Computer Physics Communications 182 (2011) 2132–2160

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

Efficient magnetohydrodynamic simulations on graphics processing units with CUDA

Hon-Cheng Wong [a,b,*], Un-Hong Wong [b], Xueshang Feng [c], Zesheng Tang [b]

[a] Faculty of Information Technology, Macau University of Science and Technology, Macao, China
[b] Space Science Institute, Macau University of Science and Technology, Macao, China
[c] SIGMA Weather Group, State Key Laboratory for Space Weather, Center for Space Science and Applied Research, Chinese Academy of Sciences, Beijing 100190, China

ARTICLE INFO

Article history:
Received 8 February 2010
Received in revised form 12 March 2011
Accepted 15 May 2011
Available online 18 May 2011

Keywords:
MHD simulations
GPUs
CUDA
Parallel computing

ABSTRACT

Magnetohydrodynamic (MHD) simulations based on the ideal MHD equations have become a powerful tool for modeling phenomena in a wide range of applications including laboratory, astrophysical, and space plasmas. In general, high-resolution methods for solving the ideal MHD equations are computationally expensive and Beowulf clusters or even supercomputers are often used to run the codes that implemented these methods. With the advent of the Compute Unified Device Architecture (CUDA), modern graphics processing units (GPUs) provide an alternative approach to parallel computing for scientific simulations. In this paper we present, to the best of the author's knowledge, the first implementation of MHD simulations entirely on GPUs with CUDA, named *GPU-MHD*, to accelerate the simulation process. *GPU-MHD* supports both single and double precision computations. A series of numerical tests have been performed to validate the correctness of our code. Accuracy evaluation by comparing single and double precision computation results is also given. Performance measurements of both single and double precision are conducted on both the NVIDIA GeForce GTX 295 (GT200 architecture) and GTX 480 (Fermi architecture) graphics cards. These measurements show that our GPU-based implementation achieves between one and two orders of magnitude of improvement depending on the graphics card used, the problem size, and the precision when comparing to the original serial CPU MHD implementation. In addition, we extend *GPU-MHD* to support the visualization of the simulation results and thus the whole MHD simulation and visualization process can be performed entirely on GPUs.

1. Introduction

Magnetohydrodynamic (MHD) equations can be used in modeling phenomena in a wide range of applications including laboratory [5], astrophysical [48], and space plasmas [11]. For example, 3D MHD simulations have been widely adopted in space weather simulations. The historical review and current status of the existing popular 3D MHD models can be found in [8] and [9], respectively. However, MHD equations form a nonlinear system of hyperbolic conservation laws, which is so complex that high-resolution methods are necessary to solve them in order to capture shock waves and other discontinuities. These high-resolution methods are in general computationally expensive and parallel computational resources such as Beowulf clusters or even supercomputers are often utilized to run the codes that implemented these methods [29,16,22,17,59].

In the last few years, the rapid development of graphics processing units (GPUs) makes them more powerful in performance and more programmable in functionality. By comparing the computational power of GPUs and CPUs, GPUs exceed CPUs by orders of magnitude. The theoretical peak performance of the current consumer graphics card NVIDIA GeForce GTX 295 (with two GPUs) is 936G floating-point operations per second (FLOPS) per GPU in single precision while a CPU (Core 2 Quad Q9650 — 3.0 GHz) gives a peak performance of around 96GFLOPS in single precision. The release of the *Compute Unified Device Architecture* (*CUDA*) [30] hardware and software architecture is the culmination of such development. With CUDA, one can directly exploit a GPU as a data-parallel computing device by programming with the standard C language and avoid working with a high-level shading language such as Cg [28], which requires a significant amount of graphics specific knowledge and was previously used for performing computation on GPUs. Detailed performance studies on GPUs with CUDA can be found in [4] and [41].

CUDA is a general purpose parallel computing architecture developed by NVIDIA. It includes the CUDA Instruction Set

* Corresponding author at: Faculty of Information Technology, Macau University of Science and Technology, Macao, China.
E-mail address: hcwong@ieee.org (H.-C. Wong).

0010-4655/$ – see front matter
doi:10.1016/j.cpc.2011.05.011

Architecture (ISA) and the parallel compute engine. An extension to C programming language and its compiler are provided, making the parallelism and high computational power of GPUs available not only for rendering and shading, but also for solving many computationally intensive problems in a fraction of the time required on a CPU. CUDA also provides basic linear algebra subroutines (CUBLAS) and fast Fourier transform (CUFFT) libraries to leverage GPUs' capabilities. These libraries release developers from rebuilding the frequently used basic operations such as matrix multiplications. Graphics cards from G8x series support the CUDA programming mode; and the latest generation of NVIDIA GPUs (GT2x0 series or later) unifies vertex and fragment processors and provides shared memory for interprocessor communications.

A increasing number of new GPU implementations with CUDA in different astrophysical simulations have been proposed. Belleman et al. [2] re-implemented the direct gravitational N-body simulations on GPUs using CUDA. For $N \gtrsim 10^5$, they reported a speedup of about 100 compared to the host CPU and about the same speed as the GRAPE-6Af. A library `Sapporo` for performing high precision gravitational N-body simulations was developed on GPUs by Gaburov et al. [13]. This library achieved twice as fast as commonly used GRAPE6A/GRAPE6-BLX cards. Stantchev et al. [46,47] implemented a Particle-In Cell (PIC) code on GPUs for plasmas simulations and visualizations and demonstrated a speedup of 11–22 for different grid sizes. Sainio [39] presented an accelerated GPU cosmological lattice program for solving the evolution of interacting scalar fields in an expanding universe, achieving speedups between one and two orders of magnitude in single precision. In the aforementioned work, no discussion on using double precision on GPUs was reported. In MHD simulations, the support of double precision is important, especially for nonlinear problems. We will evaluate the performance and accuracy of double precision on GPUs in this work.

In this paper, we present an efficient implementation to accelerate the computation of MHD simulations on GPUs, called *GPU-MHD*. To the best of our knowledge, this is the first report describing MHD simulations on GPUs in detail. The goal of our work is to perform a pilot study on numerically solving the ideal MHD equations on GPUs. In addition, the trend of today's chip design is moving to streaming and massively parallel processor models, developing new MHD codes to exploit such architecture is essential. *GPU-MHD* can be easily ported to other many-core platforms such as Intel's upcoming Larrabee [43], making it more flexible for the user's choice of hardware. This paper is organized as follows: A brief description of the CUDA programming model is given in Section 2. The numerical scheme in which *GPU-MHD* adopted is presented in Section 3. In Section 4, we present the GPU implementation in detail. Numerical tests are given in Section 5. Accuracy evaluation by comparing single and double precision computation results is given in Section 6. Performance measurements are reported in Section 7 and visualization of the simulation results is described in Section 8. We conclude our work and indicate some directions for possible future work in Section 9.

2. A brief description of the CUDA

The Compute Unified Device Architecture (CUDA) was introduced by NVIDIA as a general purpose parallel computing architecture, which includes GPU hardware architecture as well as software components (CUDA compiler and the system drivers and libraries). The CUDA programming model [20,30,40] consists of functions, called *kernels*, which can be executed simultaneously by a large number of lightweight *threads* on the GPU. These threads are grouped into one-, two-, or three-dimensional *thread blocks*, which are further organized into one- or two-dimensional *grids*. Only threads in the same block can share data and synchronize with each other during execution. Thread blocks are independent of each other and can be executed in any other. A graphics card that supports CUDA, for example, the GT200 GPU [27], consists of 30 streaming multiprocessors (SMs). Each multiprocessor consists of 8 streaming processors (SPs), providing a total of 240 SPs. Threads are grouped into batches of 32 called *warps* which are executed in single instruction multiple data (SIMD) fashion independently. Threads within a warp execute a common instruction at a time.

For memory access and usage, there are four types of memory, namely, *global memory*, *constant memory*, *texture memory* as well as *shared memory*. Global memory has a separate address space for obtaining data from the host CPU's main memory through the PCIE bus, which is about 8 GB/sec in the GT200 GPU. Any valued stored in the global memory can be accessed by all SMs via load and store instructions. Constant memory and texture memory are cached, read-only and shared between SPs. Constants that are kept unchanged during kernel execution may be stored in constant memory. Built-in linear interpolation is available in texture memory. Shared memory is limited (16 KB for GT200 GPU) and shared among all SPs in an MP. For detailed information concerning memory optimizations, we refer the reader to "CUDA Best Practice Guide" [32].

Double precision is one important concern in many computational physics applications, however, support of double precision is limited to the NVIDIA cards having Compute Capability 1.3 (see Appendix A in [30]) such as the GTX 260, GTX 280, Quadro FX 5800 (contains one GT200 GPU), and Tesla C1060 (contains one GT200 GPU) and S1070 (contains four GT200 GPUs). In GT200 GPU, there are eight single precision floating point (FP32) arithmetic logic units (ALUs) (one per SP) in SM, but only one double precision floating point (FP64) ALU (shared by eight SPs). The theoretical peak performance of GT200 GPU is 936 GFLOPS in single precision and 78 GFLOPS in double precision. In CUDA, double precision is disabled by default, ensuring that all double numbers are silently converted into float numbers inside kernels and any double precision calculations computed are incorrect. In order to use double precision floating point numbers, we need to call `nvcc: "-arch = sm_13"`. The flag "`-arch = sm_13`" in the command tells "`nvcc`" to use the Compute Capability 1.3 which means enabling the double precision support. The recent Fermi architecture [32] (GTX 480, for example) significantly improves the performance of double precision calculations by introducing better memory access mechanisms.

In Sections 6 and 7 we will compare the accuracy and actual performance of *GPU-MHD* in single and double precision on both GT200 and Fermi architectures.

3. Numerical scheme

The ideal MHD equations with the assumption of the magnetic permeability $\mu = 1$ can be represented as hyperbolic system of conservation laws as follows [14]

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho \boldsymbol{v}) = 0 \tag{1}$$

$$\frac{\partial \rho \boldsymbol{v}}{\partial t} + \nabla\cdot(\rho \boldsymbol{v}\boldsymbol{v} - \boldsymbol{B}\boldsymbol{B}) + \nabla P^* = 0 \tag{2}$$

$$\frac{\partial \boldsymbol{B}}{\partial t} - \nabla\times(\boldsymbol{v}\times\boldsymbol{B}) = 0 \tag{3}$$

$$\frac{\partial E}{\partial t} + \nabla\cdot\left(\left(E + P^*\right)\boldsymbol{v} - \boldsymbol{B}(\boldsymbol{B}\cdot\boldsymbol{v})\right) = 0 \tag{4}$$

Here, ρ is the mass density, $\rho\boldsymbol{v}$ the momentum density, $\boldsymbol{B}$ the magnetic field, and E the total energy density. The total pressure $P^* \equiv P + \frac{B^2}{2}$ where P is the gas pressure that satisfies the equation

of state, $P \equiv (\gamma-1)(E-\rho\frac{v^2}{2}-\frac{B^2}{2})$. In addition, the MHD equations should obey the divergence-free constraint $\nabla \cdot \boldsymbol{B} = 0$.

Over the last few decades, there has been a dramatic increase in the number of publications on the numerical solution of ideal MHD equations. In particular the development of shock-capturing numerical methods for ideal MHD equations. We do not provide an exhaustive review of the literature here. A comprehensive treatment of numerical solution of MHD equations can be found in [21], for example. Pen et al. [35] proposed a free, fast, simple, and efficient total variation diminishing (TVD) MHD code featuring modern high-resolution shock capturing on a regular Cartesian grid. This code is second-order accuracy in space and time and enforces the $\nabla \cdot \boldsymbol{B} = 0$ constraint to machine precision and it was successfully used for studying nonradiative accretion onto the supermassive black hole [36] and fast magnetic reconnection [34]. Due to these advantages and convenience for GPU versus CPU comparison, the underlying numerical scheme in *GPU-MHD* is based on this work. A detailed comparison of shock capturing MHD codes can be found in [52], for example. We plan to explore other recent high-order Godunov schemes such as [23] and [50] for *GPU-MHD* as our planned future work.

We briefly review the numerical scheme [35] we adopted in *GPU-MHD* here. In this numerical scheme, the magnetic field is held fixed first and then the fluid variables are updated. A reverse procedure is then performed to complete a one time step. The three-dimensional problem is split into one-dimensional subproblems by using a Strang-type directional splitting [51].

