Effectiveness of GPGPU for Solving the Magnetohydrodynamics Equations Using the CIP-MOCCT Method*)

Ryosuke UEDA, Yutaka MATSUMOTO, Masafumi ITAGAKI and Shun-ichi OIKAWA

Graduate School of Engineering, Hokkaido University, Sapporo 060-8628, Japan

(Received 5 January 2011 / Accepted 22 April 2011)

A simple parallelization approach using General Purpose computation on Graphics Processing Unit was applied for solving the MHD equations using the CIP-MOCCT method. We investigated the efficiency of this parallelization approach and found that the computational speed of the modified code is significantly improved despite the simple modification.

Keywords: MHD simulation, CIP method, MOCCT method, GPGPU, parallel computing

DOI: 10.1585/pfr.6.2401092

1. Introduction

A method for the plasma dynamics analysis is numerical simulation by solving the MHD equations. However, solving them in three dimensions is expensive and requires excessive computing time; hence, faster processors are required. Rapid improvement in Central Processing Units (CPUs) over the last several decades has enabled us to run larger and more accurate numerical simulations. The results of these simulations have made great contributions to plasma and fusion research. The field of numerical simulation has a high potential for future growth. Recently, however, there have been few CPU clock speed enhancements. Instead, CPU performance is increased using multiple processors via the multi-core strategy. This indicates that further improvements of computing speed are obtained by parallel computation. Therefore, it will be necessary for us to employ the multi-core processor technology.

We are interested in using the Graphics Processing Unit (GPU) as a multi-core processor to perform parallel computations. The GPU is a multi-core processor developed for graphics processing and affords massive parallel computations. In recent years, the high parallel performance of the GPU has motivated research outside graphics processing, i.e., general purpose computation; such techniques are called General Purpose computation on GPU (GPGPU).

GPGPU has been used in some simulation studies. In fluid dynamics, Brandvik and Pullan [1] solved two-dimensional (2D) and three-dimensional (3D) Euler equations with uniform grids reducing the computational time by a factor of 29 (2D) and 16 (3D). Elsen *et al.* [2] solved 3D Euler equations on multi-block meshes. In plasma physics, Stantchev *et al.* [3] discussed the efficient implementation of the particle-in-cell plasma simulation code. Wang *et al.* [4] solved the MHD equations by using an adaptive mesh finite volume method. However, there are few studies on the application of GPU for solving the full MHD equations.

Because the experts in numerical simulations are not always experts in parallel computation, the approach involving GPGPU must be made simple to use. It is important for us to know the efficiency of a simple parallelization approach for solving numerical simulations. In the present study, we apply this approach to solve the MHD equations and investigate its efficiency.

For simple parallelization, an explicit scheme is convenient, because the solution of simultaneous equations is unnecessary. We adopt a combination of the Constrained Interpolation Profile (CIP) [5], Method of Characteristics (MOC) [6], and Constrained Transport (CT) methods [7]; we call it the CIP-MOCCT method [8], which is an explicit scheme. The numerical phase errors of the advection equation solved by the CIP method are smaller than those produced by other schemes [9]. In addition, the CIP method can accurately follow the contact discontinuity and the boundary between the plasma and vacuum. This method has been used in a number of hydrodynamic [10] and MHD simulations [11, 12]. However, as mentioned in [13], the application of the CIP method to the MHD problems is difficult because of the traverse mode Alfvén wave and the divergence-free condition of a magnetic field. Thus, a special treatment of the Alfvén wave and the divergence-free condition is required. The MOCCT method is a standard technique for treating the Alfvén wave and for satisfying the divergence-free condition. The CIP-MOCCT method is not used widely in the field of fusion science, but it has been employed in several astrophysics studies [14-19]. To the best of our knowledge, this is the first attempt to apply GPGPU to the CIP-MOCCT method.

author's e-mail: ru04srk@fusion.qe.eng.hokudai.ac.jp

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

2. GPGPU and CUDA

Although performance improvements using GPGPU have been studied [20], GPU is essentially the hardware for graphics processing. In early studies employing GPGPU, programmers had to apply their algorithms and data structures to the graphics programming model, which was not always straightforward, and hence, in-depth knowledge of graphics programming was required.

