Acceleration of PIC Simulation with GPU*)

Junya SUZUKI, Hironori SHIMAZU[1)], Keiichiro FUKAZAWA[2)] and Mitsue DEN[3)]

Department of Information System Fundamentals, The University of Electro-Communications, Chofu 182-8585, Japan
[1)]*Keihanna Research Laboratories, National Institute of Information and Communications Technology, Soraku-gun 619-0289, Japan*
[2)]*Department of Earth and Planetary Sciences, Kyushu University, Fukuoka 812-8581, Japan*
[3)]*National Institute of Information and Communications Technology, Koganei 184-8795, Japan*

(Received 7 December 2010 / Accepted 24 March 2011)

Particle-in-cell (PIC) is a simulation technique for plasma physics. The large number of particles in high-resolution plasma simulation increases the volume computation required, making it vital to increase computation speed. In this study, we attempt to accelerate computation speed on graphics processing units (GPUs) using KEMPO, a PIC simulation code package [H. Matsumoto and Y. Omura, *Computer Space Plasma Physics*, pp.21-65 (1985)]. We perform two tests for benchmarking, with small and large grid sizes. In these tests, we run KEMPO1 code using a CPU only, both a CPU and a GPU, and a GPU only. The results showed that performance using only a GPU was twice that of using a CPU alone. While, execution time for using both a CPU and GPU is comparable to the tests with a CPU alone, because of the significant bottleneck in communication between the CPU and GPU.

Keywords: numerical simulation, particle-in-cell method, graphics processing units, high-performance computing, OpenMP

DOI: 10.1585/pfr.6.2401075

1. Introduction

In general, there are two plasma simulation methods: grid and particle. In this study we use a particle method called a particle-in-cell (PIC) simulation, in which electrons and ions are treated as particles, and electromagnetic fields are grid values. This method is then utilized for superior microscopic analysis compared to grid methods. It does, however, have a disadvantage in the much larger volume of computation required because of the numerous calculation points on the number of particles. To obtain statistical accuracy in the plasma simulation, an order of 1000 particles is required for one grid, but this large number increases the volume of computation. Consequently, it is important to raise computation speed. This study seeks to accelerate computation speed on Graphics Processing Units (GPUs) using KEMPO1 [1], a PIC simulation code package.

GPUs have been developed as image processing chips, but recently they are begun to be used as new computational resources since they offer high computational and cost performance. The theoretical calculation performance of some GPUs exceeds 1TFLOPS for a single GPU, which is very high compared to CPUs. A GPU has several hundred operation cores on a single chip. The ideal parallel computing with these cores contributes to the high theoretical calculation performance. In addition, CUDA [2], the GPU development environment, makes GPU programming easier.

Enabling high-performance computing using GPUs should also mean that the computation is done at low cost. GPUs, however, have only recently begun to be used in numerical calculations and the performance attainable when executing an actual application on a GPU is not known.

In this study, we examine the performance of KEMPO1 code on a GPU with CUDA.

2. KEMPO1 Algorithm

In KEMPO1, the following equations are solved. First, Maxwell's equations are written as

$$\nabla \times \boldsymbol{B} = \mu_0 \boldsymbol{J} + \frac{1}{c^2}\frac{\partial \boldsymbol{E}}{\partial t},$$

$$\nabla \times \boldsymbol{E} = -\frac{\partial \boldsymbol{B}}{\partial t},$$

where $\boldsymbol{B}$, $\boldsymbol{E}$, $\boldsymbol{J}$, c, and μ_0 are the magnetic field, electric field, current density, speed of light, and magnetic permeability, respectively.

The equation of motion for particles is then

$$\frac{d\boldsymbol{v}}{dt} = \frac{q}{m}(\boldsymbol{E} + \boldsymbol{v} \times \boldsymbol{B}),$$

where q, m, and $\boldsymbol{v}$ are the charge, mass, and velocity of particles, respectively. The initial condition is given by the Poisson equation,

$$\frac{\partial E_x}{\partial x} = \frac{\rho}{\epsilon_0},$$

author's e-mail: junya@hpc.is.uec.ac.jp

*) This article is based on the presentation at the 20th International Toki Conference (ITC20).

$$\frac{\partial B_x}{\partial x} = 0,$$

where ρ and ϵ_0 are the charge density and electric permittivity, respectively.