Firstly, we describe the fluid update step in which the fluid variables are updated while holding the magnetic field fixed. The magnetic field is interpolated to cell centers for second-order accuracy. By considering the advection along the x direction, the ideal MHD equations can be written in flux-conservative vector form as follows

$$\frac{\partial \boldsymbol{u}}{\partial t} + \frac{\partial \boldsymbol{F}(u)}{\partial x} = 0 \tag{5}$$

where the flux vector is given by

$$\boldsymbol{F} = \begin{pmatrix} \rho v_x \\ \rho v_x^2 + P^* - B_x^2 \\ \rho v_x v_y - B_x B_y \\ \rho v_x v_z - B_x B_z \\ (E+P^*)v_x - B_x \mathbf{B}\cdot\mathbf{v} \end{pmatrix} \tag{6}$$

Eq. (5) is then solved by Jin and Xin's relaxing TVD method [18]. With this method, a new variable $\boldsymbol{w} = \boldsymbol{F}(\boldsymbol{u})/c$ is defined, where $c(x,t)$ is a free positive function called the *flux freezing speed*. For ideal MHD equations, we have $\boldsymbol{u} = (u_1,u_2,u_3,u_4,u_5) = (\rho, \rho v_x, \rho v_y, \rho v_z, E)$ and equations

$$\frac{\partial \boldsymbol{u}}{\partial t} + \frac{\partial}{\partial x}(c\boldsymbol{w}) = 0 \tag{7}$$

$$\frac{\partial \boldsymbol{w}}{\partial t} + \frac{\partial}{\partial x}(c\boldsymbol{u}) = 0 \tag{8}$$

These equations can be decoupled through a change of left- and right-moving variables $\boldsymbol{u}^R = (\boldsymbol{u}+\boldsymbol{w})/2$ and $\boldsymbol{u}^L = (\boldsymbol{u}-\boldsymbol{w})/2$

$$\frac{\partial \boldsymbol{u}^R}{\partial t} + \frac{\partial}{\partial x}(c\boldsymbol{u}^R) = 0 \tag{9}$$

$$\frac{\partial \boldsymbol{u}^L}{\partial t} - \frac{\partial}{\partial x}(c\boldsymbol{u}^L) = 0 \tag{10}$$

The above pair of equations is then solved by an upwind scheme, separately for right- and left-moving waves, using cell-centered fluxes. Second-order spatial accuracy is achieved by interpolating of fluxes onto cell boundaries using a monotone upwind schemes for conservation laws (MUSCL) [56] with the help of the flux limiter. Runge–Kutta scheme is used to achieve second-order accuracy of time integration.

We denote $\boldsymbol{u}_n^t$ as the cell-centered values of the cell n at time t, $\boldsymbol{F}_n^t$ as the cell-centered flux in cell n. As an example, we consider the positive advection velocity, negative direction can be obtained in a similar way. We obtain the first-order upwind flux $F_{n+1/2}^{(1),t}$ from the averaged flux F_n^t in cell n. Two second-order flux corrections can be defined using three local cell-centered fluxes as follows

$$\Delta \boldsymbol{F}_{n+1/2}^{L,t} = \frac{\boldsymbol{F}_n^t - \boldsymbol{F}_{n-1}^t}{2} \tag{11}$$

$$\Delta \boldsymbol{F}_{n+1/2}^{R,t} = \frac{\boldsymbol{F}_{n+1}^t - \boldsymbol{F}_n^t}{2} \tag{12}$$

When the corrections have opposite signs,there is no second-order correction in the case of near extrema. With the aid of a flux limiter ϕ we then obtain the second-order correction

$$\Delta \boldsymbol{F}_{n+1/2}^t = \phi\left(\Delta \boldsymbol{F}_{n+1/2}^{L,t}, \Delta \boldsymbol{F}_{n+1/2}^{R,t}\right) \tag{13}$$

The van Leer limiter [55]

$$\text{vanleer}(a,b) = \frac{2ab}{a+b} \tag{14}$$

is used in *GPU-MHD*. By adding the second-order correction to the first-order fluxes we obtain second-order fluxes. For example, the second-order accurate right-moving flux $\boldsymbol{F}_{n+1/2}^{R,t}$ can be calculated

$$\boldsymbol{F}_{n+1/2}^{R,t} = \boldsymbol{F}_n^t + \Delta \boldsymbol{F}_{n+1/2}^t \tag{15}$$

The time integration is performed by calculating the fluxes $\boldsymbol{F}(u_n^t)$ and the freezing speed c_n^t in the first half time step is given as follows

$$\boldsymbol{u}_n^{t+\Delta t/2} = \boldsymbol{u}_n^t - \left(\frac{\boldsymbol{F}_{n+1/2}^t - \boldsymbol{F}_{n-1/2}^t}{\Delta x}\right)\frac{\Delta t}{2} \tag{16}$$

where $\boldsymbol{F}_{n+1/2}^t = \boldsymbol{F}_{n+1/2}^{R,t} - \boldsymbol{F}_{n+1/2}^{L,t}$ is computed by the first-order upwind scheme. By using the second-order TVD scheme on $\boldsymbol{u}_n^{t+\Delta t/2}$, we obtain the full time step $\boldsymbol{u}_n^{t+\Delta t}$

$$\boldsymbol{u}_n^{t+\Delta t} = \boldsymbol{u}_n^t - \left(\frac{\boldsymbol{F}_{n+1/2}^{t+\Delta t/2} - \boldsymbol{F}_{n-1/2}^{t+\Delta t/2}}{\Delta x}\right)\Delta t \tag{17}$$

To keep the TVD condition, the flux freezing speed c is the maximum speed information can travel and should be set to $|v_x| + (\gamma p/\rho + B^2/\rho)^{1/2}$ as the maximum speed of the fast MHD wave over all directions is chosen. As the time integration is implemented using a second-order Runge–Kutta scheme, the time step is determined by satisfying the CFL condition

$$\begin{aligned} c_{\max} &= \left[\max\left(|v_x|,|v_y|,|v_z|\right) + \left(\gamma p/\rho + B^2/\rho\right)^{1/2}\right] \\ \Delta t &= cfl/c_{\max} \end{aligned} \tag{18}$$

where cfl is the Courant number and $cfl \lesssim 1$ is generally set to $cfl \simeq 0.7$ for stability, and B is the magnitude of the magnetic field. Constrained transport (CT) [10] is used to keep the $\nabla \cdot \boldsymbol{B} = 0$ to machine precision. Therefore, the magnetic field is defined on cell faces and it is represented in arrays [35]

$$\begin{aligned} Bx(i,j,k) &= (B_x)_{i-1/2,j,k} \\ By(i,j,k) &= (B_y)_{i,j-1/2,k} \\ Bz(i,j,k) &= (B_z)_{i,j,k-1/2} \end{aligned} \tag{19}$$

where the cell centers are denoted by $(i, j, k) \equiv (x_i, y_j, z_k)$, and faces by $(i \pm 1/2, j, k)$, $(i, j \pm 1/2, k)$, and $(i, j, k \pm 1/2)$, etc. The cells have unit width for convenience.

Secondly, we describe the update of the magnetic field in separate two-dimensional advection-constraint steps along x-direction while holding the fluid variables fixed. The magnetic field updates along y- and z-directions can be handled in a similar matter. We follow the expressions used in [19]. For example, we can calculate the average of the v along x-direction as follows

$$(v_x)_{i,j+1/2,k} = \frac{1}{4}\left[(v_x)_{i+1,j+1/2,k} + 2(v_x)_{i,j+1/2,k} + (v_x)_{i-1,j+1/2,k}\right] \quad (20)$$

A first-order accurate flux is then obtained by

$$(v_x B_y)_{i+1/2,j+1/2,k} = \begin{cases} (v_x B_y)_{i,j+1/2,k}, & (v_x)_{i+1/2,j+1/2,k} > 0 \\ (v_x B_y)_{i+1,j+1/2,k}, & (v_x)_{i+1/2,j+1/2,k} \leqslant 0 \end{cases} \quad (21)$$

where the velocity average is

$$(v_x)_{i+1/2,j+1/2,k} = \frac{1}{2}\left[(v_x)_{i,j+1/2,k} + (v_x)_{i+1,j+1/2,k}\right] \quad (22)$$

B_x is updated by constructing a second-order accurate upwind electromotive force (EMF) $v_y B_x$ using Jin and Xin's relaxing TVD method [18] in the advection step. Then this same EMF is immediately used to update B_y in the constraint step.

Extension to three dimensions can be performed through a Strang-type directional splitting [51]. Eq. (5) is dimensionally split into three separate one-dimensional equations. For a time step Δt, let $fluid_x$ be the fluid update along x, $B_{x\to y}$ be the update of B_x along y, and L_i be the update operator of $\boldsymbol{u}^t$ to $\boldsymbol{u}^{t+\Delta t}$ by including the flux along i direction. Each L_i includes three update operations in sequence, for example, L_x includes $fluid_x$, $B_{y\to x}$, and $B_{z\to x}$. A forward sweep and a reverse sweep are defined as $\boldsymbol{u}^{t+\Delta t} = L_z L_y L_x \boldsymbol{u}^t$ and $\boldsymbol{u}^{t+2\Delta t} = L_x L_y L_z \boldsymbol{u}^{t+\Delta t}$, respectively. A complete update combines a forward sweep and reverse sweep. The dimensional splitting of the relaxing TVD can be expressed as follows [54]

$$\boldsymbol{u}^{t_2} = \boldsymbol{u}^{t_1+2\Delta t_1} = L_x L_y L_z L_z L_y L_x \boldsymbol{u}^{t_1} \quad (23)$$

$$\boldsymbol{u}^{t_3} = \boldsymbol{u}^{t_2+2\Delta t_2} = L_z L_x L_y L_y L_x L_z \boldsymbol{u}^{t_2} \quad (24)$$

$$\boldsymbol{u}^{t_4} = \boldsymbol{u}^{t_3+2\Delta t_3} = L_y L_z L_x L_x L_z L_y \boldsymbol{u}^{t_3} \quad (25)$$

where Δt_1, Δt_2, and Δt_3 are sequential time steps after each double sweep. For Cartesian coordinate system, it is easy to apply Strang-type directional splitting [51] on a high-dimensional problem and split it into one-dimensional sub-problems in Cartesian coordinate system [25]. In principle, we can also apply directional splitting for cylindrical or spherical coordinate systems. We may need to split the edges of grid in any direction into equal-distance pieces and determine the positions of the cell centers and face centers. Similar techniques from Li and Li [26] can be utilized to extend the usage of directional splitting for cylindrical or spherical coordinate systems. This extension will be left as possible future work.

4. GPU implementation

In this section, we provide the implementation details of *GPU-MHD*. With *GPU-MHD*, all computations are performed entirely on GPUs and all data is stored in the GRAM of the graphics card. Currently, *GPU-MHD* works on a regular Cartesian grid and supports both single and double precision modes. Considering the rapid development of graphics hardware, our GPU implementation was design in general for the GT200 architecture (GTX 295 in our study) and the Fermi architecture (GTX 480 in our study). Therefore, *GPU-MHD* can be used on newer architectures without significant modification.

Before we explain our GPU implementation in detail, the consideration and strategy of our design is presented first. During the computational process, the TVD numerical scheme for solving the MHD equations will generate many intermediate results such as the "flux" and some interpolated values of each grid point. These intermediate results will then be used in the next calculation step. One important aspect is that both these intermediate results of the current grid point and also those of the neighboring grid points are needed to be stored. This means the intermediate results of the neighboring grid points have to be calculated before going to the next calculation step. As a result, each calculation step in the algorithm was designed with one or several kernels and huge amount of data should be stored. In order to avoid the data transmission between CPU and GPU during the computation, *GPU-MHD* was designed to be run entirely on GPUs. To reduce the memory usage, the storage for the intermediate results will be reused to store the intermediate results generated by the next step. The eight components $(u_1, u_2, u_3, u_4, u_5, B_x, B_y, B_z)$ for solving the MHD equations are stored in the corresponding eight arrays. Each component of a grid point is stored close to the same component of the neighboring gird points. In any calculation step, only the necessary component of a calculation (kernel) will be accessed, thus providing more effective input/output access. The strategy of our design is summarized as follows:

- Each step of the numerical scheme is handled with one or several kernels to exploit the parallelism of GPUs;
- Storage of the intermediate results are reused to reduce memory usage;
- Components of the MHD equations are stored in separate arrays to provide effective memory access.

4.1. Memory arrangement

Although shared memory provides much faster access rate than global memory, its size is very limited (16 kB in GTX 295 and 48 kB in GTX 480). As we have to process many intermediate results in each calculation step, shared memory is too small to fit in our GPU implementation. Of course there are some techniques of using shared memory, the basic idea is to copy the data from global memory to the shared memory first, and then use the data in shared memory to do the calculations. After the calculations have been completed, write these results back to global memory. This will benefit those computations that need many data accesses during the calculation period. However, as we mentioned in the beginning of this section, due to the nature of the algorithm, *GPU-MHD* was designed with many separated CUDA kernels. Calculation of each kernel is actually simple and variables of grid points in each kernel are mostly accessed only once (read) or twice (read and then write the result). In order to provide fast access speed, parameters and temporary results (generated and used only within kernel) in each kernel are stored with registers. The parameters for the whole simulation such as the data size and size of dimensions are stored using constant memory. Thus in our case, the shared memory does not show its advantages. On the other hand, the size of the shared memory is too small for our problem, especially when double precision is used in the calculations. We did try to use the shared memory in *GPU-MHD* by copying the amount of data that the shared memory is capable to store to the shared memory for the calculations, but there is no speedup compared to our current approach. Therefore, our code mainly uses global

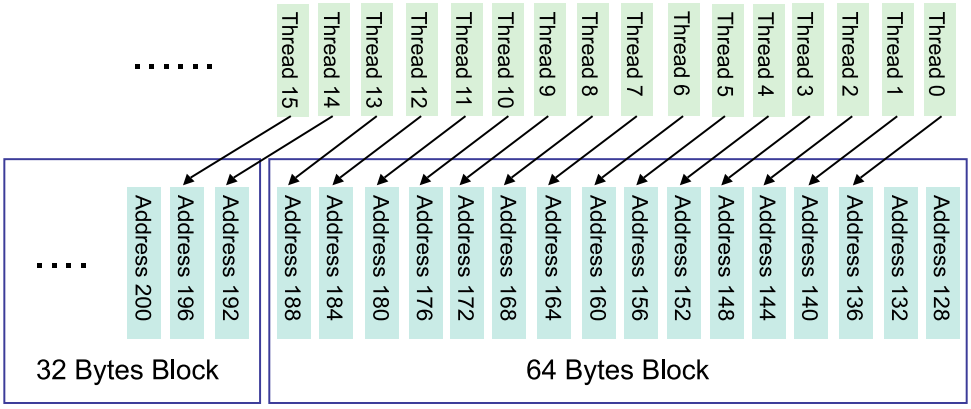

Fig. 1. An example demonstrating the automatic coalescing in the GT200 architecture.

