Journal of Computational Physics 229 (2010) 5486–5497

Contents lists available at ScienceDirect

Journal of Computational Physics

journal homepage: www.elsevier.com/locate/jcp

GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows

Alfeus Sunarso [*], Tomohiro Tsuji, Shigeomi Chono

Department of Mechanical Engineering, Kochi University of Technology, Kami-shi, Kochi 782-8502, Japan

ARTICLE INFO

Article history:
Received 20 October 2009
Received in revised form 24 February 2010
Accepted 30 March 2010
Available online 13 April 2010

Keywords:
Molecular dynamics
Large scale simulation
Graphics Processing Unit (GPU)
Macroscopic flow
Anisotropic molecule
Liquid crystals

ABSTRACT

We have developed a GPU-based molecular dynamics simulation for the study of flows of fluids with anisotropic molecules such as liquid crystals. An application of the simulation to the study of macroscopic flow (backflow) generation by molecular reorientation in a nematic liquid crystal under the application of an electric field is presented. The computations of intermolecular force and torque are parallelized on the GPU using the cell-list method, and an efficient algorithm to update the cell lists was proposed. Some important issues in the implementation of computations that involve a large number of arithmetic operations and data on the GPU that has limited high-speed memory resources are addressed extensively. Despite the relatively low GPU occupancy in the calculation of intermolecular force and torque, the computation on a recent GPU is about 50 times faster than that on a single core of a recent CPU, thus simulations involving a large number of molecules using a personal computer are possible. The GPU-based simulation should allow an extensive investigation of the molecular-level mechanisms underlying various macroscopic flow phenomena in fluids with anisotropic molecules.

1. Introduction

Detailed studies on macroscopic phenomena at the molecular-level are attracting attention from various scientific communities. For this type of study, molecular dynamics simulations have become an important tool. However, performing molecular dynamics simulations to investigate the mechanisms of macroscopic phenomena is challenging as the simulations involve a large number of molecules, and thus have a large computational load. For simulations that involve a large number of molecules, single CPU computation systems are not sufficient, and therefore, systems with more computational power such as supercomputers and cluster systems are required. Because of their high investment and operational costs, supercomputers and cluster systems are only available in large institutions.

Recently, Graphics Processing Units (GPUs) have provided an alternative means of accelerating scientific computations. The use of GPUs for scientific computations was made possible by the advance of programming toolkits such as the GL Shading Language (GLSL) [1], C for Graphics (Cg) [2] and the NVIDIA Compute Unified Device Architecture (CUDA) [3]. The much better performance/cost ratio of GPUs compared with CPU-based computing systems such as supercomputers and cluster systems has motivated the development of various GPU-based scientific applications [4]. Furthermore, since computational power has reached the teraflop level, GPUs enable large-scale simulations to be performed on a personal computer.

In the field of molecular dynamics simulations, there have been some studies in which the capability of GPUs for accelerating the molecular dynamics simulations has been demonstrated. In an early study, a GPU was used to accelerate the

* Corresponding author. Tel./fax: +81 887572150.
E-mail address: sunarso@kochi-tech.ac.jp (A. Sunarso).

0021-9991/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jcp.2010.03.047

molecular dynamics simulation of thermal conductivity [5]. The use of a GPU to accelerate Coulomb summation and the non-bonded force calculation, which are usually encountered in the molecular modeling of large biomolecules, has been reported by Stone et al. [6]. Full GPU implementations of molecular dynamics simulations of Lennard–Jones particle systems have been reported by Van Meel et al. [7] and Anderson et al. [8]. The simulation of Van Meel et al. was based on a cell-list structure, while the simulation of Anderson et al. was based on a neighbor-list structure. Most recently, a complete all-atom protein molecular dynamics simulation fully implemented on a GPU was reported by Friedrichs et al. [9]. Despite the diversity of problems and simulation algorithms used in previous studies, it was demonstrated that computations on GPU are faster than those on a single CPU core by one to two orders of magnitude.

In the present work, we present a GPU implementation of molecular dynamics simulations for the study of fluids with anisotropic molecules such as liquid crystals. This is the first report on the GPU implementation of molecular dynamics simulations that deal with nonspherical particle (ellipsoid) systems undergoing macroscopic flows, while the previous works [5–9] deal with simulations of spherical particle systems in equilibrium. It should be noted that GPUs have a large number of processors but a limited amount of high-speed memory (register and shared memory) resources. A large amount of global memory is available, but intensive access to the global memory would introduce a performance penalty due to its high latency. As the computations of intermolecular interactions for ellipsoidal systems are not only computationally intensive but also data intensive, the applicability of GPU for this kind of computation needs further investigation. The present work is aimed to address this issue by presenting the suitable computation techniques and their detailed implementations on the GPU. Even though the simulation methods used in the present work are similar to that used by Van Meel et al. [7], the detailed implementations on the GPU require a specific consideration on the memory accesses management to obtain an optimal performance. Some modifications in the GPU implementation details are required in order to fit the memory requirement to the memory availability. Another important issue is that simulations involving macroscopic flows require the data structures such as cell-lists and neighbor-lists to be updated at every time step. The main drawback of method proposed in Ref. [7] is that cell-lists is not precisely up to date because the cell-lists is not updated after the calculation of position but after the calculation of force. In addition, implementation of the method to the present work is problematic due to the limited amount of shared memory. To overcome this problem, we propose a new algorithm to update the cell-lists by using a cell index. The proposed algorithm is very efficient such that the cell-lists can be updated at every time step without significant additional cost. It is also important to note that due to the limitation in high-speed memory resources, high performance simulations on GPU are only achieved for single-precision simulations. The use of single-precision computations raises a question about the reliability of the simulation results. Using a numerical example we demonstrate that the GPU-based simulations are still reliable for simulation of systems that involve macroscopic flows.

The presentation in this paper focuses on GPU implementation of molecular dynamics simulations for the study of liquid crystalline flows. General computation algorithms and their specific implementations on a GPU, as well as some problems related to the optimization of computations, are discussed in detail. Even though the presentation focuses on simulation of liquid crystalline flows, the simulation methods and GPU specific implementations presented here should be useful for simulations of other fluids that involve anisotropic particles, such as simulations of amphiphilic self-assembly (surfactant) system [10] as well as simulations of giant biomolecules [11,12]. Furthermore, the use of ellipsoid particle in the modeling of polymer nanocomposite suspensions has been reported [13]. The proposed algorithm for updating the cell-lists should be useful not only for simulations of anisotropic particle systems but also for molecular dynamics simulations in general.

To demonstrate the applicability of the developed simulation, the dynamics of a nematic liquid crystal in a parallel plate cell under the application of an electric field is simulated, and the calculation of bulk flow fields from the molecular states is presented. A molecular-level study of liquid crystalline flows should be interesting from both scientific and engineering viewpoints, as liquid crystals exhibit a strong coupling between the macroscopic flow field and molecular orientation [14]. The flow field changes the molecular orientation, and conversely, the change in molecular orientation generates a flow field known as backflow [15]. Recently, there has been an increasing interest in backflow owing to its potential applications to microactuators [16,17] and micromanipulators [18]. For the development of microactuators and micromanipulators, the understanding of the molecular-level mechanism of backflow is essential. A study using a 2D molecular dynamics simulation [19] showed that the rotation and rearrangement of molecules under the application of an electric field introduces a local bulk velocity gradient, which plays an important role in the generation of macroscopic flow. The computed velocity profile in the 2D simulation is qualitatively in agreement with that observed in visualization experiment [20]. However, for more precise investigations of the effects of molecular properties such as molecular polarity, molecular shape and initial molecular arrangement, a full 3D molecular dynamics simulation is required. The GPU-based simulation presented here should enable the full 3D simulations to be performed on a personal desktop computer.