Compute Unified Device Architecture (CUDA) [21] is the development environment for GPGPU provided by NVIDIA. as NVIDIA® CUDA™. CUDA makes the development of GPU computing applications much easier and more efficient than earlier attempts of using GPGPU. CUDA-C is an extension of the C Programming Language and is easy to learn for programmers with knowledge of C.

3. Implementation of CUDA

To calculate efficiently in CUDA, we have to appropriately use the several types of memories on GPU, e.g., global memory, shared memory, constant memory, texture memory and register. In particular, the register and shared memory accessed via high bandwidths are essential for higher optimization, although these memories are difficult to use efficiently. For this reason, the optimization strategy involves the effectively use of these memories. On the other hand, although the access speed of the global memory is low, it has much larger memory space and is easier to program compared with the shared memory. In this study, a computational code is modified via a simple parallelization approach using only the global memory.

In a parallelization approach, we need to find bottlenecks in the existing code. Such bottlenecks were found in triple nested loops in our existing 3D MHD simulation code. In CUDA, the simplest parallelization approach for these loops is achieved by mapping each array element to the corresponding thread on the GPU (Fig. 1). This means that the number of threads invoked on the GPU is equal to the number of array elements. Because the shared memory is not used, this approach is not the best for optimization in terms of the calculation speed. However, it has major advantages: (i) we can easily modify the existing code and (ii) it is applicable to other numerical schemes.

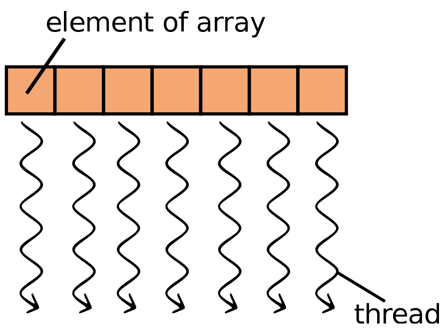

Fig. 1 Mapping of each array element to the corresponding thread on the GPU.

List 1 Example of existing CPU code

```
void func(...) {
    for(int i = 0; i<NX; ++i) {
        for(int j = 0; j<NY; ++j) {
            for(int k = 0; k<NZ; ++k) {
                // operations for
                // each element
                a[i][j][k]
                    = b[i][j][k-1]+ ...;
            }
        }
    }
}
```

List 2 Example of modified code

```
void func(...) {
   func_kernel<<<BLOCKS,THREADS>>>(...);
}

__global__
void func_kernel(...) {
    const unsigned int idx
           = blockDim.x*blockIdx.x +
             threadIdx.x;
    if(idx>= NX*NY*NZ) return;
    const unsigned int i = idx/(NY*NZ);
    const unsigned int j = (idx/NZ)%NY;
    const unsigned int k = idx%NZ;

    a[idx] = b[IDX(i,j,k-1)] + ...;
}

__device__
unsigned int IDX(int i,int j,int k) {
    return NY*NZ*i + NZ*j + k;
}
```

List 1 shows the triple nested loop typically used in the 3D finite difference method in the computation on the CPU. In List 1, the elements of the 3D array are calculated while incrementing the indices `i`, `j`, and `k` of the triple nested loop. The modified version of the code in List 1 is shown in List 2. There are three functions `func`, `func_kernel`, and `IDX`. `func` is executed on the CPU and calls `func_kernel`, which is executed on the GPU. Variables `BLOCKS` and `THREADS` in parentheses, <<< and >>>, are parameters specifying the number of threads run on the GPU. In the CUDA architecture, threads are arranged into threadblocks. `BLOCKS` and `THREADS` indicate the number of threadblocks and the number of threads per threadblock, respectively. The product of `BLOCKS` and `THREADS` is the number of the total threads running on the GPU. `func_kernel` is a "kernel function" invoked by the CPU and runs on the GPU. The processes performed in each thread are implemented in this kernel. Such kernel functions require a `__global__` qualifier for their definition. At the beginning of this kernel, the thread index `idx`, which indicates the location of the threads, is calculated using built-in variables `blockDim`, `blockIdx`, and `threadIdx`. The index `idx` corresponds to the index of the array. We manipulate the 3D array as the 1D array in List 2. The function `IDX` is invoked to obtain the index of the 1D array from the index of the 3D array. The `__device__` qualifier is required for the functions executed on the GPU.