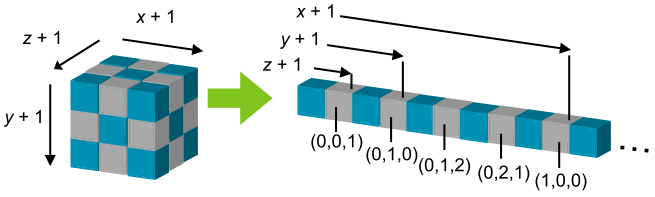

Fig. 2. Mapping from 3D array to 1D array in column major.

memory. There are three phases in our code: transfer of data from the host memory into the global memory, execution of the kernels, and transfer of data from the GPU into the host memory.

For global memory, if the data is well organized in the global memory with the form that a load statement in all threads in a warp accesses data in the same aligned 128-byte block, then the threads can efficiently access data from the global memory. The process of organizing the data in such a form is so-called coalescing [31,20]. Actually, the GT200 architecture (with Compute Capability 1.2 or 1.3) has more flexibility in handling data in the global memory than those cards with Compute Capability 1.1 or lower. Coalescing of loading and storing data that are not aligned perfectly to 128-byte boundaries is handled automatically on this architecture (see Appendix G.3.2.2 in [31]). We illustrate this new feature in Fig. 1. The GT200 architecture supports 32 bytes memory block and has less limitation to memory address, which is accessed by the header (first) thread. Even without "shifting" the address to aligned 64 bytes or 128 bytes, the GPU kernels can still keep good performance, especially when we only process with 2^n data.

The memory arrangement of *GPU-MHD* is presented here. The most intuitive way to write a parallel program to solve a multi-dimensional problem is to use multidimensional arrays for data storage and multidimensional threads for the computation. However, the ability of the current CUDA is limited in supporting multidimensional threads, therefore, we could not implement our code in such a straightforward way. Especially in three dimensions or higher dimensions, there are still some limitations in handling multidimensional arrays and multidimensional threads. As a result, the most primitive way is to store the data in one-dimensional arrays and perform the parallel computation with one-dimensional threads. By using an indexing technique, our storage and threading method can be extended to solve multidimensional problems. Our data storage arrangement is expressed in Fig. 2 and in Eqs. (26) to (28).

$$\begin{cases} INDEX_x = index/(SIZE_y \times SIZE_z) \\ INDEX_y = [index \ \mathbf{mod} \ (SIZE_y \times SIZE_z)]/SIZE_z \\ INDEX_z = index \ \mathbf{mod} \ SIZE_z \end{cases} \tag{26}$$

$$INDEX_x \pm 1 = index \pm (SIZE_y \times SIZE_z) \tag{27}$$

$$INDEX_y \pm 1 = index \pm SIZE_z \tag{28}$$

$$INDEX_z \pm 1 = index \pm 1 \tag{29}$$

Here $INDEX_x$, $INDEX_y$, and $INDEX_z$ are the indexes of a 3D matrix. *index* is the 1D index used in *GPU-MHD*, $SIZE_y$, and $SIZE_z$ are the matrix size (number of grid points in our study) of a 3D matrix.

Eq. (26) expresses the mapping of three-dimensional (3D) indexes to one-dimensional (1D) indexes. Eqs. (28) to (29) express the shift operations. Shift operations are very important in numerical solution of conservation laws because some calculations are based on the neighboring grid points. The above indexing technique is used to prepare suitable values (vectors) as input values for the calculation kernels we implemented in CUDA. As an example, we give a conceptual calculation kernel for a calculation in the *x*-dimension to show how the indexing technique works for this task in the following. This kernel calculates the result with the grid point itself and neighboring gird points in the *x*-dimension. The calculations in the *y*- or the *z*-dimension have a similar form.

```
Calculate_X(data, result) {
index = getID();
        //self-increment index for multi-threading
grid_point_index = index; //(x, y, z)
neighbor_1 = grid_point_index + (SIZEy * SIZEz);
             //(x + 1, y, z)
neighbor_2 = grid_point_index - (SIZEy * SIZEz);
             //(x - 1, y, z)

calculate_kernel(data, result, grid_point_index,
                 neighbor_1, neighbor_2, ...);
......
}
```

The indexing technique is a common way to represent multidimensional arrays using 1D arrays by mapping a 3D index (x, y, z) to a 1D index $(x \times Y_{size} \times Z_{size} + y \times Z_{size} + z)$. The GPU kernels of TVD were designed such that each kernel calculates using the actual index of a particular grid point and its neighbors. For example, if the calculation needs the information in a particular gird point and its neighboring grid points in the *z*-dimension, then the indexing operation will retrieve $[x \times Y_{size} \times Z_{size} + y \times Z_{size} + (z-1)]$, $[x \times Y_{size} \times Z_{size} + y \times Z_{size} + z]$ and $[z \times Y_{size} \times Z_{size} + y \times Z_{size} + (z+1)]$ and proceed with the calculation. If the calculation needs the information in a particular grid point and its neighboring grid points in the *y*-dimension, then the indexing operation will retrieve $[x \times Y_{size} \times Z_{size} + (y-1) \times Z_{size} + z]$, $[x \times Y_{size} \times Z_{size} + y \times Z_{size} + z]$ and $[x \times Y_{size} \times Z_{size} + (y+1) \times Z_{size} + z]$. Then these resulting indexes from indexing operation will pass to the GPU kernels of

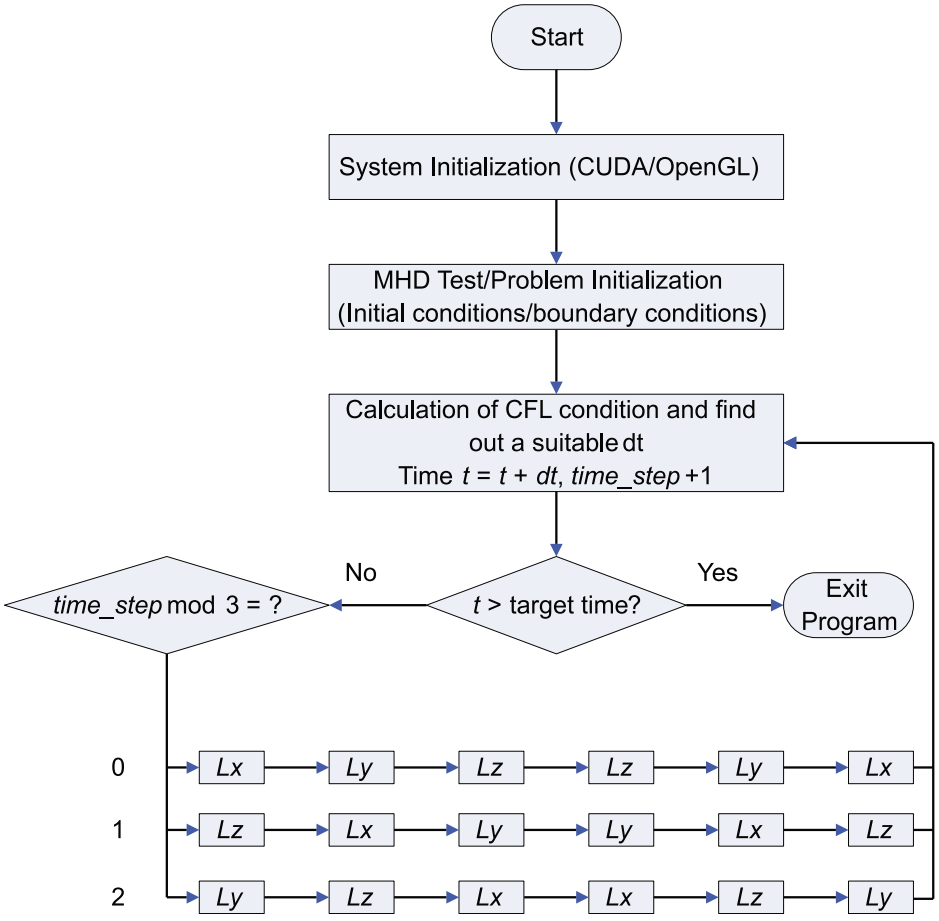

Fig. 3. The flow chart of *GPU-MHD*.

TVD for performing the calculation. As a result, for N-dimension problems, what we need are the N-dimension indexing operation kernels while only one TVD kernel is needed at all the time.

For efficiency, *GPU-MHD* only processes the problem with the number of grid points satisfying the 2^n condition. One reason is that the size of a warp of GPU contains 32 threads, problems with grid point number of 2^n are easier to determine the number of threads and blocks to fit in multiple of a warp before the GPU kernel is called. That means we do not need to check if the ID of the grid point being processed (calculated by the block ID and thread ID) is out of the range. It is very helpful in making the GPU code run more efficiently. On the other hand, it is also effective to reduce logical operations in a GPU kernel, which is known to be a little bit slow in the current GPU architecture. As a result, warp divergence caused by the number of the data is avoided (there is still a little bit warp divergence caused by the "if" operation in the calculation of our algorithm). A similar method is used in the CUDA SDK code sample "reduction".

The actual memory pattern used in *GPU-MHD* will be presented at the end of next subsection after introducing our algorithm.

4.2. Program flow

A "CUDA kernel" is a function running on GPU [20,30,40]. Noted that the CUDA kernel will process all grid points in parallel, therefore, a `For` instruction is not needed for going through all grid points. *GPU-MHD* includes the following steps:

(1) CUDA initialization
(2) Setup the initialize condition for the specified MHD problem: $\boldsymbol{u} = (u_1, u_2, u_3, u_4, u_5)$ of all grid points, $\boldsymbol{B} = (B_x, B_y, B_z)$ of cell faces, and set parameters such as time t, etc.
(3) Copy the initialize condition $\boldsymbol{u}$, $\boldsymbol{B}$ to device memory (CUDA global memory)
(4) For all grid points, calculate the $c_{\max}$ by Eq. (18) (implemented with a CUDA kernel)
(5) Use `cublasIsamax` (in single precision mode) function or `cublasIdamax` (in double precision mode) function of the CUBLAS library to find out the maximum value of all $c_{\max}$, and then determine the Δt
(6) Since the value of Δt is stored in device memory, read it back to host memory (RAM)
(7) Sweeping operations of the relaxing TVD (calculation of the L_i, $i = x, y, z$, implemented with several CUDA kernels, will be explained in the next subsection)
(8) $t = t + 2\Delta t$
(9) `If` t reaches the target time, go to next step `else` repeat the procedure from step (4)
(10) Read back data $\boldsymbol{u}$, $\boldsymbol{B}$ to host memory
(11) Output the result

The program flow of *GPU-MHD* is shown in Fig. 3. After the calculation of the CFL condition, the sweeping operations will be performed. The sweeping operation L_i will update both the fluid variables and orthogonal magnetic fields along the i dimension. This is a core computation operation in the relaxing TVD scheme described in Section 3.