2. Numerical methods

In this section, numerical methods for the simulation of backflow in a nematic liquid crystal confined between parallel plates under the application of an electric field is presented. However, as noted previously, the simulation techniques presented here should be useful for simulations of other fluids that involve anisotropic particles. The developed simulation can also be used for simulations of other flow problems such as shear flow by modifying the boundary conditions.

2.1. Governing equations

For the simulation of liquid crystalline flows, we consider the computational domain and molecular model as shown in Fig. 1. The simulations involve a large number of molecules, and thus have a large computational load. Therefore, simulations with atomistic details are prohibitively expensive. To reduce the computational load the liquid crystal molecules are modeled as ellipsoids. The intermolecular interaction of the ellipsoids is governed by the Gay–Berne potential [21].

$$\phi_{ij}^{\mathrm{GB}} = 4\epsilon(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \hat{\boldsymbol{r}}_{ij})\left\{\left[\frac{\sigma_0}{r_{ij} - \sigma(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \hat{\boldsymbol{r}}_{ij}) + \sigma_0}\right]^{12} - \left[\frac{\sigma_0}{r_{ij} - \sigma(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \hat{\boldsymbol{r}}_{ij}) + \sigma_0}\right]^{6}\right\}. \tag{1}$$

Here, $\hat{\boldsymbol{u}}_i$ and $\hat{\boldsymbol{u}}_j$ are unit vectors representing the orientations of molecules i and j, respectively, and $\hat{\boldsymbol{r}}_{ij}$ is the unit vector of vector $\boldsymbol{r}_{ij}(\hat{\boldsymbol{r}}_{ij} = \boldsymbol{r}_{ij}/r_{ij})$ that connects the centers of mass of molecules i and j. $\sigma(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \hat{\boldsymbol{r}}_{ij})$ represents the contribution of the molecular orientation to the intermolecular distance, while $\epsilon(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \hat{\boldsymbol{r}}_{ij})$ defines the potential well. The parameters of the potential (related to $\sigma(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \hat{\boldsymbol{r}}_{ij})$ and $\epsilon(\hat{\boldsymbol{u}}_i, \hat{\boldsymbol{u}}_j, \hat{\boldsymbol{r}}_{ij})$) are molecular length scale σ_0, energy scale ϵ_0, molecular aspect ratio σ_r, energy ratio ϵ_r and the constants for the potential well μ and ν.

Most of liquid crystal molecules are polar, thus they have a permanent dipole $\boldsymbol{p}^p$. Under the application of an electric field $\boldsymbol{E} = E\hat{\boldsymbol{e}}$, in addition to the permanent dipole $\boldsymbol{p}^p$, an induced dipole $\boldsymbol{p}^i$ also exists. The magnitude and direction of the induced dipole depend on the strength and direction of the electric field, $\boldsymbol{p}^i = \alpha E(\hat{\boldsymbol{e}} \cdot \hat{\boldsymbol{u}})\hat{\boldsymbol{u}}$. Here, α is the polarizability, and E and $\hat{\boldsymbol{e}}$ are the strength and unit vector of the electric field, respectively. The contribution of the molecular dipole $\boldsymbol{p}_i = \boldsymbol{p}_i^p + \boldsymbol{p}_i^i$ to the molecular interaction is computed using the dipole–dipole interaction potential [22].

$$\phi_{ij}^{\mathrm{DP}} = \frac{1}{r_{ij}^3}\{\boldsymbol{p}_i \cdot \boldsymbol{p}_j - 3(\boldsymbol{p}_i \cdot \hat{\boldsymbol{r}}_{ij})(\boldsymbol{p}_j \cdot \hat{\boldsymbol{r}}_{ij})\}. \tag{2}$$

The interaction between the electric field and the molecular dipoles induces a torque on each molecule of $\boldsymbol{T}_i^E = \boldsymbol{p}_i \times \boldsymbol{E}$. Therefore, the force $\boldsymbol{F}$ and torque $\boldsymbol{T}$ can be computed as follows.

$$\boldsymbol{F}_i = -\sum_{j=0, j\neq i}^{N}\left(\frac{\partial \phi_{ij}}{\partial \boldsymbol{r}_{ij}}\right), \tag{3}$$

$$\boldsymbol{T}_i = -\sum_{j=0, j\neq i}^{N}\left(\hat{\boldsymbol{u}}_i \times \frac{\partial \phi_{ij}}{\partial \hat{\boldsymbol{u}}_i}\right) + \boldsymbol{T}_i^E. \tag{4}$$

Here, $\phi_{ij} = \phi_{ij}^{\mathrm{GB}} + \phi_{ij}^{\mathrm{DP}}$ is the total intermolecular potential. After the force and torque have been computed for all molecules, the motion of the molecules can be computed by integrating the following equations:

$$m_i \frac{d^2 \boldsymbol{r}_i}{dt^2} = m_i \boldsymbol{a}_i = \boldsymbol{F}_i, \tag{5}$$

$$\boldsymbol{I}_i \frac{d^2 \boldsymbol{\theta}_i}{dt^2} = \boldsymbol{I}_i \boldsymbol{\alpha}_i = \boldsymbol{T}_i, \tag{6}$$

where $\boldsymbol{r}$ and $\boldsymbol{\theta}$ represent the position and direction (orientation angle), and $\boldsymbol{a}$ and $\boldsymbol{\alpha}$ represent the translational and rotational acceleration, respectively. m and $\boldsymbol{I}$ are the mass and moment of inertia tensor of a molecule, respectively, and N is the number of molecules.

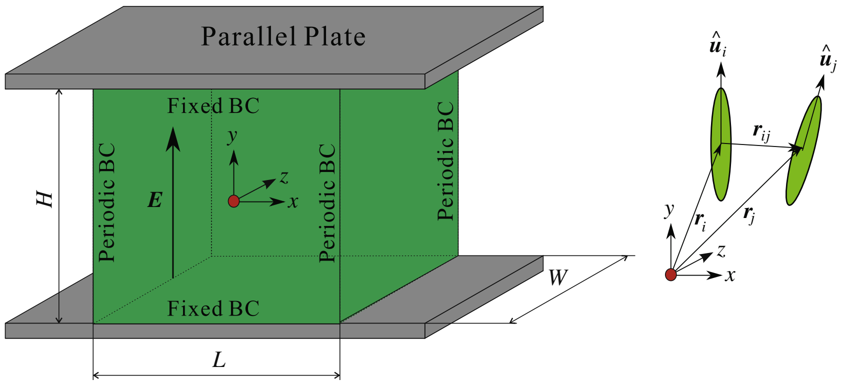

Fig. 1. Computational domain and molecular model.