These equations are solved with leap-frog time integration. The calculation sequence is as follows: First, the magnetic field is obtained by solving Maxwell's equations and the equation of motion for particles is solved using the value of electromagnetic field assigned to the grids. We then update the particle positions, calculate the current density by using the particle information, and obtain the magnetic field and electric field. These processes constitute one step. Time integration is performed by repeating the processes, and we advance the simulation.

One time step of KEMPO1 comprises the following five main functions:
BFIELD: calculates the magnetic field
VELCTY: calculates the particle velocities
POSITN: updates the particle positions
CURRNT: calculates the current density at the grids
EFIELD: calculates the electric field

The VELCTY function obtains the particle velocity by using the value of the electromagnetic field assigned to the grids. This process occupied more than 50 percent of the total calculation time on the CPU. The POSITN function then updates the position of the particles called twice in one step. This process occupies about 10 percent of the total calculation time on the CPU. The CURRNT function calculates the current at the grids based on the physical quantities of particles. This process occupies more than 35 percent of the total calculation time on the CPU.

For CURRNT, it is necessary to gather the particles' physical quantities into the grids. This method generally handles many more particles than there are grids, and so requires a reduction operation in parallel computing [3]. In the GPU, the degree of parallel computing is large because of the large number of threads described in sec. 3, and bottlenecks on the GPU. Moreover, there is frequent random memory access since the particles are constantly moving. This random access must be avoided, since it is extremely slow on the GPU [4].

The EFIELD function calculates the electric field and BFIELD calculates the magnetic field. These processes occupy less than one percent of the total volume of calculations on the CPU.

The calculation of particles occupies a large percentage of the total calculation and is independent, so it can be computed in parallel. This should accelerate this calculation on the GPU.

3. Implementation of KEMPO1 on GPU

On a GPU the maximum number of threads that can be created simultaneously is in the order of 10^{12}, which is larger than the number for the physical computing core. Therefore, threads can be changed readily. In this study, then, we perform a calculation of particles with a large number of threads, in which each thread manages multiple particles. To calculate the grids, we create the same number of threads as there are grids.

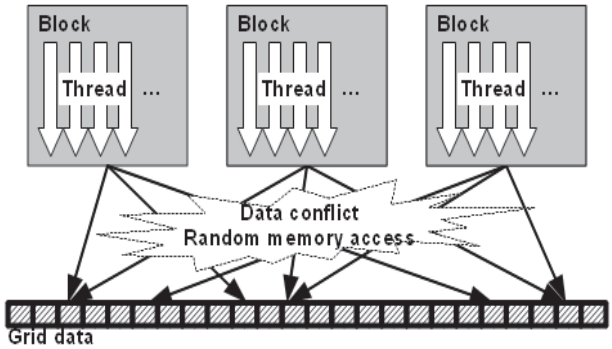

Fig. 1 CURRNT on GPU

With CUDA, it is relatively easy to translate the functions of BFIELD, VELCTY, POSITN, and EFIELD on the CPU to the GPU. However, it is difficult to translate the CURRNT function on the CPU to the GPU. Thus, as mentioned above, a bottleneck in the CURRNT calculation forms on the GPU.

In the CURRNT process, random memory access and data conflicts occur frequently, as shown in Fig. 1. With several hundred cores of GPU in fine-grained parallel computing and the shared memory of the GPU architecture, calculation of the current density on the grids must be handled with care when using the particle information.

There are two issues making the CURRNT process on a GPU slower than it is on a CPU:
(1) Random memory access using the grid information
(2) Frequent synchronization for exclusive control prevented because of data conflict among threads

The delay caused by (1) can be improved by conducting calculations on high-speed memory of 16 KB in each block, which is shared memory. In (2), unexpected overhead occurs because the GPU handles a larger number of threads than other parallel computing. We assign multiple arrays for the grid data of the simulation domain on the shared memory, so that each thread that calculates simultaneously can access these separated arrays. Thus, data conflict among many threads can be avoided without synchronization. However, because the shared memory's volume is very small, the arrays of all grid data cannot be assigned for each thread, or else we would not be able to store all grid information in it.

We therefore introduce the domain decomposition method, which is frequently used in parallel computing. By dividing the domain to reduce information on the grids, we can use the shared memory on the GPU.