The CFL condition for the three-dimensional relaxing TVD scheme is obtained by Eq. (18). The procedure is to calculate all the $c_{\max}$ of each grid point and find out the maximum value. In *GPU-MHD*, the parallel computation power of CUDA is exploited to calculate the $c_{\max}$ of each grid point in parallel and all the $c_{\max}$ values are stored in a matrix. Then the `cublasIsamax` function

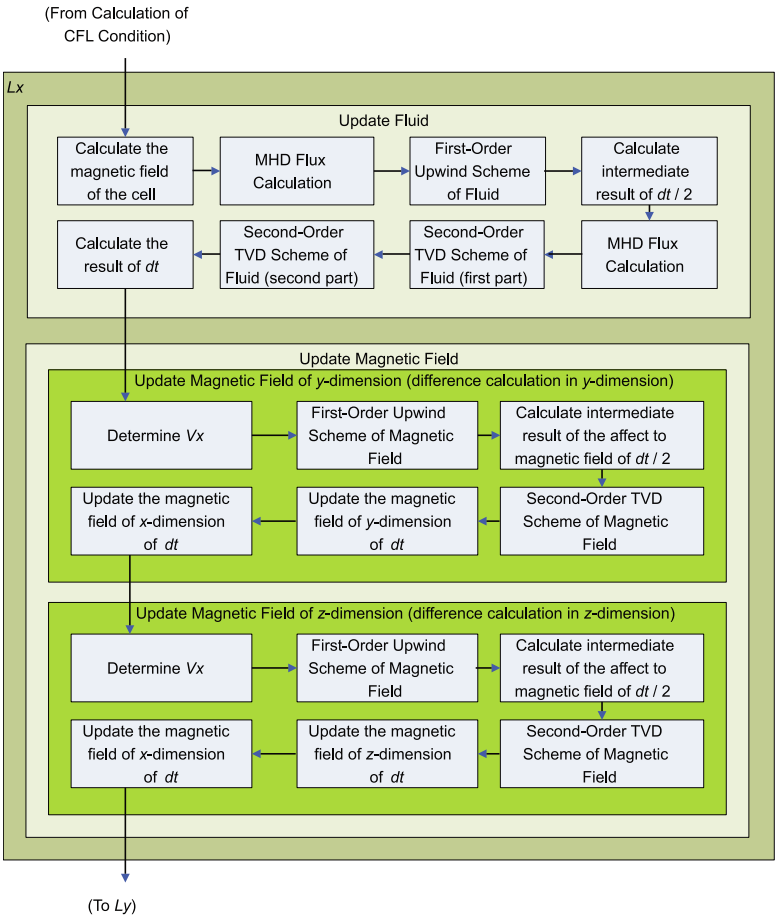

Fig. 4. Calculation process of L_x.

is used (in double precision mode, the `cublasIdamax` function is used) to find out the maximum $c_{\max}$ of the matrix in parallel (called the reduction operation). The `cublasIsamax` function is provided in the CUBLAS library — a set of basic operations for vector and matrix provided by NVIDIA with the CUDA toolkit [30]. The reason we read the Δt back and store both Δt and t in the host memory is due to the data in the device memory cannot be printed out directly in the current CUDA version. This information is useful for checking if there is any problem during the simulation. The implementation of sweeping operations will be explained in the next subsection.

4.3. Sweeping operations

Before we start to describe the sweeping operations, consideration of the memory arrangement is presented first in the following.

Implementing parallel computation using CUDA kernels is somewhat similar to parallel implementation on a CPU-cluster, but it is not the same. The major concern is the memory constrain in GPUs. CUDA makes parallel computation process on GPUs which can only access their graphics memory (GRAM). Therefore, data must be stored in GRAM in order to be accessed by GPUs. There are several kinds of memory on graphics hardware including registers, local memory, shared memory, and global memory, etc., and they have different characteristics and usages [30], making memory management of CUDA quite different compared to the parallel computation on a CPU-cluster. In addition, even though the size of the GRAM in a graphics card increases rapidly in newer models (for example, the latest NVIDIA graphics card — GeForce GTX 295 has 1.75G GRAM), not all the capacity of GRAM can be used to store data arbitrarily. Shared memory and local memory are flexible to use, however, their sizes are very limited in a block and thus they cannot be used for storing data with large size. In general, numerical solution of conservation laws will generate many intermediate results (for example, $\boldsymbol{u}^{t+\Delta t/2}$, $\boldsymbol{F}$, c, w, etc.) during the computation process, these results should be stored for subsequent steps in the process. Therefore, global memory was mainly used in *GPU-MHD*.

After the maximum value of $c_{\max}$ in Eq. (18) is found, we can obtain the Δt by determining the Courant number (*cfl*). The sequential step is the calculation of L_i $(i = x, y, z)$. The implementation of L_i includes two parts: update the fluid variables and update the orthogonal magnetic fields. As an example, the process for calculating L_x is shown in Fig. 4 where each block was implemented with one or several CUDA kernels. The process for calculating L_y or L_z is almost the same as L_x except that the dimensional indexes are different.

The first part of the L_x calculation process is $fluid_x$. The fluid variables will be updated along x. Algorithm 1 shows the steps and GPU kernels of this process (the data of $\boldsymbol{u}$ and $\boldsymbol{B}$ are already copied to device memory), all the steps are processed on all grid points with CUDA kernels in parallel.

In this process, we have to calculate the magnetic fields of the grid point (Eq. (19)) first because all the magnetic fields are defined on the faces of the grid cell [35]. To update the fluid variables of L_x, the main process, which includes one or even several CUDA kernels, is to calculate the affect of the orthogonal magnetic fields to the fluid variables of Eqs. (6), (9) and (10). One such main process gives the flux of the $\Delta t/2$ step. After two main processes of flux calculation and the other difference calculations, the value of the fluid — $\boldsymbol{u}$ is updated from $\boldsymbol{u}^t$ to $\boldsymbol{u}^{t+\Delta t}$ in one L_x process.

Algorithm 1 Algorithm of $fluid_x$, all equations and difference calculations are processed using CUDA kernels

1: load $\boldsymbol{u}$, $\boldsymbol{B}$ and Δt
2: memory allocation for the storage of the intermediate results: $\boldsymbol{B}_{temp}$, $\boldsymbol{u}_{temp}$, $\mathbf{flux}_{\mathbf{temp}}$, $\mathbf{other}_{\mathbf{temp}}$, ($\mathbf{other}_{\mathbf{temp}}$ includes the storage of $\boldsymbol{F}$, c, w, etc.)
3: $\boldsymbol{B}_{temp} \leftarrow$ results obtained by Eq. (19) with $\boldsymbol{B}$, ($\boldsymbol{B}$ stored the magnetic field of the cell faces)
4: $\mathbf{other}_{\mathbf{temp}} \leftarrow$ results obtained by Eqs. (6) and (9) with $\boldsymbol{u}$
5: $\mathbf{flux}_{\mathbf{temp}} \leftarrow$ the flux of a half time step: difference calculation ("First-Order Upwind Scheme of Fluid" CUDA kernels in Fig. 4) obtained by Eq. (16) using $\mathbf{other}_{\mathbf{temp}}$
6: $\boldsymbol{u}_{\mathbf{temp}} \leftarrow$ calculate the intermediate result ($\boldsymbol{u}^{t+\Delta t/2}$) using Eq. (16) with $\boldsymbol{u}$ and $\mathbf{flux}_{\mathbf{temp}}$
7: $\mathbf{other}_{\mathbf{temp}} \leftarrow$ results obtained by Eqs. (6) and (9) with $\boldsymbol{u}_{temp}$ (the same algorithm and same CUDA kernels in Step 4)
8: $\mathbf{flux}_{\mathbf{temp}} \leftarrow$ the flux of another half time step: difference calculation ("Second-Order TVD Scheme of Fluid" CUDA kernels in Fig. 4) obtained by Eq. (17) and the limiter (Eq. (14)) using $\mathbf{other}_{\mathbf{temp}}$
9: calculate the result of $\boldsymbol{u}^{t+\Delta t}$ with $\mathbf{flux}_{\mathbf{temp}}$ using Eq. (17) and save it back to $\boldsymbol{u}$
10: free the storage of the intermediate results
11: (continue to the second part of L_x, update the orthogonal magnetic fields)

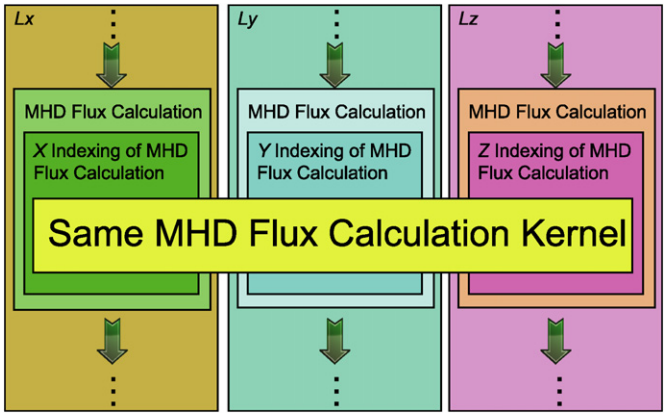

Fig. 5. Flux computation in *GPU-MHD*.

The second part of the L_x calculation process is to update the orthogonal magnetic fields in the y-dimension ($B_{y\to x}$), and the z-dimension ($B_{z\to x}$) with the fluid along the x-dimension. The strategy and implementation are similar to those in the first part but with a different algorithm for the orthogonal magnetic fields.

In Algorithm 1, the calculations in steps (4) to (9) are the steps for $B_{y\to x}$, and steps (11) to (16) are the steps for $B_{z\to x}$. The steps for $B_{y\to x}$ and $B_{z\to x}$ are almost the same, and the only different parts are the dimensional indexes of the difference calculations, and the affected magnetic fields: B_y and B_z. After the first part of L_x the fluid $\boldsymbol{u}^t$ is updated to $\boldsymbol{u}^{t+\Delta t}$. This change of the fluid affects the orthogonal magnetic fields. Therefore, the corresponding change (flux) of orthogonal magnetic fields can be calculated with the density and velocity of the updated fluid $\boldsymbol{u}^{t+\Delta t}$. Then the orthogonal magnetic fields are also updated to $\boldsymbol{B}_y^{t+\Delta t}$ and $\boldsymbol{B}_z^{t+\Delta t}$, and also, these changes give effects to $\boldsymbol{B}_x$.

After one process of L_x, both fluid and magnetic fields are updated to $t+\Delta t$ with the affect of the flow in the x-dimension. A sweeping operation sequence includes two L_x, L_y, and L_z (see Eq. (23)). So we actually obtain the updated fluid and magnetic fields of $t+2\Delta t$ after one sweeping operation sequence. Note that the second L_x in the sequence is a reverse sweeping operation, the order of $fluid_x$, $B_{y\to x}$ and $B_{z\to x}$ has to be reversed: $B_{y\to x}$ and $B_{z\to x}$ first, and $fluid_x$ second.

As we mentioned previously, numerical solution of conservation laws needs much memory storages because there are many intermediate results generated during the computation process. These intermediate results should be stored for the next calculation steps which need the information of the neighboring grid points obtained in the previous calculation steps. Otherwise, we have to perform many redundant processes in order to avoid the asynchronous problem in parallel computation. This is due to the processors on GPUs will not automatically start or stop working synchronously. Without storing the intermediate results, it will be hard to guarantee the values of the neighboring grid points updated synchronously. With the purpose to minimizing the memory usage, not only the calculation process of L_x is divided into several steps (CUDA kernels), but also the intermediate results are stored as little as possible. The processes dealing with the difference calculations are also divided into several steps to minimize the storage of the intermediate results and to guarantee there is no wrong result caused by asynchronous problem.

It should be realized that most of the processes in the three-dimensional relaxing TVD scheme with the dimensional splitting technique is similar. Pen et al. [35] swapped the data of x, y, and z-dimensions while *GPU-MHD* used one-dimensional arrays. But the similar swapping technique can be applied in our case with some indexing operations. Instead of transposing or swapping the data, we implemented each calculation part of the flux computation with two sets of CUDA kernels: one set is the CUDA kernels for calculating the relaxing TVD scheme (we call it TVD kernel here) and the other set is the CUDA kernels actually called by L_i operations (we call them L_i kernels here). Indexing operations are contained in all L_i kernels. After the index is calculated, TVD kernels are called and the indexes are passed to the TVD kernels, letting the TVD kernels calculate the flux of the corresponding dimension. Therefore, the difference among L_x, L_y, and L_z is the dimensional index. The flux computation of *GPU-MHD* is shown in Fig. 5.

The indexing operation swaps the target that will be updated and the neighboring relationship will also be changed accordingly. For example, the calculation that uses $x+1$ as the neighboring element in L_x will be changed to $y+1$ in L_y. As transposing the data in a matrix needs more processing time, it is efficient and flexible to extend the code to multidimensional case by dividing the indexing operation and flux calculation.

As we mentioned in Section 4.1, the data is stored in 1D array, the data accesses of L_x, L_y, and L_z are depicted in Fig. 6. In L_x, the data of (x,y,z), $(x,y+1,z)$, $(x,y-1,z)$, $(x,y,z+1)$, $(x,y,z-1)$ are used to calculate and update the data of (x,y,z). The data of $(x+1,y,z)$, $(x+1,y+1,z)$, $(x+1,y-1,z)$, $(x+1,y,z+1)$, $(x+1,y,z-1)$ are used to calculate and update the data of $(x+1,y,z)$, and so on. Similarly, in L_y, the data of (x,y,z), $(x+1,y,z)$, $(x-1,y,z)$, $(x,y,z+1)$, $(x,y,z-1)$ are used to calculate and update the data of (x,y,z). The data of $(x,y+1,z)$, $(x+1,y+1,z)$, $(x-1,y+1,z)$, $(x,y+1,z+1)$, $(x,y+1,z-1)$ are used to calculate and update the data of $(x,y+1,z)$, and so on. In L_z, the data of (x,y,z), $(x+1,y,z)$, $(x-1,y,z)$, $(x,y+1,z)$, $(x,y-1,z)$ are used to calculate and update the data of (x,y,z). The data of $(x,y,z+1)$, $(x+1,y,z+1)$, $(x-1,y,z+1)$, $(x,y+1,z+1)$, $(x,y-1,z+1)$ are used to calculate and update the data of $(x,y,z+1)$, and so on. It seems that the data accesses of L_x and L_y will slow down the performance since these accesses are not in so-called "coalescing" pattern. However, experimental results show that the computational times spending on calculating each dimensional component such as $fluid_x$ and $B_{y\to x}$ in L_x, L_y, and L_z are very close in our current arrangement (see Tables 4, 5, and 6 in Section 7). This is due to the fact that the GT200 and the Fermi GPU are more flexible to handle the data access that is not perfectly coalesced (see Section 4.1). Thus we did not further perform the coalescing to make these data accesses in optimal coalescing pattern.

