ARTICLE IN PRESS

Computer Physics Communications ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at SciVerse ScienceDirect

Computer Physics Communications

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

Accelerating dissipative particle dynamics with multiple GPUs

Sibo Wang [a,b], Junbo Xu [a,*], Hao Wen [a]

[a] State Key Laboratory of Multiphase Complex Systems, Institute of Process Engineering, Chinese Academy of Sciences, Beijing 100190, China

[b] University of Chinese Academy of Sciences, Beijing 100049, China

ARTICLE INFO

Article history:
Received 24 January 2013
Received in revised form
7 June 2013
Accepted 12 June 2013
Available online xxxx

Keywords:
Graphic processing unit
CUDA
Dissipative particle dynamics
Lubricant
Dispersant

ABSTRACT

Dissipative particle dynamics (DPD) simulation is implemented on multiple GPUs by using NVIDIA's Compute Unified Device Architecture (CUDA) in this paper. Data communication between each GPU is executed based on the POSIX thread. Compared with the single-GPU implementation, this implementation can provide faster computation speed and more storage space to perform simulations on a significant larger system. In benchmark, the performance of GPUs is compared with that of Material Studio running on a single CPU core. We can achieve more than 90x speedup by using three C2050 GPUs to perform simulations on an $80 * 80 * 80$ system. This implementation is applied to the study on the dispersancy of lubricant succinimide dispersants. A series of simulations are performed on lubricant–soot–dispersant systems to study the impact factors including concentration and interaction with lubricant on the dispersancy, and the simulation results are agreed with the study in our present work.

1. Introduction

Dissipative particle dynamics (DPD) is a mesoscopic simulation technique introduced by Hoogerbrugge [1] and Koelman [2]. It was devised to simulate dynamical and rheological properties of complex fluids; by introducing bead-and-spring type particles, polymers can be simulated as well [3,4]. With 20 years of development, this method has been successfully applied in different researches, including block-copolymer micro-phase separation [5], surfactant behavior [6,7], phase separation in binary immiscible fluids [8], aggregate structure in heavy crude oil [9] and so forth. Since DPD is a coarse-grained technique that only captures the gross features of mesoscopic portions, it can process a much larger spatial and temporal scale system than molecular dynamics (MD) can. Even though, computational cost is still a limiting factor. Most implementations of the classic DPD algorithm are running on CPU by serial programming. However, in recent years, the compute force of a single CPU core can hardly continue increasing. The developers start to try parallelizing the DPD simulation program with a variety of parallel computing platforms and technologies to improve the performance. On one hand, some commercial softwares like Material Studio, Culgi Library, LAMMPS, have already been available to run DPD simulations on multiple CPU cores in parallel using message-passing techniques. On the other hand, graphics processing units (GPUs) have become easily programmable and highly parallel processors; a GPUs-based implementation can be an alternative.

In recent years, GPUs have evolved to very powerful and highly parallel compute devices, and their parallel structure makes them even more effective than CPUs for processing large blocks of data in parallel. Besides, the NVDIA compute unified device architecture (CUDA) [10] has developed into a powerful and flexible programming toolkit for general-purpose computing on graphics processing units (GPGPU). The researchers started to use GPU's significantly computational power in scientific computing applications. A few previous works have investigated GPU implementations of specific algorithms used for MD. Stone et al. [11] have examined a GPU implementation for electrostatics. Anderson et al. [12] developed a general purpose MD code based on a neighbor-list structure that runs entirely on a single GPU, and the tests show the performance almost equivalent to that of fast 30 processor core distributed memory cluster. Later, van Meel et al. [13] present a double buffered partial updates of the cell-list, and conventional MD simulations using their GPU-based code can run 25–80 times faster than on a single CPU core at comparable prices. Friedrichs et al. [14] implemented all-atom protein molecular dynamics running entirely on GPU. In the field of DPD simulations, the study of GPU implementation has been reported rarely. HOOMD is a MD code package that can perform DPD simulations on GPU. The performance benchmark of HOOMD shows that DPD simulations running on a single NVIDIA GTX480 can even be faster than the CPU code parallelized over 64 cores [15]. In our previous work, we developed a GPU implementation of DPD simulations as well [16]. All parts of the simulation are running on

* Corresponding author. Tel.: +86 1082612330.
E-mail address: jbxu@home.ipe.ac.cn (J. Xu).

0010-4655/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.cpc.2013.06.011

GPU, including cell-list updating, force calculation and integrating forward. The benchmark results run on a single NVIDIA GTX285 under different scales of simulation system, all shows over 20× speedup against the serial version DPD provided by the Material Studio. Several implementations of MD [17,18] and lattice Boltzmann method (LBM) [19,20] have supported running simulations on multiple GPU devices. General computing programs running on multiple GPUs can achieve better performance. More importantly, simulations on quite large systems require such a great amount of memory that a single GPU cannot provide. However, to our knowledge, the study of DPD simulation running on multiple GPUs has not been reported yet.