2.2. Integration scheme

To compute the position and direction of molecules, Eqs. (5) and (6) are integrated using a leapfrog scheme [23]. The implementation of the leapfrog scheme for translational motion (Eq. (5)) is simple and straightforward. However, for rotational motion, the integration of Eq. (6) in terms of rotation angle θ would lead to a singular condition when θ approaches 0° or 180°, and therefore, a special treatment is required. In this work, we use the DLM (Dullweber, Leimkuhler and McLachlan) method [24], in which the rotation matrix is propagated instead of the rotation angle. Details of the rotation matrix computation can be found in [23].

In the integration of the equations of motion, we must compute the force and torque acting on all molecules (Eqs. (3) and (4)). Note that ϕ^{GB} is a short-range interaction, while ϕ^{DP} is a long-range interaction. For the treatment of the long-range interaction, we used the reaction field method, similar to that used in [22].

The computation of the intermolecular interaction is the most expensive part of the molecular dynamics simulation. In a naive approach, in which each molecule interacts with all other molecules, the computational load of the intermolecular interaction scales quadratically with the number of molecules N. For systems with large number of molecules, an N-squared algorithm is not suitable, and thus the cell-list or neighbor-list algorithm is used instead. In the cell-list method, the simulation box is decomposed into smaller domains called cells. The side length of each cell should be greater than or equal to the cutoff radius, which represents the maximum range of the intermolecular interaction. A molecule interacts only with other molecules in its own cell and the neighbor cells. In the neighbor-list method, a list of neighbor molecules with a distance smaller than the cutoff radius is constructed for each molecule. Anderson et al. [8] demonstrated that a highly optimized force calculation using cell-list method is faster than that using cell-list method reported in [7]. However, the time spent on generating the neighbor lists is much larger than that spent on force computation. Updating the neighbor-lists at every time step is time-consuming, and thus this method is not suitable for our simulation. As the arithmetic and memory operations in updating cell lists are proportional to number of molecules N, while the operations in updating neighbor-lists using a naive algorithm are proportional to N^2, updating the cell lists should be less time-consuming than updating the neighbor-lists. Therefore, in the present work, we implemented our simulation using the cell-list method and developed an efficient method for updating the cell lists at every time step.

In the simulation, we consider the constant-energy (NVE) ensemble as well as the constant-temperature (NVT) ensemble. For the NVT ensemble, the temperature is controlled using the Gaussian thermostat [23]. Here, the translational and rotational accelerations are corrected using the translational and rotational velocities ($\boldsymbol{v}$ and $\boldsymbol{\omega}$) as follows:

$$\boldsymbol{a} = m_i^{-1}\boldsymbol{F}_i - \zeta\boldsymbol{v}_i; \quad \boldsymbol{\alpha} = \boldsymbol{I}_i^{-1}\boldsymbol{T}_i - \zeta\boldsymbol{\omega}_i, \tag{7}$$

$$\zeta = \frac{\sum_i^N(\boldsymbol{v}_i\cdot\boldsymbol{F}_i) + \sum_i^N(\boldsymbol{\omega}_i\cdot\boldsymbol{T}_i)}{m_i\sum_i^N\boldsymbol{v}_i^2 + \sum_i^N(\boldsymbol{I}_i\boldsymbol{\omega}_i)^2}. \tag{8}$$

The details of the computation at each time step are shown in Table 1.

Table 1
Computation algorithm.

Step1 (*computation of position and direction*): Using the current states of molecules (at time t), calculate the new position $\boldsymbol{r}$ and direction $\hat{\boldsymbol{u}}$ at time $t + \Delta t$.
For translational motion:
$\boldsymbol{v}(t+\frac{\Delta t}{2}) = \boldsymbol{v}(t) + \frac{\Delta t}{2}\boldsymbol{a}(t)$
$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \Delta t\boldsymbol{v}(t+\frac{\Delta t}{2})$
Using $\boldsymbol{r}(t + \Delta t)$, update the cell index that maps each molecule to a cell.
For rotational motion:
$\boldsymbol{\omega}(t+\frac{\Delta t}{2}) = \boldsymbol{\omega}(t) + \frac{\Delta t}{2}\boldsymbol{\alpha}(t)$
$\boldsymbol{U}_1 = \boldsymbol{R}_x(\frac{\Delta t}{2}\omega_x)$
$\boldsymbol{U}_2 = \boldsymbol{R}_y(\frac{\Delta t}{2}\omega_y)$
$\boldsymbol{U}_3 = \boldsymbol{R}_z(\Delta t\omega_z)$
$\boldsymbol{R}^T(t+\Delta t) = \boldsymbol{U}_1\boldsymbol{U}_2\boldsymbol{U}_3\boldsymbol{U}_2\boldsymbol{U}_1\boldsymbol{R}^T(t)$
$\hat{\boldsymbol{u}}(t+\Delta t) = \boldsymbol{R}(t+\Delta t)\hat{\boldsymbol{u}}_{\text{ref}}$
Note: $\boldsymbol{R}_x$, $\boldsymbol{R}_y$ and $\boldsymbol{R}_z$ represent the rotation along the x, y and z axes respectively.

Step2 (*construction of cell lists*): Using the cell index, construct the cell lists.

Step3 (*computation of force and torque*): Using the new position $\boldsymbol{r}(t + \Delta t)$ and direction $\hat{\boldsymbol{u}}(t+\Delta t)$, calculate the force $\boldsymbol{F}(t + \Delta t)$ and torque $\boldsymbol{T}(t + \Delta t)$ (see Eqs. (3) and (4)).

Step4 (*computation of acceleration*): Using the new force $\boldsymbol{F}(t + \Delta t)$, torque $\boldsymbol{T}(t + \Delta t)$ and rotation matrix $\boldsymbol{R}(t + \Delta t)$, calculate the accelerations $\boldsymbol{a}$ and $\boldsymbol{\alpha}$.
$\boldsymbol{a}(t+\Delta t) = \frac{1}{m}\boldsymbol{F}(t+\Delta t)$
$\boldsymbol{\alpha}(t+\Delta t) = \boldsymbol{R}(t+\Delta t)\boldsymbol{I}_p^{-1}\boldsymbol{R}^T(t+\Delta t)\boldsymbol{T}(t+\Delta t)$

Step5 (*application of Gaussian thermostat*): Using temporary velocities $\boldsymbol{v}'(t+\Delta t) = \boldsymbol{v}(t+\frac{\Delta t}{2}) + \frac{\Delta t}{2}\boldsymbol{a}(t+\Delta t)$ and $\boldsymbol{\omega}'(t+\Delta t) = \boldsymbol{\omega}(t+\frac{\Delta t}{2}) + \frac{\Delta t}{2}\boldsymbol{\alpha}(t+\Delta t)$, update the accelerations $\boldsymbol{a}(t + \Delta t)$ and $\boldsymbol{\alpha}(t + \Delta t)$ (see Eqs. (7) and (8)).