Algorithm 2 Algorithm of ($B_{y\to x}$) and ($B_{z\to x}$), all equations and difference calculations are processed using CUDA kernels

(after the processes of fluid, we obtain an updated $\boldsymbol{u}$)
load u_1 (density ρ), u_2 (ρv_x), $\boldsymbol{B}$ and Δt
memory allocation for the intermediate results: $\boldsymbol{B}_{temp}$, $\mathbf{flux_{temp}}$, $\mathbf{vx_{temp}}$ and $\mathbf{vx_{face}}$
$\mathbf{vx_{temp}} \leftarrow$ determine the fluid speed with the updated u_1 and u_2 in $fluid_x$, with the difference calculated in the y-dimension
$\mathbf{vx_{face}} \leftarrow$ Results obtained by Eq. (20)
$\mathbf{flux_{temp}} \leftarrow$ the flux of a half time step: difference calculation of "flux of magnetic field in the y-dimension" ("First-Order Upwind Scheme of Magnetic Field" CUDA kernels in Fig. 4) obtained by Eqs. (16) and (21))
$\boldsymbol{B}_{temp} \leftarrow$ calculate the intermediate result ($\boldsymbol{u}^{t+\Delta t/2}$) by applying Eq. (16) to B_y (not by applying Eq. (16) to $\boldsymbol{u}$) with B_y and $\mathbf{flux_{temp}}$
$\mathbf{flux_{temp}} \leftarrow$ the flux of another half time step: difference calculation ("Second-Order TVD Scheme of Magnetic Field" CUDA kernels in Fig. 4) obtained by Eq. (16), the limiter of Eqs. (14) and (21)
calculate the result of $B_x^{t+\Delta t}$ and $B_z^{t+\Delta t}$ with $\mathbf{flux_{temp}}$ by applying Eq. (17) to B_y, and save it back to $\boldsymbol{B}$
(the following steps is similar to above steps but the affected orthogonal magnetic field is changed from y to z)
$\mathbf{vx_{temp}} \leftarrow$ determine the fluid speed with the updated u_1 and u_2 in $fluid_x$, with the difference calculated in the z-dimension
$\mathbf{vx_{face}} \leftarrow$ Results obtained with Eq. (20) using index of i, j, $k+1/2$
$\mathbf{flux_{temp}} \leftarrow$ the flux of a half time step: difference calculation of "flux of magnetic field in the z-dimension" ("First-Order Upwind Scheme of Magnetic Flied" CUDA kernels in Fig. 4) obtained by Eqs. (16) and (21)
$\mathbf{b_{temp}} \leftarrow$ calculate the intermediate result ($\mathbf{u}^{t+\Delta t/2}$) by applying Eq. (16) to B_z (not by applying Eq. (16) to $\boldsymbol{u}$) with B_z and $\mathbf{flux_{temp}}$
$\mathbf{flux_{temp}} \leftarrow$ the flux of another half time step: difference calculation ("Second-Order TVD Scheme of Magnetic Flied" CUDA kernels in Fig. 4) obtained by Eq. (16), the limiter of Eq. (14) and Equation (21)
calculate the results of $B_x^{t+\Delta t}$ and $B_z^{t+\Delta t}$ with $\mathbf{flux_{temp}}$ by applying Eq. (17) to B_z, and save it back to $\boldsymbol{B}$
free the storage of the intermediate results

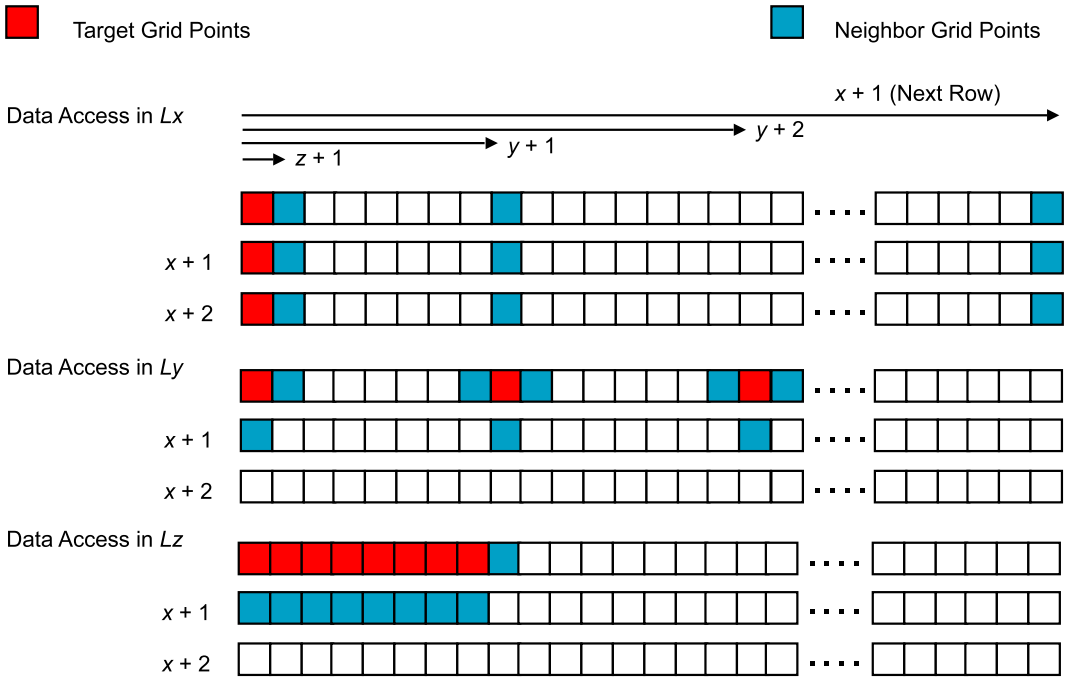

Fig. 6. The data accesses of L_x, L_y and L_z.

After the whole pipeline of Fig. 3 is completed, the MHD simulation results will be stored in GRAM and these results are readily to be further processed by the GPU for visualization or read back to the CPU for other usage. Due to the data-parallel nature of the algorithm and its high arithmetic intensity, we can expect our GPU implementation to exhibit a relatively good performance on GPUs.

5. Numerical tests

In this section, several numerical tests in one-dimensional (1D), two-dimensional (2D), and three-dimensional (3D) cases for validation of *GPU-MHD* are given. Two graphics cards NVIDIA GeForce GTX 295 and GTX 480 were used. GTX 295 has two GPUs inside but only one was used in these numerical tests. The results shown in this section are computed with a single precision mode in *GPU-MHD* on GTX 295. The difference between single precision and double precision computation results will be discussed in Section 6.

5.1. One-dimensional problems

5.1.1. Brio–Wu shock tube

1D Brio–Wu shock tube problem [3] which is an MHD version of the Sod problem [44], consisting of a shock tube with two initial equilibrium states as follows

Left side ($x < 0.5$)

$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{30}$$

$$\begin{Bmatrix} B_x \\ B_y \\ B_z \end{Bmatrix} = \begin{Bmatrix} 0.75 \\ 1 \\ 0 \end{Bmatrix} \tag{31}$$

$$\rho = 1, \qquad p = 1 \tag{32}$$

Right side ($x \geqslant 0.5$)

$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{33}$$

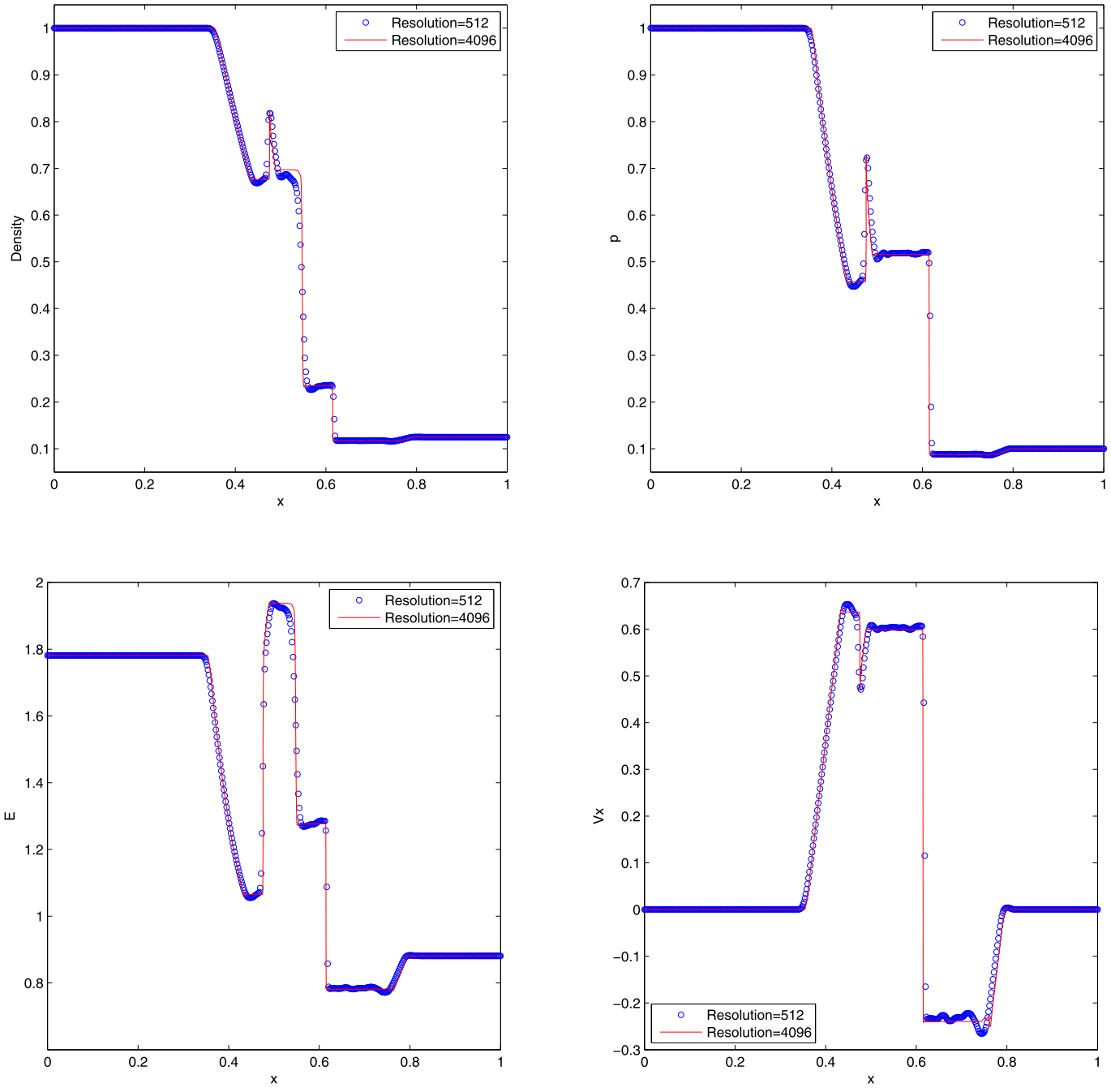

Fig. 7. Results (part I) of Brio–Wu shock tube problem at $t = 0.08L$. The result computed with 512 grid points is shown with circles and the solid line shows the reference high-resolution result of 4096 grid points.

$$\begin{Bmatrix} B_x \\ B_y \\ B_z \end{Bmatrix} = \begin{Bmatrix} 0.75 \\ -1 \\ 0 \end{Bmatrix} \tag{34}$$

$$\rho = 0.125, \qquad p = 0.1 \tag{35}$$

A constant value of $\gamma = 2$ was used and the problem was solved for $x \in [0, 1]$ with 512 grid points. Numerical results are presented at $t = 0.08L$ in Figs. 7 and 8, which include the density, the pressure, the energy, the y- and z-magnetic field components, and the x-, y- and z-velocity components. The results are in agreement with those obtained by Brio and Wu [3] and Zachary et al. [58].

5.1.2. MHD shock tube

The second 1D test is the MHD shock tube problem considered in [6].