First, we divide the one-dimensional domain. Figure 2 shows that the domain is divided into three parts. The particles are also divided by the domain where they exist. Then, on the GPU, the processes of each divided domain are per-

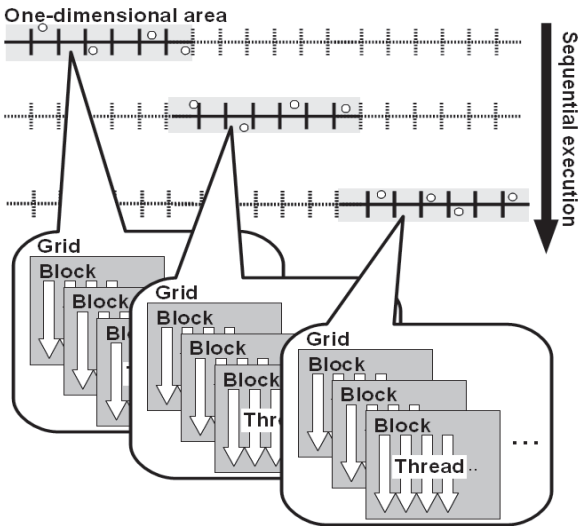

Fig. 2 Domain decomposition method on GPU

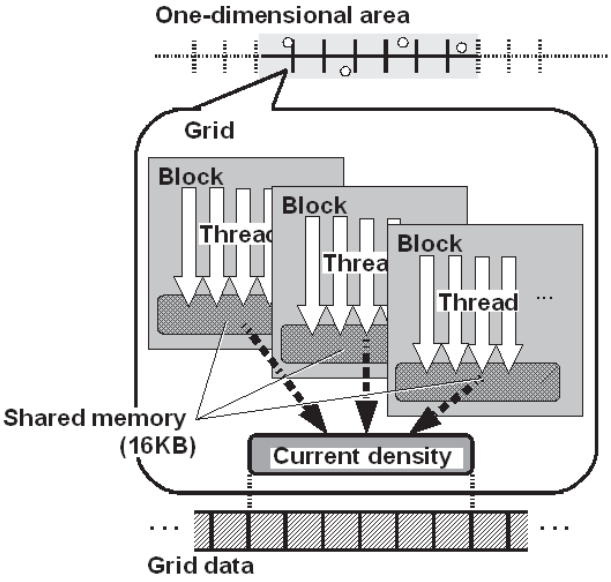

Fig. 3 Sum of current density on GPU

formed sequentially and the current density in each divided domain is obtained. Finally, we join the domains and obtain the current density in the whole domain.

Here, the process of the sum of current density in the divided domain is described in Fig. 3. In the block, the sum of the current density to which each particle contributes is calculated on the shared memory. We then sum the current density in each block and so obtain the current density in the divided domain by summing up those in each block.

On the calculation in the block, we also assign multiple arrays for the current density of the divided domain on the shared memory. The same number of threads as for these arrays can access a corresponding array without synchronization, so that we can reduce the frequent synchronization. These arrays are then summed. As a consequence, the size of the divided domain becomes very small.

We also take the larger area compared to the divided area, so that the adjacent domains overlap. Thus, we can keep the grouping of particles to once every few dozen steps, as opposed to every step without overlapping region. The process of grouping particles is unsuitable for the GPU since the branch process and many instructions for replacement of the particle data occur in the device memory. Implementation also becomes complicated, so in this study we execute the program for these parts on the CPU.

4. Results

In the tests conducted in this study, we perform two cases of benchmarking with a small and large grid number. In these two cases, we run the KEMPO1 code using only a CPU, both a CPU and GPU, and only a GPU, respectively. Thus there are a total of six runs.

Here, in the implementation using both the CPU and GPU, we run only the CURRNT function on the CPU and the other functions on the GPU. We expect to easily speed up the program without using a complicated algorithm to optimize it for the GPU. However, since the CPU and GPU have separate memory, it is necessary to transfer data between them at every step.

For the test environment, we used a CPU with an Intel Xeon X5550 and a GPU with an NVDIA Tesla C1060 on the same server. Their theoretical calculation performances are 43.56×4 TFLOPS and 933 TFLOPS, respectively. The size of the main memory is 74 GB on this server and the GPU has 4 GB of device memory. For the CPU we used four cores and performed parallel computation with OpenMP. For the GPU, we used CUDA. Here, we measured the execution time of the five main functions of the KEMPO1 code: VELCTY, POSITN, CURRNT, EFIELD, and BFIELD. In the small size there are 128 grids, and the large size has 8192. The number of particles per grid is 10,000. The memory needed depends on the number of particles. For the small size, 20 MB is needed; while for the large size case, 1.2 GB is required. Here each divided domain has 32 grid points in all calculation cases. That is, in the small grid calculation case (128 grid points), the entire domain is divided by 4, and in the large case (8192), it is divided by 256. We also define each block as having 256 threads. The number of blocks is altered by changing the number of particles.