Step6 (*computation of velocity*): Using $\boldsymbol{a}(t + \Delta t)$ and $\boldsymbol{\alpha}(t + \Delta t)$, calculate the velocity $\boldsymbol{v}$ and $\boldsymbol{\omega}$.
$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t+\frac{\Delta t}{2}) + \frac{\Delta t}{2}\boldsymbol{a}(t+\Delta t)$
$\boldsymbol{\omega}(t+\Delta t) = \boldsymbol{\omega}(t+\frac{\Delta t}{2}) + \frac{\Delta t}{2}\boldsymbol{\alpha}(t+\Delta t)$

2.3. Boundary conditions

For the simulation of backflow in a parallel plate cell, we impose periodic boundary conditions in the x and z directions (see Fig. 1). To model the anchoring effect of the wall, layers of molecules with fixed position and orientation are introduced at the upper and lower boundaries. To account for the restricting effect of the wall (the molecules are not allowed to escape from the upper and lower boundaries), a wall potential is introduced. The wall potential is similar to the Lennard–Jones potential, but only depends on smallest distance in the y direction of a molecule from the upper and lower boundaries.

3. Implementation on GPU

3.1. General parallelization method

To accelerate the molecular dynamics simulation using the GPU, we implemented our simulation using the CUDA programming environment [3]. As a background to assist understanding of the discussion on the parallelization method, an overview of the GPU architecture is given here.

A GPU consists of hundreds of processors that work in parallel. For example, an NVIDIA GTX280 card has 30 multiprocessors, with 8 processors on each multiprocessor, and thus there are a total of 240 processors on the card. The card has a 1 GB DDR3 global memory, a 65536B constant memory and a texture memory that can all be accessed by all processors. In addition, each multiprocessor has a 16384B high-speed shared memory and 16,384 registers (32-bit) that can be used only by the processors in the multiprocessor.

The GPU multiprocessors have a "Single Instruction Multiple Data" (SIMD) architecture, which is suitable for parallel computations. As shown in Fig. 2, in performing a parallel computation, the computation tasks are organized in blocks of threads. Each block of threads is handled by a multiprocessor, and each thread in a block is handled by a processor in the multiprocessor. Therefore, all threads can share the data in the global, constant and texture memories, but only threads in the same block can share the data in the shared memory. The data in the register are private to each thread and cannot be accessed by other threads.

To parallelize the calculations in the molecular dynamics simulations, the calculations of position, velocity and force of each molecule are assigned to the GPU threads. One thread handles the calculations for one molecule. This can be realized by mapping the global index of threads to the index of molecules. In the CUDA programming environment, the global index of a thread can be calculated from the size and index of the block (stored in `blockDim` and `blockIdx` structures) that the thread belongs to and from the relative index of the thread in the block (stored in the `threadIdx` structure). For a 1D block, for example, the global index can be calculated as follows: `gidx = blockIdx.x*blockDim.x+threadIdx.x`.

In the present work, most of the calculations were performed on the GPU, except for the generation of cell lists and the summation of properties such as kinetic and potential energies owing to their poor parallelization property. To perform the calculations on the GPU, the data of molecules (position, direction, rotation matrix, translational and rotational velocities and accelerations, and molecular dipole) and the data of cell lists were initialized on the CPU and then copied to global memory of the GPU. Constant data such as the size of the simulation box, the cell structure and the potential parameters were copied to the constant memory of the GPU.

For the calculation of position (see Step1 in Table 1 and Fig. 2), for example, N_T threads are launched on the GPU ($N_T = N_B \times N_{TB} \geqslant N$; N_T, N_B, N_{TB} and N are the total number of threads, the number of blocks, the number of threads per block and the number of molecules, respectively). Each thread loads the position, velocity and acceleration data of each molecule from the global memory into the register, and then computes the new position and writes the results back into the global memory.

In accessing the global memory, there is a delay from the time a memory address is requested to the time it is available, which is equal to the time required for hundreds of arithmetic operations. The memory latency can be hidden by the execution interleaved warps (32 threads per warp) on the GPU, as the warps whose data have been read can perform arithmetic operations while other warps (on the same multiprocessor) are waiting for their data. However, if the data are shared by threads, better performance can be obtained by using the high-speed shared memory instead of the global memory.

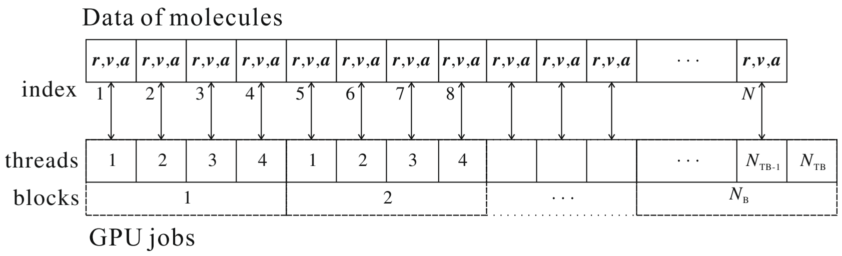

Fig. 2. General method for parallelization of computations in molecular dynamics simulation.

3.2. Optimization of force calculation

In the computations of position, direction, velocity, acceleration, electric torque and wall force, the calculations for a molecule only requires the data of the molecule itself, and therefore, the computations can be easily parallelized using the method described previously. In contrast, in the calculation of intermolecular force and torque on molecule i (see Eqs. (1)–(4)), the data of position, direction and dipole moment of each neighbor molecule j are required. Therefore, special treatments are required to compute the intermolecular interactions efficiently.

The computational load in the computation of intermolecular interactions can be reduced by reducing the number of interactions per molecule. This can be realized by using the cell-list method. In this method, only the neighbor molecules in the same cell and in the neighbor cells need to be considered. For calculation on a CPU, the number of interactions can be further reduced by applying Newton's third law (the force on molecule j resulting from the interaction of molecules i and j equals the negative of the force on molecule i). However, the implementation of Newton's third law on the GPU will result in simultaneous writes by multiple threads on the same memory address, which cannot be handled efficiently by the GPU. The simultaneous writes can be avoided by accumulating subsets of force in shared memory and writing them out to global memory buffers [9]. For simulation of systems with long-range interaction that typically use the N-squared algorithm, this method is acceptable because the cost of accessing the data of force in the global memory is covered by the speedup gained from the reduction of the N^2 operations in computation of intermolecular interactions. However, as noted in Ref. [8], for simulation of systems with short-range interactions, in which the interactions are usually computed using cell-list or neighbor-list method, the cost for accessing global memory would be larger than time saving from the reduction of $N \times N_i$ (N_i is number of interaction per molecules which depends on number density and cutoff radius) operations in computation of intermolecular interactions. Furthermore, as the computation of force and torque in the present work requires a large amount of shared memory, there are no sufficient space for the accumulation of subsets of force and torque in the shared memory. Therefore, Newton's third law is not implemented on the GPU.

The parallelization of intermolecular force and torque computations uses the basic concept described in Fig. 2. One thread is devoted to the calculations of force and torque of a molecule. In a simple algorithm, as illustrated in Fig. 3, each thread loads the data of the reference molecule into the register, then loads the data of the counterpart molecule and computes the interaction force and torque as it loops over all other molecules in the same cell and neighbor cells. In this simple approach, the memory access is not efficient because many threads read the same data from the global memory. Memory access can be optimized by using the high-speed shared memory to reduce the redundancy of access to the relatively slow global memory.