Left side ($x < 0.5$)

$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{Bmatrix} 1.2 \\ 0.01 \\ 0.5 \end{Bmatrix} \tag{36}$$

$$\begin{Bmatrix} B_x \\ B_y \\ B_z \end{Bmatrix} = \begin{Bmatrix} 2\sqrt{(4\pi)} \\ 3.6/\sqrt{(4\pi)} \\ 2/\sqrt{(4\pi)} \end{Bmatrix} \tag{37}$$

$$\rho = 1.08, \qquad p = 0.95 \tag{38}$$

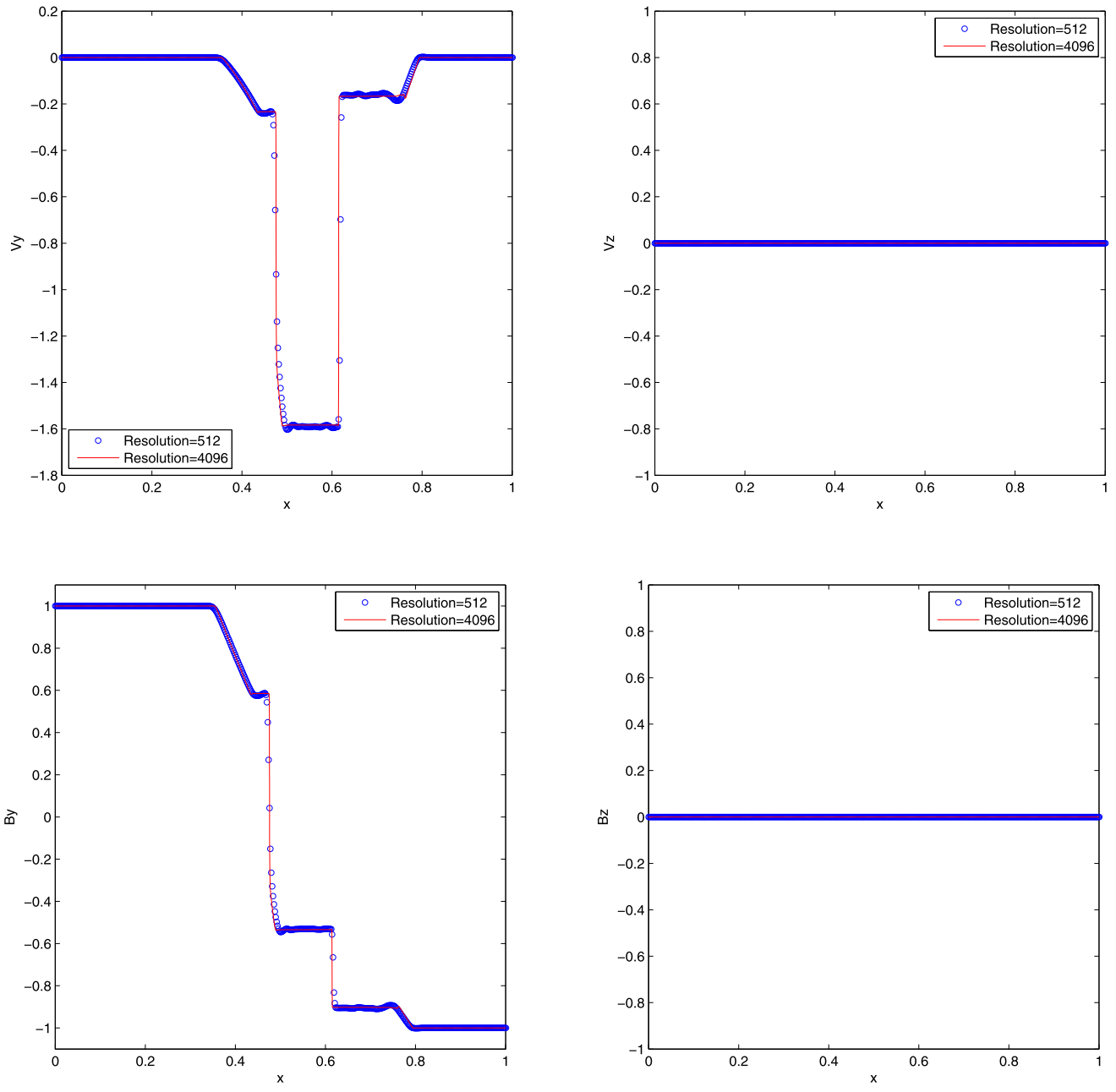

Fig. 8. Results (part II) of Brio–Wu shock tube problem at $t = 0.08L$. The result computed with 512 grid points is shown with circles and the solid line shows the reference high-resolution result of 4096 grid points.

Right side ($x \geqslant 0.5$)

$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{39}$$

$$\begin{Bmatrix} B_x \\ B_y \\ B_z \end{Bmatrix} = \begin{Bmatrix} 2\sqrt{(4\pi)} \\ 4/\sqrt{(4\pi)} \\ 2/\sqrt{(4\pi)} \end{Bmatrix} \tag{40}$$

$$\rho = 1, \qquad p = 1 \tag{41}$$

A constant value of $\gamma = 5/3$ was used and the problem was solved for $x \in [0, 1]$ with 512 grid points. Numerical results are presented at $t = 0.2L$ in Figs. 9 and 10, which include the density, the pressure, the energy, the y- and z-magnetic field components, and the x-, y- and z-velocity components. The results are in agreement with those obtained by [6] and [38].

5.2. Two-dimensional problems

5.2.1. Orszag–Tang problem

The first 2D test is Orszag–Tang problem [33], which is used to study incompressible MHD turbulence. In our test, the boundary conditions are periodic everywhere. The density ρ, pressure p, initial velocities (v_x, v_y, v_z), and magnetic field (B_x, B_y, B_z) are given by

$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{Bmatrix} -\sin(2\pi y) \\ \sin(2\pi x) \\ 0 \end{Bmatrix} \tag{42}$$

$$\begin{Bmatrix} B_x \\ B_y \\ B_z \end{Bmatrix} = \begin{Bmatrix} -B_0 \sin(2\pi y) \\ B_0 \sin(4\pi x) \\ 0 \end{Bmatrix} \quad \text{where } B_0 = 1/\sqrt{4\pi} \tag{43}$$

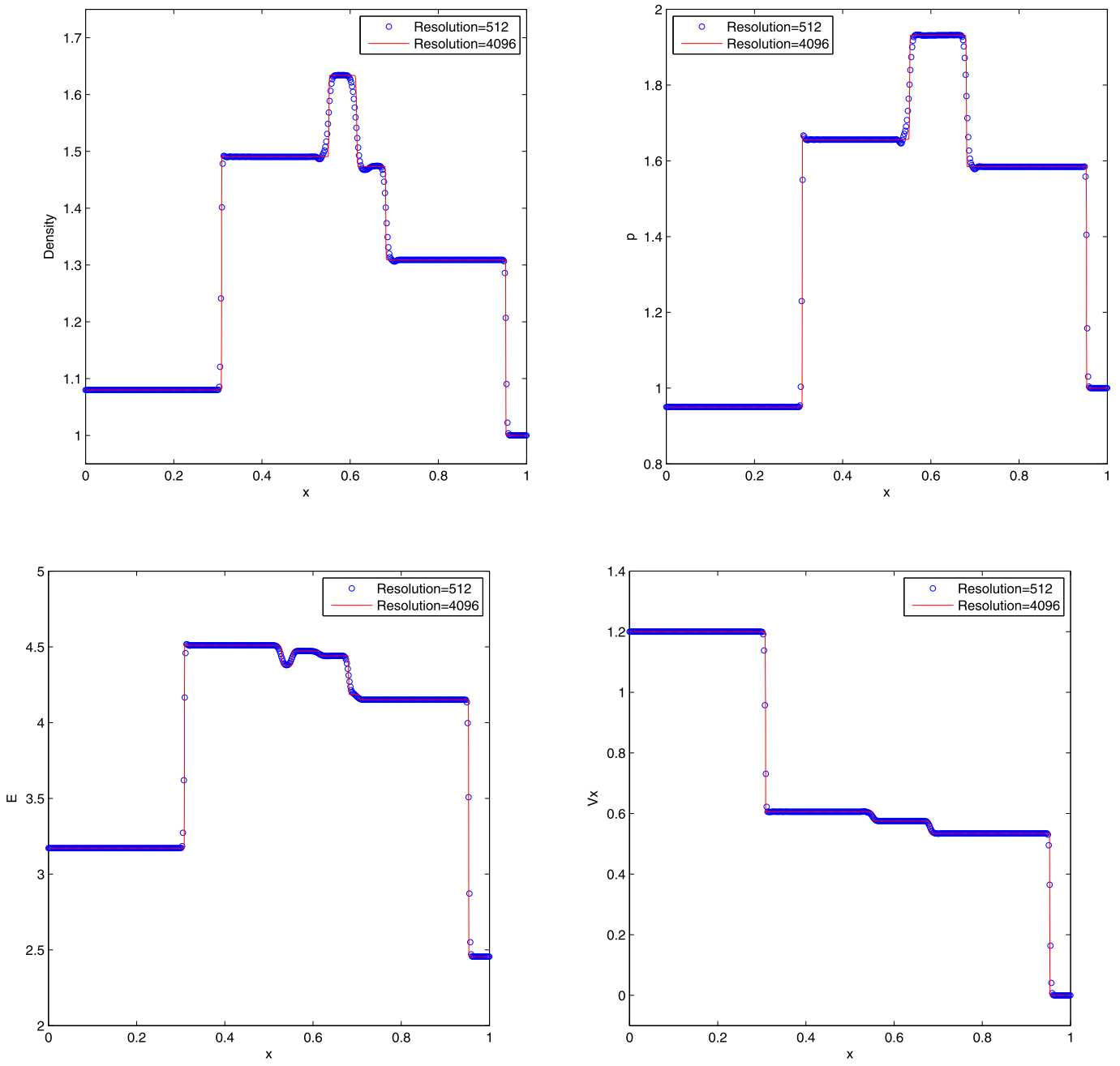

Fig. 9. Results (part I) of MHD shock tube test at $t = 0.2L$. The result computed with 512 grid points is shown with circles and the solid line shows the reference high-resolution result of 4096 grid points.

$$\rho = 25/(36\pi), \qquad p = 5/(12\pi)$$
$$\gamma = 5/3, \quad (0 \leqslant x \leqslant 1)\ (0 \leqslant y \leqslant 1) \tag{44}$$

The Orszag–Tang vertex test was performed in a two-dimensional periodic box with 512 × 512 grid points. The results of the density and gas pressure evolution of the Orszag–Tang problem at $t = 0.5L$ and $t = 1.0L$ are shown in Fig. 11, where the complex pattern of interacting waves is perfectly recovered. The results agree well with those in Lee et al. [23].

5.2.2. Two-dimensional blast wave problem

The second 2D test is the MHD blast wave problem. The MHD spherical blast wave problem of Zachary et al. [58] is initiated by an over pressured region in the center of the domain. The result is a strong outward moving spherical shock with rarefied fluid inside the sphere. We followed the test suite [45] of Athena [49]. The condition for 2D MHD blast wave problem is listed as follows [45]

$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{45}$$

$$\begin{Bmatrix} B_x \\ B_y \\ B_z \end{Bmatrix} = \begin{Bmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \\ 0 \end{Bmatrix} \tag{46}$$

$$p = \begin{cases} 10 & \text{inside the spherical region} \\ 0.1 & \text{outside the spherical region} \end{cases} \tag{47}$$

$$\rho = 1, \qquad p = 5/(12\pi), \qquad \gamma = 5/3$$

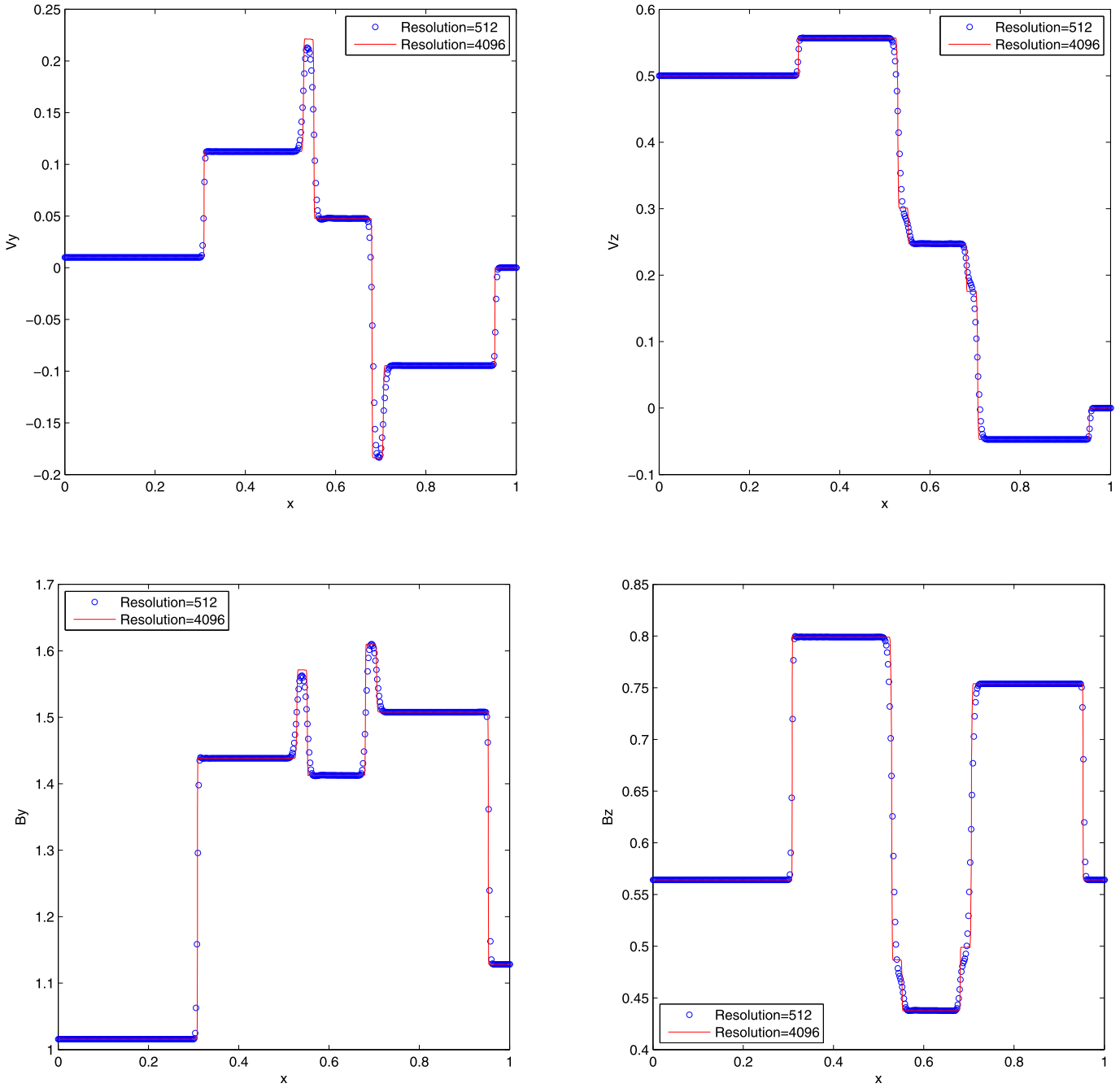

Fig. 10. Results (part II) of MHD shock tube test at $t = 0.2L$. The result computed with 512 grid points is shown with circles and the solid line shows the reference high-resolution result of 4096 grid points.