In the present study, the modifications shown in List 2 have been implemented for all triple nested loops. As a result, these loops can be parallelized to be computed on the GPU.

4. MHD Equations and Numerical Scheme in the Developed Code

4.1 MHD equations

MHD equations in the developed code are expressed as follows.

$$\frac{\partial \rho}{\partial t} + \boldsymbol{V}\cdot\nabla\rho = -\rho\nabla\cdot\boldsymbol{V}, \tag{1}$$

$$\frac{\partial \boldsymbol{V}}{\partial t} + \boldsymbol{V}\cdot\nabla\boldsymbol{V} = \frac{1}{\rho}(-\nabla P + \boldsymbol{J}\times\boldsymbol{B} + \nabla\cdot\boldsymbol{\Phi}), \tag{2}$$

$$\frac{\partial P}{\partial t} + \boldsymbol{V}\cdot\nabla P = -\gamma P\nabla\cdot\boldsymbol{V} + (\gamma-1)(\boldsymbol{\Phi}:\nabla\boldsymbol{V} + \eta J^2), \tag{3}$$

$$\frac{\partial \boldsymbol{B}}{\partial t} = -\nabla\times\boldsymbol{E}, \tag{4}$$

$$\boldsymbol{E} = -\boldsymbol{V}\times\boldsymbol{B} + \eta\boldsymbol{J}, \tag{5}$$

$$\boldsymbol{J} = \nabla\times\boldsymbol{B}, \tag{6}$$

$$\nabla\cdot\boldsymbol{B} = 0. \tag{7}$$

Here, ρ is the density, $\boldsymbol{V}$ the velocity, P the pressure, $\boldsymbol{J}$ the current density, $\boldsymbol{\Phi}$ the viscosity stress tensor, $\boldsymbol{B}$ the magnetic field, $\boldsymbol{E}$ the electric field, γ the ratio of specific heat and η the resistivity.

4.2 CIP method

For fluid dynamics equations (1)-(3), we apply the CIP method [5], which is an explicit numerical technique applied to various problems, including fluid dynamics. In this method, a value and its derivative are treated as independent physical variables. It enables us to accurately compute phenomena such as shockwaves, whose values change sharply in space.

4.3 CT method

The divergence-free condition of magnetic field $\boldsymbol{B}$ (Eq. (7)) does not appear in the time evolution equations (Eqs. (4)-(5)). It is pointed out that even small errors in satisfying the divergence-free condition cause artifacts such as a non-physical force parallel to the magnetic field [22]. In order to satisfy the above condition, the CT method [7] is adopted in the developed code.

Other schemes such as the projection scheme [22] require a Poisson solver. Therefore, we also have to develop a Poisson solver that can use the GPU. On the other hand,

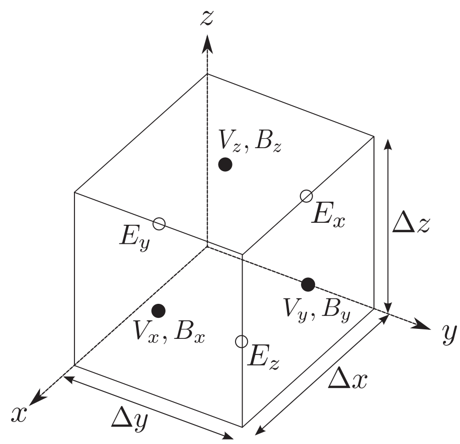

Fig. 2 Location of the physical variables $\boldsymbol{V}$, $\boldsymbol{B}$, and $\boldsymbol{E}$ in the CT method.

the CT method enables the satisfaction of the divergence-free condition with the accuracy of the finite difference method by only defining $\boldsymbol{V}$, $\boldsymbol{B}$, and $\boldsymbol{E}$ on the staggered grid (Fig. 2).