To take advantage of the shared memory, the computation is performed on a cell basis; a block of threads is assigned to a cell and the computation of force and torque for a molecule in the cell is assigned to a thread in the block. As depicted in

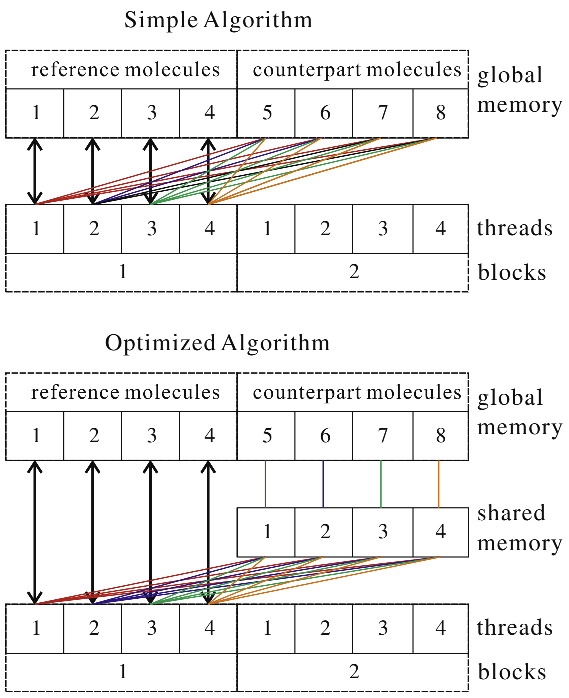

Fig. 3. Memory access in the calculation of intermolecular interaction using a simple algorithm (upper) and an optimized algorithm (lower). The access to the data of counterpart molecules by different threads are indicated by different colors. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Fig. 3, each thread in the block loads the data of the reference molecule from the global memory into the registers, and then each thread loads the data of one counterpart molecule from the global memory into the shared memory. After all threads have finished loading the data, each thread computes the interaction force and torque from the data in the registers and shared memory and accumulates the force for the reference molecule. The loading of data of counterpart molecules and the calculation of intermolecular interactions are first performed on the reference cell and then performed on all neighbor cells. By loading the data of counterpart molecules into the shared memory, the data can be shared by threads in the same block, and thus the ratio of arithmetic operations to memory operations is increased. This should lead to an improvement in performance as compared with the simple approach, in which each thread loads the data of its counterpart molecules directly from the global memory each time the data are required. In the implementation of the computation algorithm for the problem in the present work, there are some important issues that should be addressed carefully in order to obtain an optimal performance.

One of important factors that determine the performance of simulation on a GPU is GPU occupancy, which indicates the ratio of actually active threads to the maximum number of threads that can be handled by a multiprocessor unit. The number of active threads is limited by the required number of shared memory per block and the required number of registers per thread. For simulations of spherical particle systems [5–9], there should be no problem with GPU occupancy because the required shared memory and registers to store the data of particle positions are relatively small. However, the GPU occupancy is severely restricted by the large amount of shared memory and registers required for the calculation of force and torque from the Gay–Berne potential and dipole–dipole potential in the present work. In addition, there is a large computational load due to a large amount of arithmetic operations involved. The computational load can be reduced by pre-calculating repeatedly-used terms and storing them into temporary variables in registers, but consequently the number of active threads would be limited by the number of required registers per thread. Therefore, there are two possible approaches to optimize the performance that need to be investigated. In the first approach, the GPU occupancy is maximized by reducing the required shared memory and registers and put some data in the global memory, while in the second approach, the use of shared memory and registers is maximized while GPU occupancy is kept to a relatively low level. For illustration, we consider the case in which number of threads per block N_{TB} is set to 128. If all data of counterpart molecules (position, direction, dipole and type) are stored in shared memory the GPU occupancy is limited to 25%, but each thread can use up to 64 registers, which enables the repeatedly-used terms to be stored in the registers. The GPU occupancy can be increased to 38% by storing the data of type of molecule in the global memory instead of shared memory and reducing the amount of the required registers to 40 registers per thread, and therefore, some terms should be computed repeatedly. It was found that the two alternatives are almost equal in performance; the occupancy-maximized approach is only faster about 10% than the memory-maximized approach.

For the computation of force and torque, the number of blocks is set to the number of cells, and the number of threads per block is set to a value larger than or equal to the maximum number of molecules in the cells N_{MC} to ensure that one thread handles not more than one molecule and each molecule has a handling thread. It is possible for a thread to loop over more than one molecules. However, doing this would lead to a performance drop due to severe warp divergence, and thus it should be avoided. As the execution of threads in a multiprocessor is managed in warps of 32 threads, the number of threads per block N_{TB} should be set to a multiple of 32, $N_{TB} = 32 \times \text{ceil}\,(N_{MC}/32)$. This results in a large number of redundant threads (more than $N_B \times (N_{TB} - N_{MC})$), which should lead to a decrease in performance as the difference between the number of threads per block and the number of molecules in a cell increases. In the simple method, the number of blocks is computed from a predefined number of threads per block, $N_B = \text{ceil}\,(N/N_{TB})$, and thus the thread redundancy is less than N_{TB} ($N_{TB} \leqslant 512$), which is not significant compared with the large number of molecules ($N \geqslant 100{,}000$). The trade-off between memory optimization and thread redundancy will be addressed in the performance tests.

3.3. Cell lists with cell index

With the presence of macroscopic flow, the molecules would cross the cell boundaries intensively. Therefore, the cell-lists should be updated after the calculation of positions at every time step. Updating the cell lists using method proposed in [7] results in cell lists that are not precisely up to date. It is possible to overcome this drawback by setting the side length of the cell to a value larger than maximum interaction range plus skin thickness λ, where λ is the maximum displacement of the molecule per time step. However, the implementation of this scheme in our simulation would lead to a large skin that would impose a performance penalty due the computation of a large number of unnecessary interactions. Furthermore, the method requires the buffering of cell lists in shared memory that would further reduce the GPU occupancy. Therefore, a new scheme to updated the cell lists is required.

To overcome the problems noted above, we propose a scheme to update cell lists by using cell index, as depicted in Fig. 4. It should be noted that the cell lists contain the number of molecules and the list of molecules for each cell. The data of cell boundaries, the number of neighbor cells and the list of neighbor cells for each cell are stored in cell structure that stored in the constant memory. The cell index is stored in the global memory and contains the index of the corresponding cell for each molecule. Using the cell index, the mapping of each molecule to the cell where it belongs to can be performed in parallel immediately after its position has been computed at every time step. The cell index is updated by comparing new position with the boundaries of the corresponding cell and the neighbor cells. When constructing the cell lists, cell index is reloaded instead of molecular positions, and thus memory operations can be reduced.