$$\text{spherical region center} = (0.5, 0.5), \qquad r = 0.1$$
$$(0 \leqslant x \leqslant 1)\ (0 \leqslant y \leqslant 1) \tag{48}$$

In Fig. 12, we present images of the density and gas pressure at $t = 0.2L$ computed with 512×512 grid points. The results are in excellent agreement with those presented in [45].

5.2.3. MHD rotor problem

The third 2D test is the MHD rotor problem. The problem was taken from [1]. It initiates a high density rotating disk with radius $r_0 = 0.1$ of fluid measured from the center point $(x, y) = (0.5, 0.5)$. The ambient fluid outside of the spherical region of $r_1 = 0.115$ has low density and $v_x = v_y = 0$, and the fluid between the high density disk fluid and ambient fluid ($r_1 > r > r_0$, where $r = \sqrt{(x-0.5)^2 + (y-0.5)^2}$) has linear density and angular speed profile with $\rho = 1 + 9f$, $v_x = -f v_0 (y - 0.5)/r$ and $v_y = -f v_0 (x - 0.5)/r$ where $f = (r_1 - r)/(r_1 - r_0)$. Two initial value sets of v_0, p, B_x and γ provided in [1] and [53] were tested. The initial condition for 2D MHD Rotor problem is listed as follows

$$\text{spherical region center} = (0.5, 0.5), \qquad r_0 = 0.1$$
$$r_1 = 0.115, \qquad f = (r_1 - r)/(r_1 - r_0)$$
$$(0 \leqslant x \leqslant 1)\ (0 \leqslant y \leqslant 1) \tag{49}$$
$$r < r_0$$
$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{Bmatrix} -v_0 (y - 0.5)/r_0 \\ v_0 (x - 0.5)/r_0 \\ 0 \end{Bmatrix} \tag{50}$$

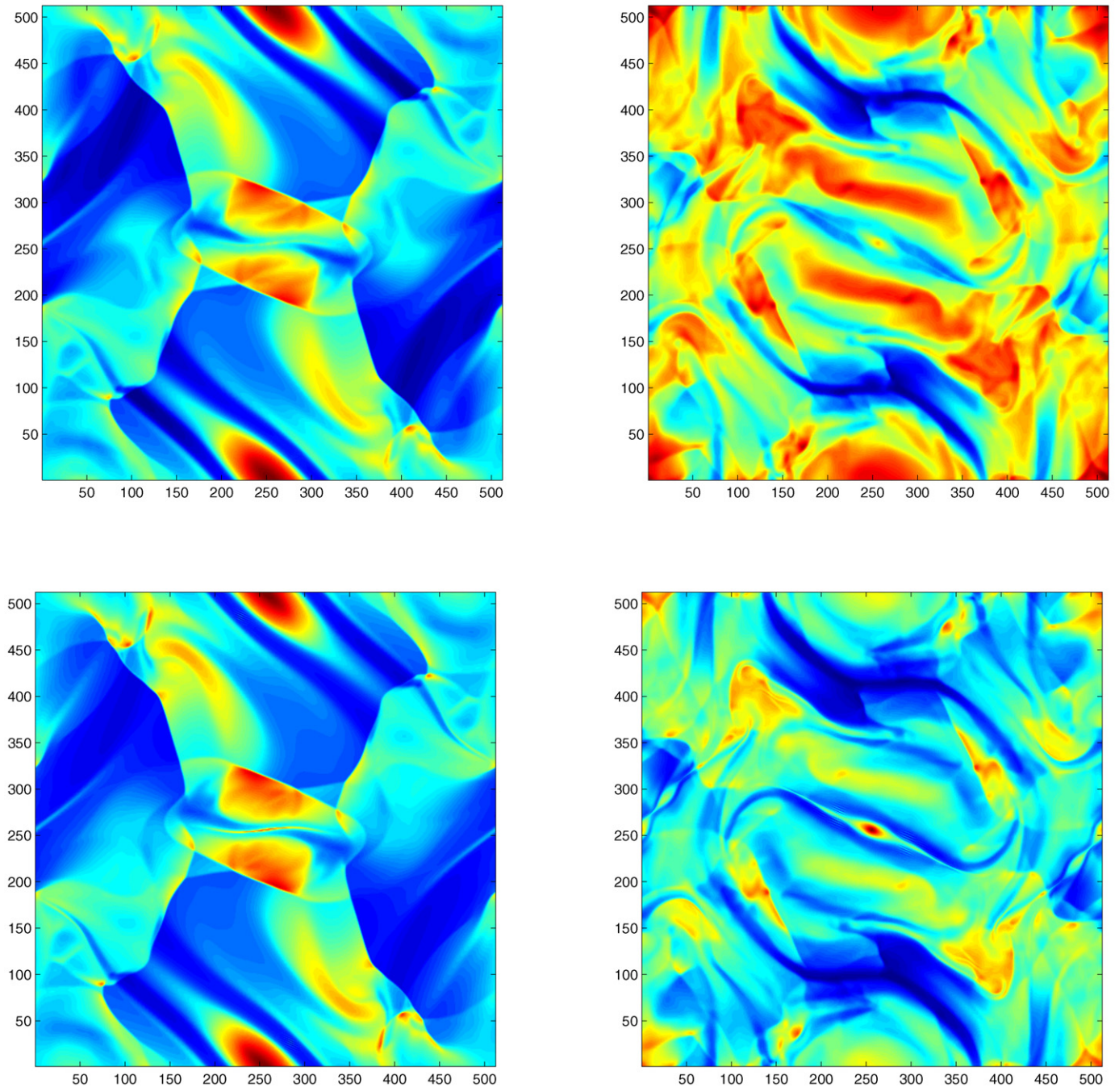

Fig. 11. Results of the density (top) and gas pressure (bottom) of the Orszag–Tang vortex test at $t = 0.5L$ (left) and $t = 1.0L$ (right) computed with 512×512 grid points.

$$r_0 < r < r_1$$

$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{Bmatrix} -f v_0 (y - 0.5)/r \\ -f v_0 (x - 0.5)/r \\ 0 \end{Bmatrix} \tag{51}$$

$$r > r_1$$

$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{52}$$

$$\rho = \begin{cases} 10 & r < r_0 \\ 1 + 9f & r_0 < r < r_1 \\ 1 & r > r_1 \end{cases} \tag{53}$$

First rotor problem:

$$v_0 = 2, \qquad p = 1, \qquad \gamma = 1.4$$

$$t_{\max} = 0.15, \qquad \begin{Bmatrix} B_x \\ B_y \\ B_z \end{Bmatrix} = \begin{Bmatrix} 5/\sqrt{4\pi} \\ 0 \\ 0 \end{Bmatrix} \tag{54}$$

Second rotor problem:

$$v_0 = 1, \qquad p = 0.5, \qquad \gamma = 5/3$$

$$t_{\max} = 0.295 \begin{Bmatrix} B_x \\ B_y \\ B_z \end{Bmatrix} = \begin{Bmatrix} 2.5/\sqrt{4\pi} \\ 0 \\ 0 \end{Bmatrix} \tag{55}$$

In Fig. 13, we present images of the density, gas pressure of the two rotor problems computed with 512×512 grid points. The results are in excellent agreement with those presented in [1] and [53].

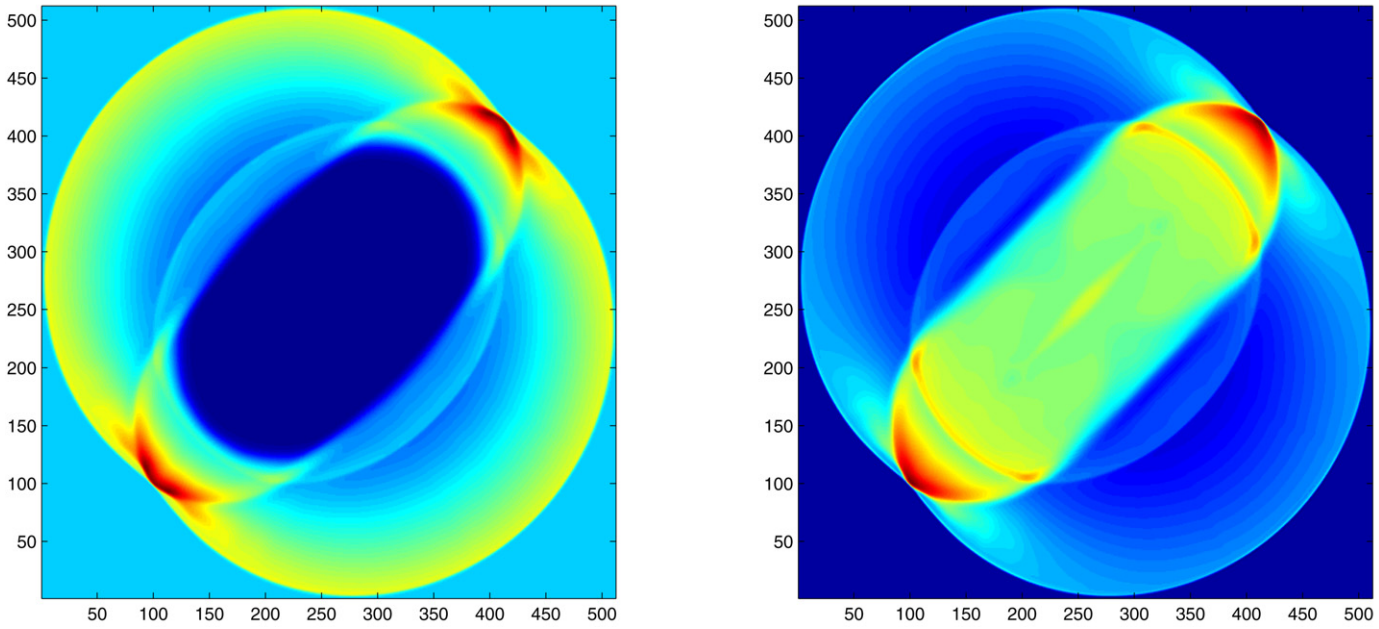

Fig. 12. Results of the density (left) and gas pressure (right) of the 2D blast wave test at $t = 0.2L$, computed with 512×512 grid points.

5.3. Three-dimensional blast wave problem

The 3D version of MHD spherical blast wave problem was also tested. The condition is listed as follows [45]

$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{56}$$

$$\begin{Bmatrix} B_x \\ B_y \\ B_z \end{Bmatrix} = \begin{Bmatrix} 1/\sqrt{3} \\ 1/\sqrt{3} \\ 1/\sqrt{3} \end{Bmatrix} \tag{57}$$

$$p = \begin{cases} 10 & \text{inside the spherical region} \\ 0.1 & \text{outside the spherical region} \end{cases} \tag{58}$$

$$\rho = 1, \qquad \gamma = 5/3$$

$$\text{spherical region center} = (0.5, 0.5, 0.5), \qquad r = 0.1$$

$$(0 \leqslant x \leqslant 1)\ (0 \leqslant y \leqslant 1)\ (0 \leqslant z \leqslant 1) \tag{59}$$

Figs. 14 and 15 show the results of 3D blast wave problem, which include the density, gas pressure, and magnetic pressure at $t = 0.1L$ and $t = 0.2L$ sliced along the x–y plane at $z = 0.5$. The test was computed with $128 \times 128 \times 128$ grid points. Due to the scarcity of published 3D test results, we do not make direct contact with results presented in the literature here. Considering only the $\boldsymbol{u}$ and $\boldsymbol{B}$, the memory requirement of 256^3 MHD problem is about 512 MB GRAM for single precision and 1024 MB GRAM for double precision, respectively. If the storage of intermediate results such as $\boldsymbol{B}_{temp}$, $\boldsymbol{u}_{temp}$, $\mathbf{flux_{temp}}$ and F etc. (see Section 4.3) are considered, the amount of memory requirement will be about 2.25 GB (single precision). As we mentioned in Section 4.3, not all the capacity of GRAM can be used to store data arbitrarily. As we said in the beginning of this section, there are actually two GPUs inside the GTX 295 and the 1.75 GB GRAM is the total amount of the GRAM shared by two GPUs, so that only less than $1.75/2 = 0.875$ GB GRAM can be used. As a result, the test of 3D problem with 256^3 resolution are not able to be provided on a graphics card.