In this paper, we present a POSIX-based implementation of DPD simulation which can be running on multiple GPUs in a server or workstation. Compared with the MPI-based application running on a multi-node cluster, our implementation has a limitation on the number of GPUs, but it is still a meaningful work. Large clusters cost much more money, and the maintenance work is heavy. Besides, it can be seen as a basis to develop an MPI-based implementation. The paper is organized as follows. We first briefly introduce technologies about CUDA programming on multiple GPUs and our domain decomposition method. Next, we give the details of the executions on each GPU in one time step of a simulation, including the computation and the communication. Then the performance of our implementation is tested. We run series of benchmarks to discuss how the performance can be impacted by simulation system size, GPU hardware, and the number of GPU used. Last, simulations on lubricant–soot–dispersant systems are performed by our implementation running on six Tesla C2050 GPUs.

2. Implementation

2.1. CUDA on multiple GPUs

In recent years, a computer containing multiple graphics devices has become more and more common. Users can add graphics card in PCI Express slot conveniently. Furthermore, some products of NVIDIA, such as GeForce GTX295, GTX 590 and GTX690, contain two GPUs on a single card already. Running CUDA computing programs on multiple GPUs can achieve much more throughput.

To use multiple GPUs, the CUDA program usually creates one CPU thread for each requested GPU in order to hold the corresponding CUDA context. POSIX threads and OpenMP are two APIs that are used for shared memory multiprocessing programming, and they can help to manage the CPU threads in the multi-GPU program. With CUDA and the message passing interface (MPI), the computing program can be implemented on larger systems like clusters that contain hundreds of GPUs. Besides data transfer between GPUs, CPU threads also have the ability to make global synchronizations with a barrier function provided by POSIX or other parallel programming APIs.

Since CUDA4.0 was introduced in 2011, multi-GPU programming has become easier and faster. Sharing GPUs across multiple threads is supported, and single thread can access to all GPUs now. More importantly, the new NVIDIA GPUDirect technology [21] makes a big improvement on GPU communication in a node. Data transfer between GPUs can be realized by peer-to-peer memory copy or access directly instead of using system memory as an intermediary.

In our implementation, POSIX was used for running a parallel code. Although the new version of CUDA has made a great effort on multi-GPU programming, some of the new features are just available for the devices of compute capability 2.0 and above. For the purpose that our program is supposed to be running on most kind of GPUs, we did not make optimizations intentionally for the new version of CUDA. As a result, our simulation program still has significant room for performance improvement.

2.2. Domain decomposition

For the implementation of multi-GPU DPD program, the solution domain should be decomposed into subdomains at first, and simulation in each subdomain is running on different GPU. The optimized domain decomposition must lead to a balanced distribution of computational load and also minimum communication between GPUs. Since we decided to use the cell-list neighbor-searching method before the force calculation, a cell is set as the minimum unit in domain decomposition. As Fig. 1 shows, we partitioned the 3D simulation box in the z-direction and got n piece; n is the number of GPUs in the system. Each piece has the same length in the x-direction and y-direction with the whole simulation box, while the length in the z-direction may differ slightly. The length in the z-direction of simulation box may not be divisible by n, which would have an impact on distributed load balancing.

In initialization work before simulation, the data about particles' properties, such as positions and velocities, are copied into different corresponding GPU memory. After domain decomposition, the simulations in every subdomain are running on each GPU. There is data communication between GPUs every time step, which makes the simulation in the whole domain look like running on a single GPU. The data communication mainly has two situations: when calculating the pair-force for one particle, there is a possibility that the data for the other particle are stored in another GPU; after position and velocity updating, some particles will not reside in the same subdomain any more.

2.3. Simulation details on GPUs

The communication between GPUs and CPUs is executed by data transfer in PCI Express. For some applications based on GPU, the bandwidth limitation of PCI Express is the major bottleneck for performance. In these implementations, only some computation-intensive parts are performed in parallel on GPU, while other parts just can be executed by CPU. GPU need to communicate with the host frequently to get necessary data, which results in plenty of waiting time for streaming processors (SPs). The waste of computing resource is the reason why those GPU-based applications cannot achieve a distinct performance speedup. Fortunately, all computational works in DPD simulation are highly parallel. Once the initialization work is done, the whole simulation can be executed by GPU. What we should consider is how to minimize the data transfer between GPUs. The implementation details about the computation and the communication are given as follows.

2.3.1. Generate cell-list

The pair interaction in DPD is short-ranged. Finding particles in the range of cutoff for force calculating can be extremely time-consuming on CPU when the length scale of simulation becomes large. Cell-list turned out to be an efficient method to reduce the computational complexity, and the high parallelism makes its algorithm suitable for implementation on GPU. In this method, the simulation box is typically decomposed into smaller domains, called cells. The information about correspondence between particles and cells can be achieved easily after decomposition, and plenty of data structures are available for storage [22]. In our single-GPU implementation, we choose to assign a fixed sized array of placeholders to every cell and physically copy particles' indexes into this array. The indexes are used to find the particles' properties data in the global memory, such as positions, velocities, and so on. It is an efficient scheme, although a portion of storage is wasted since each array must have enough space to store particles' indexes at the highest possible density. We set the side length

Please cite this article in press as: S. Wang, et al., Accelerating dissipative particle dynamics with multiple GPUs, Computer Physics Communications (2013), http://dx.doi.org/10.1016/j.cpc.2013.06.011