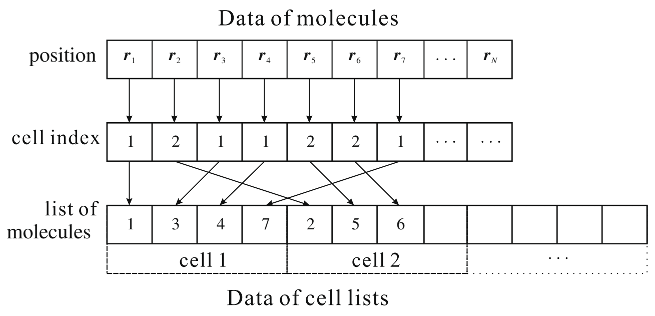

Fig. 4. Generation of cell lists using cell index.

The simple way for constructing the cell lists from the cell index is illustrated in Fig. 4. After the number of molecules in each cell is reset to zero, the corresponding cell for each molecule is evaluated from the cell index, and then the index of the molecule is added to the list of molecules of the corresponding cell. However, the implementation of this operation on the GPU leads to simultaneous writes on the same memory address by multiple threads. To avoid the simultaneous writes, updating list for a cell should be assigned to a threads block, similar to that for computing intermolecular force. Loading the cell index into the shared memory can be done in parallel by all threads in the block, but evaluation of cell index and adding data into the list are performed by only one thread while other threads stay idle. Because updating the cell lists only involves memory operations (no arithmetic operations) and is serial in nature, it should be more efficient to perform it on the CPU instead of on the GPU. To do this, the cell index is copied to the CPU and the cell lists are constructed using the cell index, and then the new cell lists are copied to the global memory of the GPU. It is found that for simulations with low number density and small cutoff radius, updating cell-lists using the full GPU approach is slightly faster than that using the GPU-CPU approach, while for simulations with low number density but relatively large cutoff radius such as for simulations of liquid crystalline flow in the present work, updating cell-lists using the GPU-CPU approach is about two times faster than that using the full GPU approach.

4. Results and discussion

In this section, we compare the performance of GPU-based simulations with the CPU-based simulation, and then present the application of the simulations to the study of backflow. For the simulation of nematic liquid crystals over a relatively wide range of temperature, the parameters of the Gay–Berne potential are set as follows: $\sigma_r = 5, \epsilon_r = 1$, $\nu = 1$ and $\mu = 2$. The number density is $\rho^* = 0.188$ and the cutoff radius is $r_{\mathrm{cut}} = 7$.

4.1. Performance tests

To investigate the performance of the GPU-accelerated simulation, we run our simulation on various computation systems as shown in Table 2. On the Windows system, the program was compiled using the CUDA toolkit and Microsoft Visual Studio 2008 compiler, while on the Linux (openSUSE 11.1) systems, the program was compiled using the CUDA toolkit and GNU compiler. Optimization level 2 (`-O2`) was used, and the option `-use_fast_math` was used to take advantage of the fast function unit of the GPU.

The integration of equations of the motions, as shown by the algorithm in Table 1, are implemented in various subroutines. Note that in the calculations of force and torque (Step3), the calculations of electric torque and wall force were separated from the calculation of intermolecular force. The computation times of the various subroutines for the simulation of 1,000,000 molecules ($N_x = 100, N_y = 1000, N_z = 10$) using the GPU are shown in Table 3. It is clear that the calculation of intermolecular force and torque is the most expensive part, comprising more than 90% of the computation time. The large computational time is due to the large number of arithmetic operations involved. Furthermore, the large number of shared memories and registers required in the calculations of interactions from the Gay–Berne and dipole potentials restricts the GPU occupancy to a low level (25%), such that optimal use of the GPU can not be achieved. It is possible to increase the GPU occupancy, as noted previously, but the memory access becomes less efficient, and thus performance improvement

Table 2
Specifications of computation systems used in performance tests.

CPU1	CPU: Intel Core 2 Extreme QX6850 3.00 GHz, Memory: DDR2 8 GB, OS: Windows XP 64 bit.
CPU2	CPU: Intel Core i7 940 2.93 GHz, Memory: DDR3 6 GB, OS: openSUSE 11.1 64 bit.
CPU3	CPU: Intel Xeon E5450 3.00 GHz, Memory: DDR2 16 GB, OS: openSUSE 11.1 64 bit.
GPU	NVIDIA GTX280, installed on the CPU2 system

Table 3
Computation time per time step for various subroutines.

Computation routine	Time [ms]	Percentage
Step3 (computation of intermolecular force and torque)	959.464	93.80
Step5 (application of Gaussian thermostat)	20.000	1.96
Step1 (computation of position and direction)	16.751	1.64
Step2 (generation of cell lists)	7.394	0.72
Step6 (computation of velocity)	6.950	0.68
Step4 (computation of acceleration)	5.816	0.57
Step3 (computation of electric torque)	4.492	0.44
Step3 (computation of wall force)	2.032	0.20

is not significant. As shown later, despite the use of GPU is not optimal, the GPU-based simulations are significantly faster than the CPU-based simulations.

The generation of cell lists using the approach described in the previous section is very fast. The total time required for the generation of cell lists for the system of 1,000,000 molecules is only 7.394 ms. This includes the times required for copying the cell index from the GPU to the CPU (1.003 ms), the generation of cell lists on the CPU (5.058 ms), and copying the cell lists from the CPU to the GPU (1.333 ms). Updating the cell index after the calculation of position and direction (Step1) only increases the computation time of Step1 less than 2% (about 0.25 ms). Using this method, the cell lists can be updated at every time step without any significant additional cost, and thus the method is suitable for simulations that involves macroscopic flow.

The total computation times per time step for different system sizes simulated on various computation systems are plotted in Fig. 5. Note that even though the CPUs are multicore, only a single CPU core was used in the simulations. Furthermore, as noted previously, Newton's third law was implemented for the simulation on the CPU but not for the simulation on the GPU. In all computation system, the computation time increases almost linearly with the system size N. Despite the variation of speedup for the different CPU types and system sizes, the simulation on the GPU is about 50 times faster than those on a single CPU core. The variation in speedup for different system sizes is due to the thread redundancy, which depends on the arrangement of molecules in the simulation box.

As noted previously, the optimization of the intermolecular force and torque calculation by using shared memory introduces a large thread redundancy. To investigate the trade-off between memory optimization and thread redundancy, the computation times of simulations using simple and optimized algorithms are shown in Table 4. In Case 1, the numbers of molecules in the x and y directions are fixed ($N_x = N_y = 100$), while the number of molecules in z direction varies ($N_z = 10 - 400$). In Case 2, the numbers of molecules in the x and z directions are fixed ($N_x = 100, N_z = 10$), while the number of molecules in y direction varies ($N_y = 100 - 4000$). The arrangement in Case 1 results in a relatively severe thread redundancy (N_T/N) compared with that in Case 2. Despite the large thread redundancy, the simulations using the optimized force calculation are faster than those using the simple force calculation. For a relatively low thread redundancy, speedup of about 40% can be obtained, while for a relatively high thread redundancy, the speedup decreases to about 10%.

Because the computation of intermolecular interactions and the specification of GPU in the present work are different from those in Ref. [7], direct performance comparison can not be carried out. However, it can be noted that the 50 folds speedup obtained in the present work is comparable to that obtained in Ref. [7], in which the speedup of 40 folds was reported. Regarding to the specification of the GPU in the present work, one should expect that the speedup of our simulations is about two times higher than that in Ref. [7], but considering the low GPU occupancy (25%) due to the limited high-speed memory resources, the speedup of our simulation would be a quarter of that in Ref. [7]. Therefore the 50 folds speedup is reasonable high for the problem considered in the present work.