4.4 MOC method

To compute the time evolution of $\boldsymbol{B}$ by using Eq. (4), it is necessary to calculate $\boldsymbol{E}$ from Eq. (5). But in the staggered grid defined by the CT method, the variables of $\boldsymbol{V}$ and $\boldsymbol{B}$ are not defined at the location of the variables of $\boldsymbol{E}$. Thus, it is necessary to interpolate the values of $\boldsymbol{V}$ and $\boldsymbol{B}$ at the location of the variables of $\boldsymbol{E}$. However, it is well known that numerical oscillation in the Alfvén wave propagation is observed when the simple average of $\boldsymbol{V}$ and $\boldsymbol{B}$ values at the location of the variables of $\boldsymbol{E}$ is used. The MOC method [6] is used for the stable computation of the Alfvén wave. From Eqs. (1), (2), and (4)-(6), a set of the characteristic equations are given by,

$$\left[\frac{\partial}{\partial t} + \left(V_x + \frac{B_x}{\sqrt{\rho}}\right)\frac{\partial}{\partial x}\right]\left(V_y - \frac{B_y}{\sqrt{\rho}}\right) = 0, \tag{8}$$

$$\left[\frac{\partial}{\partial t} - \left(V_x - \frac{B_x}{\sqrt{\rho}}\right)\frac{\partial}{\partial x}\right]\left(V_y + \frac{B_y}{\sqrt{\rho}}\right) = 0. \tag{9}$$

These equations show that $V_y \mp B_y/\sqrt{\rho}$ are constant on the characteristic lines and that the value of V_y and B_y at E_z are obtained. In addition, other values of the components of $\boldsymbol{V}$ and $\boldsymbol{B}$ at the locations of the variables of $\boldsymbol{E}$ are obtained similarly.

5. Comparison between CPU and GPU

To evaluate the accuracy of the GPU solution, we applied both the CPU and GPU codes to a 1D MHD shock tube problem [23]. The initial condition for this problem was separated into two uniform states at the position of $x = 0.5$. The left state was $(\rho, P, V_x, V_y, B_x, B_y) = (1, 1, 0, 0, 0.75, 1)$ and the right state was $(\rho, P, V_x, V_y, B_x, B_y) = (0.125, 0.1, 0, 0, 0.75, -1)$. We

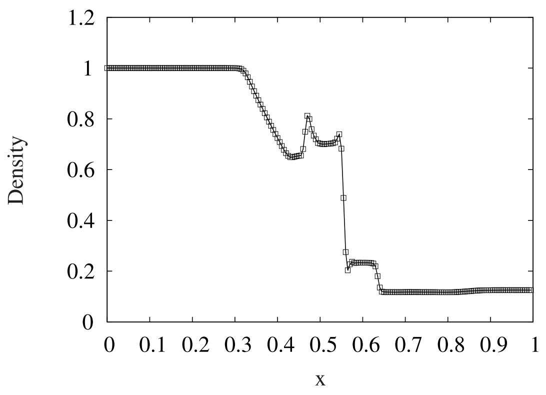

Fig. 3 Density profile at $t = 0.1$. The solid line is the solution of the CPU code. The squares are the solution of the GPU code.

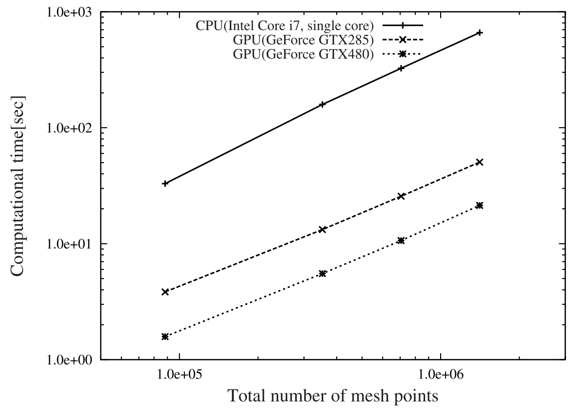

Fig. 4 Computational time using a CPU (Core i7) and two GPUs (GTX285 and GTX480). The horizontal axis is the total number of mesh points ($NX \times NY \times NZ$).

Table 1 Specifications of GPU

GPU	GTX285[a]	GTX480[b]
Total numbers of cores	240	480
Total amount of global memory	0.999 GB	1.499 GB
Total amount of shared memory per block	16384 B	49152 B
Total number of registers available per block	16384	32768
Clock rate	1.48 GHz	1.40 GHz

[a]NVIDIA Geforce GTX285
[b]NVIDIA Geforce GTX480

set $\Delta x = 1/200$, $\Delta t = \Delta x/10$, $\gamma = 2$, and $\eta = 0$. The results at $t = 0.1$ are shown in Fig. 3. It can be seen that the GPU solution is almost the same as the CPU solution. Moreover, these results also coincide with Brio-Wu's results. This proves the validity of our developed code.

