, T.: A survey of general-purpose computation on graphics hardware. In: Eurographics 2005, pp. 21–51 (2005)
7. Nvidia: NVIDIA CUDA Homepage, `http://developer.nvidia.com/object/cuda.html`
8. Sheng, H., Guerrieri, R., Sangiovanni-Vincentelli, A.L.: Three-dimensional monte carlo device simulation for massively parallel architectures. Technical Report UCB/ERL M95/53, EECS Department, University of California, Berkeley (1995)
9. Ercolessi, F.: A molecular dynamics primer. Technical report (1997) `http://www.fisica.uniud.it/~ercolessi/md/`

10. Allen, M.: Introduction to molecular dynamics simulation. Computational Soft Matter-From Synthetic Polymers to Proteins 23, 1–28 (2004)
11. Lennard-Jones, J.: Cohesion. In: Proceedings of Physical Society, pp. 461–482 (1931)
12. Shu, J., Wang, B., Zheng, W.: Cluster-based parallel simulation for large scale molecular dynamics in microscale thermophysics. In: Cao, J., Yang, L.T., Guo, M., Lau, F. (eds.) ISPA 2004. LNCS, vol. 3358, pp. 200–211. Springer, Heidelberg (2004)
13. Plimpton, S.: Fast parallel algorithms for short-range molecular dynamics. Journal of Computational Physics 117, 1–19 (1995)
14. Hut, P., Makino, J.: Computational physics – astrophysics on the grape family of special-purpose computers. Science 283, 501–505 (1999)
15. Bakker, A., Gilmer, G., Grabow, M., Thompson, K.: A special purpose computer for molecular dynamics calculations. Journal of Computational Physics 90, 313–335 (1990)
16. Fine, R., Dimmler, G., Levinthal, C.: Fastrun: A special purpose, hardwired computer for molecular simulation. Proteins 11, 242–253 (1991)
17. Narumi, T., Ohno, Y., Okimoto, N., Suenaga, A., Yanai, R., Taiji, M.: A high-speed special-purpose computer for molecular dynamics simulations: Mdgrape-3. In: NIC Workshop 2006, vol. 34, pp. 29–36 (2006)
18. Mink, A., Bailly, C.: Parallel implementation of a molecular dynamics simulation program. In: Simulation Conference Proceedings, vol. 1, pp. 13–16 (1998)
19. Nakano, P., Kalia, R.: Parallel multiple-time-step molecular dynamics with three-body interaction. Computer Physics Communications 77, 303–312 (1993)
20. Tamayo, P., Mesirov, J., Boghosian, B.: Parallel approaches to short-range molecular dynamics simulations. Supercomputing, 462–470 (1991)
21. Nvidia: NVIDIA CUDA Compute Unified Device Architecture-Programming Guide, (2007) `http://developer.download.nvidia.com/compute/cuda`
22. Berendsen, H., Van Der Spoel, D., Van Drunen, R.: Gromacs: A message-passing parallel molecular dynamics implementation. Computer Physics Communications 91, 43–56 (1995)
23. Morris, G., Goodsell, D., Huey, R., Olson, A.: Distributed automated docking of flexible ligands to proteins: Parallel applications of autodock 2.4. Journal of Computer-Aided Molecular Design 10(2), 293–304 (1996)