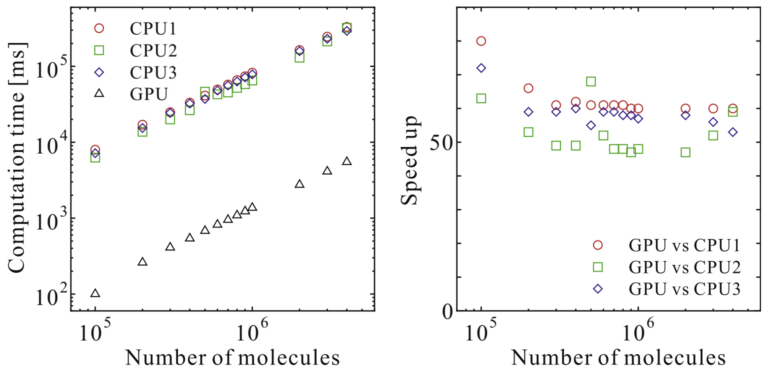

Fig. 5. Performance of simulations on CPUs and GPU for various system sizes.

Table 4
Trade-off between memory optimization and thread redundancy in the performance of simulation on GPU.

N	Case 1 ($N_z\uparrow$)				Case 2 ($N_y\uparrow$)			
	t_S	t_O	t_S/t_O	N_T/N	t_S	t_O	t_S/t_O	N_T/N
100,000	142	**100**	1.42	1.47	140	100	1.40	1.42
200,000	365	**260**	1.40	1.47	260	210	1.24	1.74
300,000	485	**410**	1.18	1.72	400	300	1.33	1.62
400,000	655	**540**	1.21	1.66	520	390	1.33	1.68
500,000	810	**680**	1.19	1.77	650	510	1.27	1.59
600,000	970	**820**	1.18	1.72	780	590	1.32	1.64
700,000	1080	**950**	1.14	1.79	920	680	1.35	1.59
800,000	1265	**1090**	1.16	1.75	1020	780	1.31	1.62
900,000	1370	**1230**	1.11	1.80	1180	870	1.36	1.57
1,000,000	1620	**1370**	1.18	1.77	1320	1030	1.28	1.57
2,000,000	3065	**2750**	1.11	1.81	2690	2080	1.29	1.47
3,000,000	4560	**4120**	1.11	1.82	4000	2950	1.36	1.55
4,000,000	6070	**5490**	1.11	1.82	5330	3900	1.37	1.56

N: total number of molecules ($N = N_xN_yN_z$, Case 1: $N_x = 100$, $N_y = 100$, $N_z = 10 - 400$, Case 2: $N_x = 100$, $N_y = 100 - 4000$; $N_z = 10$); t_S: computation time for simple algorithm (in ms); t_O: computation time for optimized algorithm (in ms); t_S/t_O: speedup resulting from using optimized algorithm; N_T/N: ratio of number of threads to number of molecules for optimized algorithm. The values in **bold** (column 3) correspond to the values plotted in Fig. 5.

4.2. Simulation of backflow

To demonstrate the application of the simulation to the study of backflow, the dynamics of molecules under the application of an electric field were simulated, and the bulk properties were computed by averaging the molecular properties. The considered ensemble consists of 100,000 ($N_x = N_y = 100, N_z = 10$) molecules with number density $\rho^* = 0.188$. The dimensions of the corresponding simulation box (see Fig. 1) are $L = H = 175\sigma_0$ and $W = 17\sigma_0$. The simulation was performed at a constant-temperature of $T^* = k_BT/\epsilon_0 = 2$ and a constant electric field strength of $\lambda^* = pE/\epsilon_0 = 2$.

Snapshot of molecules and the bulk velocity and molecular orientation after the application of an electric field for $t^* = 130$, as well as close-up views of molecules at various times, are presented in Fig. 6. In the snapshot and close-up views of the molecules, the color represents the x-position of the molecules at equilibrium ($t^* = 0$). In the plot of bulk properties, the bulk velocity is shown by blue arrows, while the bulk molecular orientation is shown by green lines. Here, the bulk properties were computed by dividing the computational domain into 10 subdomains in the y direction and averaging the properties of the molecules in the subdomains.

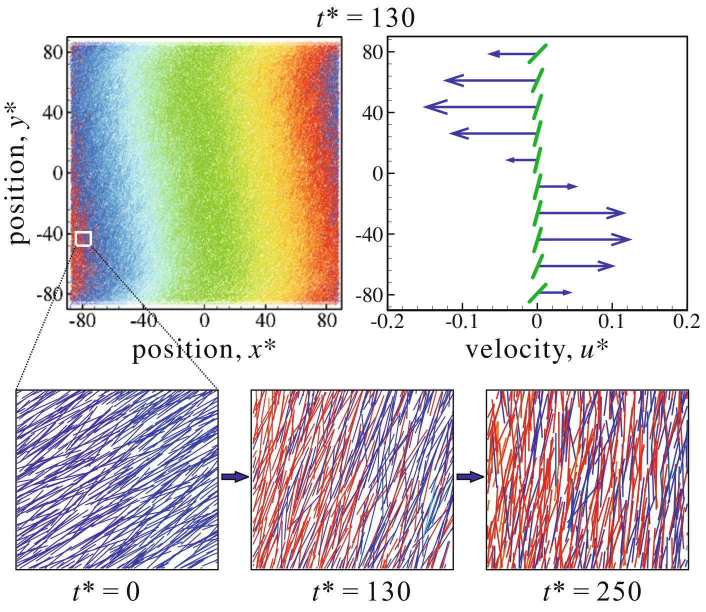

Fig. 6. Snapshot of molecules (upper left) and bulk velocity and molecular orientation (upper right) after the application of an electric field for $t^* = 130$, and close-up views of molecules (lower) at times t^* of 0, 130 and 250.

The close-up views of molecules show the change in molecular orientation upon the application of an electric field. Along with the change in molecular orientation, it can be seen that there is a bulk displacement of the molecules. As shown in the snapshot of molecules, the molecules in the upper region move to the left, while those in the lower region move to the right. The plot of bulk velocity at t^* = 130 shows an S-shaped velocity profile. This confirms the generation of a bulk flow due to molecular reorientation, as demonstrated by a 2D molecular dynamics simulation [19] and simulation using macroscopic continuum approach [17] and observed in a visualization experiment [20].

4.3. Precision of bulk properties calculation

Early GPUs only supported single-precision calculations. Recently, GPUs that support double-precision calculations such as the GeForce GTX 200 series and Tesla 10 series have emerged. However, double-precision operations are much expensive than single-precision operation, thus the performance of double-precision simulations should be lower than that for single-precision simulations. Furthermore, because the computation of intermolecular interaction in our simulation requires large numbers of registers and shared memory while the numbers of available registers and shared memory are relatively small, the use of double-precision data in the simulation would result in a large drop in performance. We verified this by running single-precision and double-precision versions of our simulation on a GPU (GTX 280), and we confirmed that the performance of double-precision simulation is about 1/30 of single-precision simulation. Therefore, to take advantage of the computational power of the GPU, we must perform single-precision simulations.