The performance of the modified 3D MHD simulation code using GPU in double precision was evaluated. The simulation settings were the same as for the above-mentioned MHD shock tube problem. We evaluated the CPU time from the start of the time integration ($t = 0$) until 100 time steps. Computational times using a CPU (Intel® Core$^{\mathrm{TM}}$ i7, 2.93 GHz) and two GPUs (NVIDIA® GeForce® GTX 285 and GTX 480) were measured. Although Intel Core i7 is a multi-core processor, the computational time was measured for the case of using a single processor. The specifications of the GPUs are shown in Table 1.

The computational times measured for various mesh points are summarized in Fig. 4. It can be seen that for this calculation GTX285 is about 13 times faster than the CPU. Furthermore, GTX480 is about 30 times faster than the CPU. These results show the effectiveness of our simple parallelization using the GPU. This means that the improvement in speed is independent of the total number of mesh points ($NX \times NY \times NZ$). In addition, these results are independent of the problems, because the CIP-MOCCT method is an explicit scheme without a relaxation algorithm.

6. Summary and Discussion

We developed an MHD simulation code using the CIP-MOCCT method and modified it to extend its application to the parallel computing code using the GPU. Although we used a simple parallelization approach without utilizing the shared memory, the computational speed of the modified code was improved significantly.

The total memory on a GPU is smaller than that on a CPU. For this reason, increase in the number of mesh points lead to shortage in the GPU memory space. The use of multiple GPUs is considered a solution for this problem. However, the transfer of data between the CPU and the GPU or between computers in the multi GPU technique incurs large costs.

[1] T. Brandvik and G. Pullan, Proc. 48th AIAA Aerospace Sciences Meeting and Exbit, AIAA Press, 2008-607 (2008).
[2] E. Elsen, P. LeGresley and E. Darve, J. Comput. Phys. **227**, 10148 (2008).
[3] G. Stantchev, W. Dorland and N. Gumerov, J. Parallel Distrib. Comput. **68**, 1339 (2008).
[4] P. Wang, T. Abel and R. Kaehler, New Astronomy **15**, 581 (2010).
[5] H. Takewaki, A. Nishiguchi and T. Yabe, J. Comput. Phys. **61**, 261 (1985).
[6] J. Hawley and J.M. Stone, Comput. Phys. Commun. **89**, 127 (1995).
[7] C.R. Evans and J.F. Hawley, ApJ **332**, 659 (1988).
[8] T. Kudoh and K. Shibata, CFD journal **8**, 56 (1999).
[9] T. Tajima, *Computational Plasma Physics: With Application to Fusion and Astrophysics* (Addison-Wesley, New York, 1989) Chap. 6.
[10] T. Yabe, F. Xiao and T. Utsumi, J. Comput. Phys. **169**, 556 (2001).
[11] N. Nishino, J. Plasma Fusion Res. SERIES **6**, 395 (2004).
[12] R. Ishizaki and N. Nakajima, J. Plasma Fusion Res. SERIES **8**, 995 (2009).
[13] Y. Matsumoto and K. Seki, Comput. Phys. Commun. **179**, 289 (2008).
[14] T. Kudoh and K. Shibata, ApJ **476**, 632 (1997b).
[15] T. Kudoh, R. Matsumoto and K. Shibata, ApJ **508**, 186

(1998).
[16] T. Kudoh, R. Matsumoto and K. Shibata, PASJ **54**, 121 (2002a).
[17] T. Kudoh, R. Matsumoto and K. Shibata, PASJ **54**, 267 (2002b).
[18] S.X. Kato, T. Kudoh and K. Shibata, ApJ **565**, 1035 (2002).
[19] Y. Sofue, H. Kigure and K. Shibata, PASJ **57**, 39 (2005).
[20] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone and J.C. Phillips, Proceedings of the IEEE **96**, 879 (2008).
[21] NVIDIA CUDA Programming Guide.
[22] J.U. Brackbill and D.C. Barnes, J. Comput. Phys. **35**, 426 (1980).
[23] M. Brio and C.C. Wu, J. Comput. Phys. **75**, 400 (1988).