Figure 4 shows the results for the case of the small grid number.

Performance, the reciprocal run time number, using only the GPU is twice that using the CPU alone. On the other hand, run with both the CPU and GPU is slower than that with the CPU alone because of the large overhead, which is attributable to the communication time between the CPU and GPU. In the run with the GPU alone, overhead is attributable to the computing time for division of the area per step and the communication time between the CPU and GPU.

The performance of VELCTY and POSITN with only

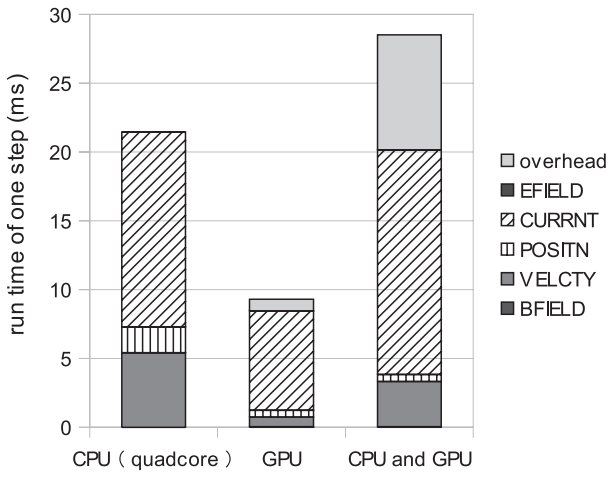

Fig. 4 Results of small grid number (128)

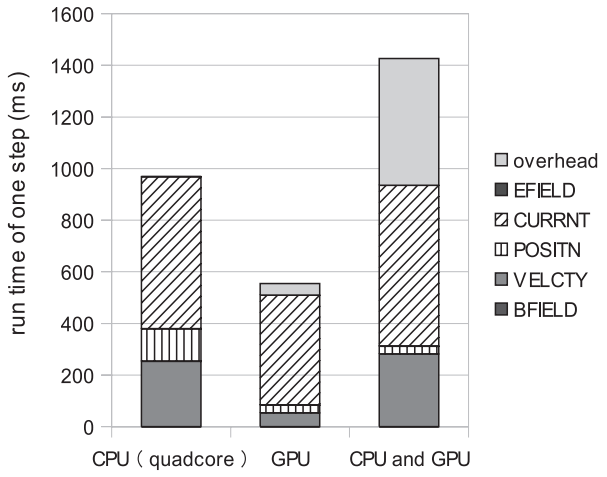

Fig. 5 Results of large grid number (8192)

the GPU is also five times that using the CPU alone. The performance of CURRNT with only the GPU is 1.5 times that using the CPU alone. The run of VELCTY with both the CPU and GPU is slower than that with the CPU alone. This is because we do not optimize part of the code processed on the GPU. To utilize the GPU's shared memory, the algorithm needs to be changed substantially.

Figure 5 shows the results for the large grid number. The performance for the large grid number is the same as that for the small number, so we expect that the performance for the higher dimensional case is the same on the GPU.

5. Summary

This study confirmed that the performance of KEMPO1 code, which is used for PIC plasma simulation, with a GPU is twice as fast for both small and large grid numbers compared to that with a CPU using four cores.

In these two cases, performance using both a CPU and GPU is lower than that using a CPU alone. Optimization for a GPU is essential, and the algorithm needs to be changed radically. This makes it difficult to obtain high-performance computation using both a CPU and GPU.

Acceleration is possible with a GPU. It is important to devise an algorithm to make it suitable for calculation using a GPU architecture, which has trouble obtaining high performance on the GPU, such as with the CURRNT function. The shared memory on the GPU must be utilized.

[1] H. Matsumoto and Y. Omura, *Computer Space Plasma Physics*, pp.21-65 (1985).
[2] *NDVIA CUDA programming guide 1.1*, http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf.
[3] Y. Akiyama *et al.*, Information Processing Society of Japan **66**, 1 (1997).
[4] G. Stantchev *et al.*, Journal of Parallel and Distributed Computing **68**, 1339 (2008).