To verify the reliability of the single-precision simulations, we compare the profiles of velocity and transient velocities computed on a CPU using double-precision calculation with those computed on the GPU using double-precision and single-precision calculations, as shown in Fig. 7. It can be seen that the difference between results computed using single-precision and double-precision calculation on the GPU is comparable to that between the results computed using double-precision calculation on the CPU and GPU. Note that the orders of the force calculation on CPU and GPU are slightly different. As shown in [8], the different orders in the force summation result in different molecular trajectories, even for double-precision simulations that start from the same initial conditions. The difference in molecular trajectories for double-precision simulations performed on the CPU and GPU results in a slight difference in the bulk velocity, as shown in Fig. 7. For simulations on the GPU, the use of different precision would result in different molecular trajectories, which in turn result in a slight difference in the values of bulk velocity. However, as shown in Fig. 7 (right), the difference between the results of single-precision and double-precision simulations is not significance, because it is comparable to the statistical 'noise' of the average (bulk) velocity, indicated by the fluctuation in the transient velocity. Therefore, the use of single-precision simulations in the study of liquid crystalline flows is acceptable.

4.4. Conclusion

We have presented the development of a large-scale molecular dynamics simulation for the study of flow of fluids with anisotropic molecules such as liquid crystals. The simulation was implemented using a CUDA programming environment to take advantage of the massive parallel processing capability of a general purpose Graphics Processing Unit (GPU). The computation algorithm and its implementation on the GPU were discussed in detail. The computation of intermolecular interactions was implemented using the cell-list algorithm, and the calculation on the GPU was optimized by taking advantage of the high-speed shared memory. To update the cell lists at every time step, an efficient method using the cell index has been developed. The performance tests showed that the computation on a GPU is about 50 times faster than that on a single CPU core, and thus simulations involving a large number of molecules on a personal computer are possible. This should allow the extensive investigation of the molecular-level mechanisms underlying the macroscopic flow phenomena in

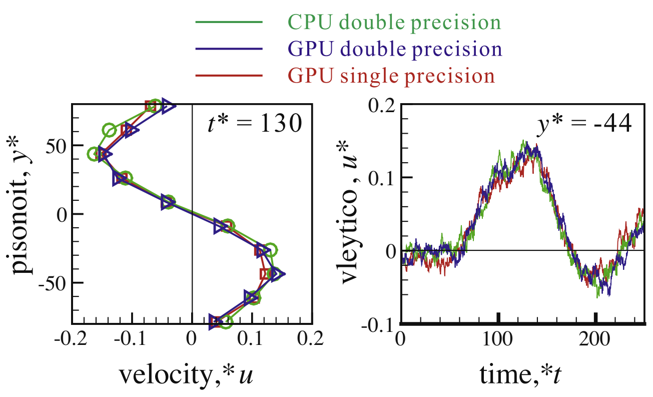

Fig. 7. Profile of velocity and transient velocity computed using single-precision and double-precision.

anisotropic fluids. The application of the simulation to the study of backflow has been demonstrated, and the issue related to the precision of the bulk flow calculation has been addressed. It has been shown that there is no significant difference in the bulk velocities computed using single-precision and double-precision simulations. Therefore, the GPU-based simulation is still reliable for simulation of systems that involve macroscopic flow.

References

[1] R.J. Rost, OpenGL Shading Language, first ed., Pearson Education, 2004.
[2] R. Fernando, M.J. Kilgard, The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, Addison-Wesley Professional, 2003.
[3] NVIDIA, CUDA Programming Guide Version 1.0, 2006 (current version 2.3, 2009).
[4] Sample of GPU accelerated applications: <http://www.nvidia.com/object/cuda_home.html>.
[5] J. Yang, Y. Wang, Y. Chen, GPU accelerated molecular dynamics simulation of thermal conductivities, J. Comput. Phys. 221 (2006) 799–804.
[6] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, K. Schulten, Accelerating molecular modeling applications with graphics processors, J. Comput. Chem. 28 (2007) 2618–2640.
[7] J.A. van Meel, A. Arnold, D. Frenkel, S. Portegies Zwart, R. Belleman, Harvesting graphics power for MD simulations, Mol. Simul. 34 (2008) 259–266.
[8] J.A. Anderson, C.D. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, J. Comput. Phys. 227 (2008) 5342–5359.
[9] M.S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. Legrand, A.L. Beberg, D.L. Ensign, C.M. Bruns, V.S. Pande, Accelerating molecular dynamic simulation on graphics processing units, J. Comput. Chem. 30 (2009) 864–872.
[10] D.J. Michel, D.J. Cleaver, Coarse-grained simulation of amphiphilic self-assembly, J. Chem. Phys. 126 (2007) 034506.
[11] G. Wedemann, J. Langowski, Computer simulation of the 30-nanometer chromatin fiber, Biophys. J. 82 (2002) 2847–2859.
[12] B. Mergell, M.R. Ejtehadi, R. Everaers, Modeling DNA structure, elasticity, and deformations at the base-pair level, Phys. Rev. E 68 (2003) 021911.
[13] T. Yamamoto, H. Kasama, Brownian dynamics simulation of multiphase suspension of disc-like particles and polymers, Rheol. Acta, in press. doi:10.1007/s00397-009-0405-5.
[14] P.G. de Gennes, J. Prost, The Physics of Liquid Crystals, second ed., Oxford University Press, 1993.
[15] F. Brochard, Backflow effects in nematic liquid crystals, Mol. Cryst. Liq. Cryst. 23 (1973) 51–58.
[16] S. Chono, T. Tsuji, Japan Patent No. 3586734 (20 August 2004) [in Japanese].
[17] S. Chono, T. Tsuji, Proposal of mechanics of liquid crystals and development of liquid crystalline microactuators, Appl. Phys. Lett. 92 (2008) 051905.
[18] Y. Mieda, K. Furutani, Two-dimensional micromanipulation using liquid crystals, Appl. Phys. Lett. 86 (2005) 101901.
[19] A. Sunarso, T. Tsuji, S. Chono, Molecular dynamics simulation of backflow generation in nematic liquid crystals, Appl. Phys. Lett. 93 (2008) 244106.
[20] T. Matsumi, T. Tsuji, S. Chono, Development of microactuators driven by liquid crystals (3rd report, visualization of velocity profiles between parallel plates), Trans. JSME B 75 (2009) 953–958 (in Japanese).
[21] J.G. Gay, B.J. Berne, Modification of the overlap potential to mimic a linear site–site potential, J. Chem. Phys. 74 (1981) 3316–3319.
[22] Houssa, L.F. Rull, S.C. McGrother, Effect of dipolar interactions on the phase behavior of the Gay–Berne liquid crystal model, J. Chem. Phys. 109 (1998) 9529–9542.
[23] D.C. Rapaport, The Art of Molecular Dynamics Simulation, second ed., Cambridge University Press, 2003.
[24] A. Dullweber, B. Leimkuhler, R. McLachlan, Symplectic splitting methods for rigid body molecular dynamics, J. Chem. Phys. 107 (1997) 5840–5851.