6. Accuracy evaluation

In MHD simulations, accuracy is always to be considered since the error may increase fast and crash the simulation if low precision is used for computation. Scientific computations such as MHD simulation mostly use double precision to reduce the computational errors. In this section, the results generated by *GPU-MHD* using single precision and double precision modes are shown and compared.

The difference between the results of double precision and single precision computation of the $512 \times 1 \times 1$ one-dimensional Brio–Wu shock tube problem is shown in Fig. 16. Two curves are almost the same but there are actually some differences with the error lying in the region of $\leqslant \pm 10^{-6}$.

In 2D cases, the absolute difference between the results of double precision and single precision computation of the MHD Rotor test ($t = 0.15L$) and the Orszag–Tang vortex test ($t = 0.5L$ and $t = 1.0L$) are shown in Figs. 17 and 18, respectively. The double precision computation results of both tests are also shown on the left-hand side of these figures.

For the MHD Rotor test, even the resulting image (left in Fig. 17) looks similar to the single precision resulting image (top-left of Fig. 13), the large differences at the dense region can be found. Experimental result shows that the maximum *error* is larger than $\pm 3.5 \times 10^{-4}$.

Fig. 18 shows the absolute difference between the results of double precision and single precision computation of the Orszag–Tang test at $t = 0.5L$ and $t = 1.0L$. As the simulation time increases, the maximum *error* increases from about $\pm 8 \times 10^{-5}$ to ± 0.03.

Figs. 19 and 20 show the resulting images of the simulation using double precision and the contours of the absolute differences between the results of double precision and single precision computation of 3D blast wave test with 128^3 grid points at $t = 0.1L$ and $t = 0.2L$. As it is a high dimension computation in low resolution, the differences between them are clear. The number of grid points having a higher difference value increases, and the *error* is still less than 10^{-6}. Small difference value makes the double precision resulting images (Figs. 19 and 20) looked similar to the single precision resulting images (Figs. 14 and 15).

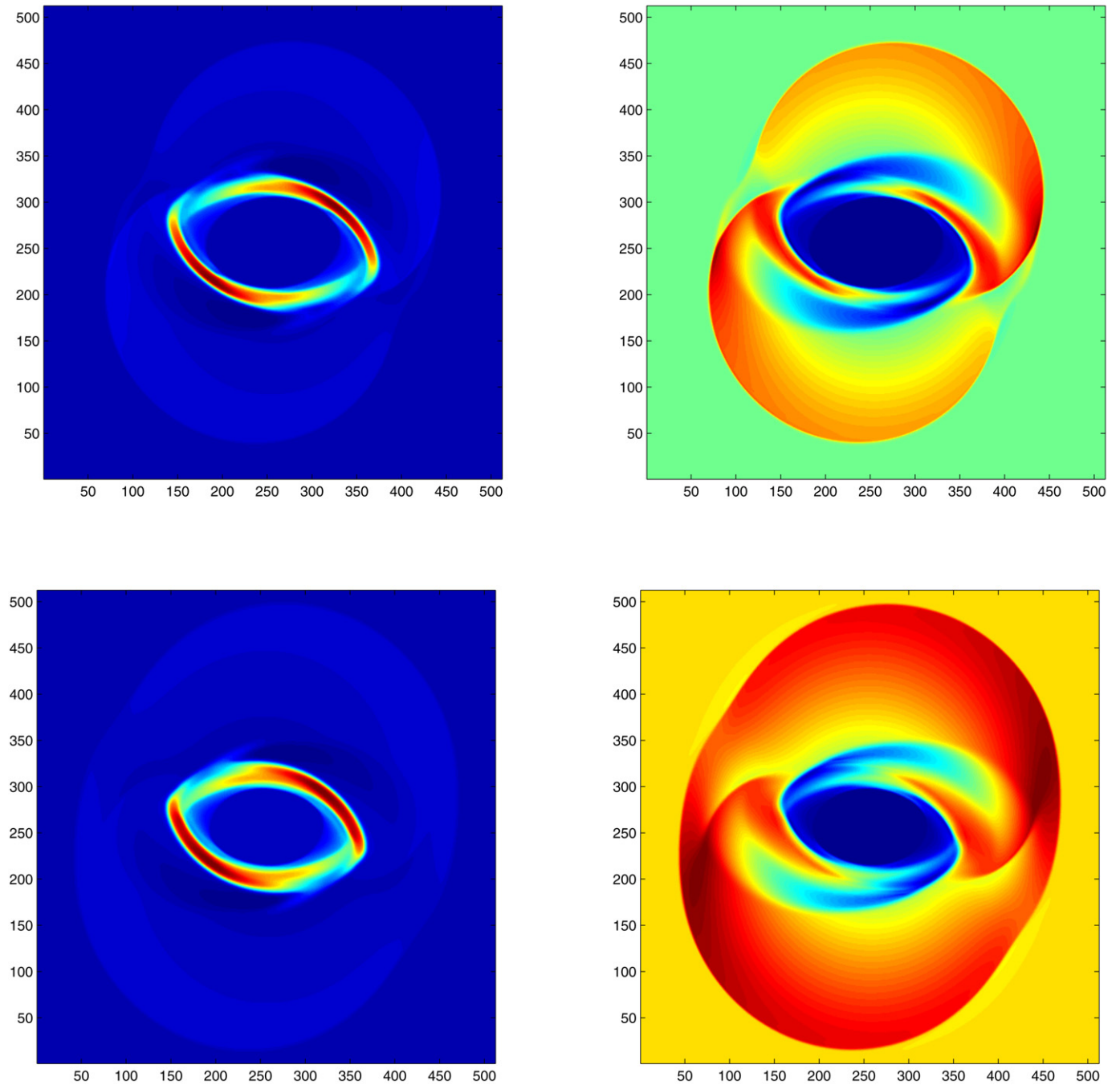

Fig. 13. Results of the density (top-left), gas pressure (top-right) of the first MHD rotor test at $t = 0.15L$, results of the density (bottom-left), gas pressure (bottom-right) of the second MHD rotor test at $t = 0.295L$, both computed with 512×512 grid points.

An important point can be inferred from these experiments that not only the grid points at the high density region has high difference value, but also the number of grid points having high difference values and the amount of the difference values are increasing along with the increase of the simulation time. Higher dimension is another factor in introducing noticeable differences between the computational results with different precisions because higher dimension means a grid point has more neighbors and more neighbors need more computation steps in one time step. As a result the differences become more obvious. Therefore, for a long-term simulation, double precision computation is a highly desirable.

The original Fortran code [35] is a second-order accurate high-resolution TVD MHD code. Empirically, as demonstrated by the experiments reported in this section, we consider that the *GPU-MHD* is sufficiently accurate to capture the forward and reverse shocks as well as any other discontinuities such as contact discontinuities which are important in space physics. As the *GPU-MHD* is a dimensional-splitting based code, there are two major drawbacks: (i) the code is unable to evolve the normal (to the sweep direction) magnetic field during each sweep direction [12], and (ii) splitting errors will generally be introduced due to the fact that the linearized Jacobian flux matrices do not commute in most of the nonlinear multidimensional problems [24].

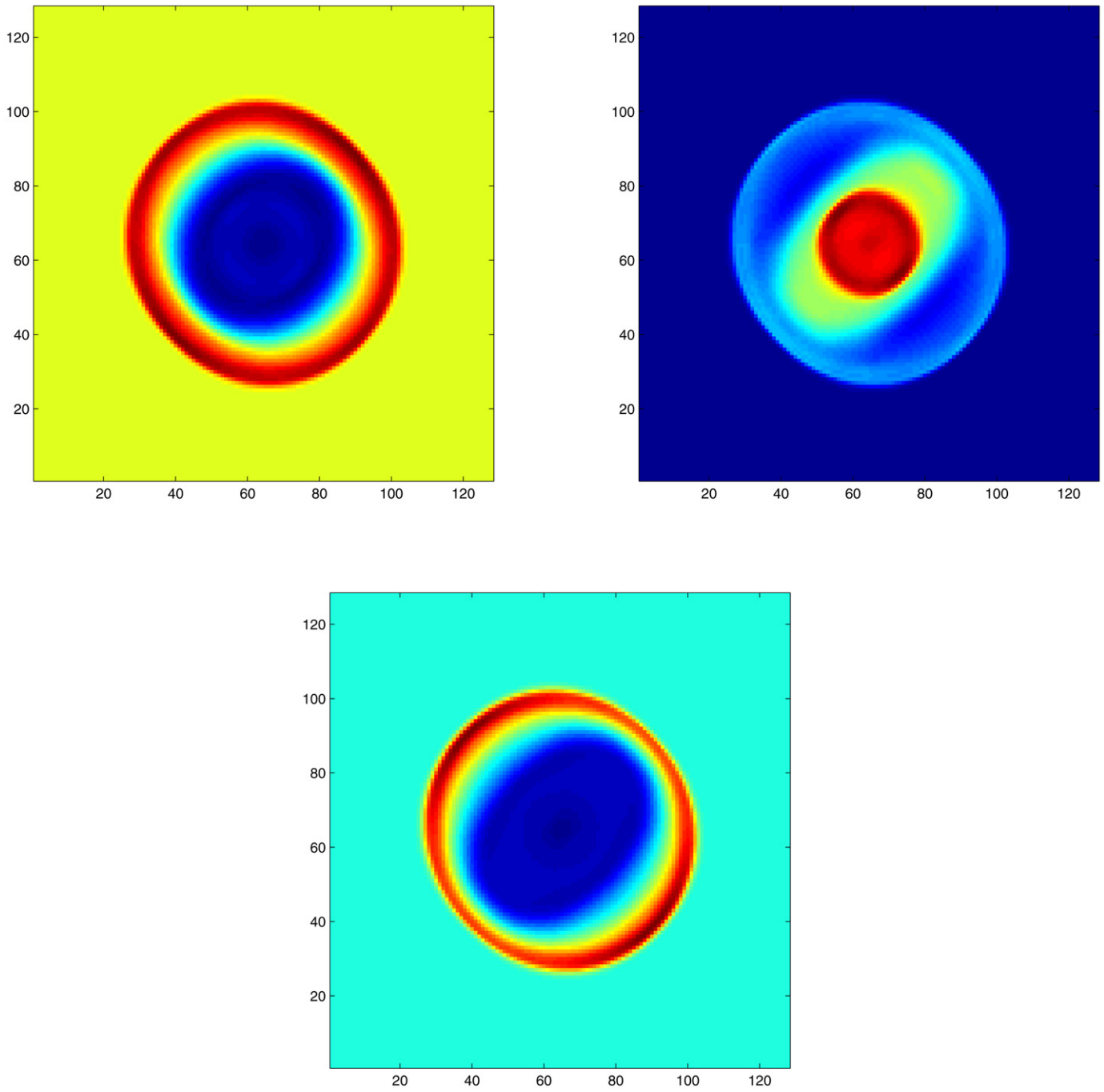

Fig. 14. Results of the density (top-left), gas pressure (top-right) and magnetic pressure (bottom) of 3D blast wave test at $t = 0.1L$ sliced along the x–y plane at $z = 0.5$ and computed with $128 \times 128 \times 128$ grid points.

7. Performance measurements

The performance measurements of the GPU and CPU implementations as well as the computation using double precision and single precision are carried out in this section. Different numbers of grid points and different dimensions were used in the performance tests. We run both *GPU-MHD* and Pen et al.'s FORTRAN/CPU MHD code [37] to perform the simulations on a PC with Intel Core i7 965 3.20 GHz CPU, 6G main memory, running Microsoft Windows XP 64-bit Professional operating system. Two graphics cards were tested: NVIDIA GeForce GTX 295 with 1.75G video memory and GTX 480 (Fermi) with 1.5G video memory. The Fortran compiler and GPU development toolkit we used are G95 Stable Version 0.92 and NVIDIA CUDA 3.2, respectively. The *GPU-MHD* was designed for three-dimensional problems, thus the dimensions are expressed in three-dimensional form in all figures shown in this section. For the 1D test, the 1D Brio–Wu shock tube problem (see Section 5.1.1) was used. For the 2D test, the 2D Orszag–Tang problem (see Section 5.2.1) was used. For the 3D test, the 3D blast wave problem (see Section 5.3) was used.

Fig. 21 reports the comparison of the *GPU-MHD* and the FORTRAN/CPU code of 1D test with different numbers of grid points in single and double precisions. In the single precision mode, basically there is only about 10 times speedup ($4096 \times 1 \times 1$ case)

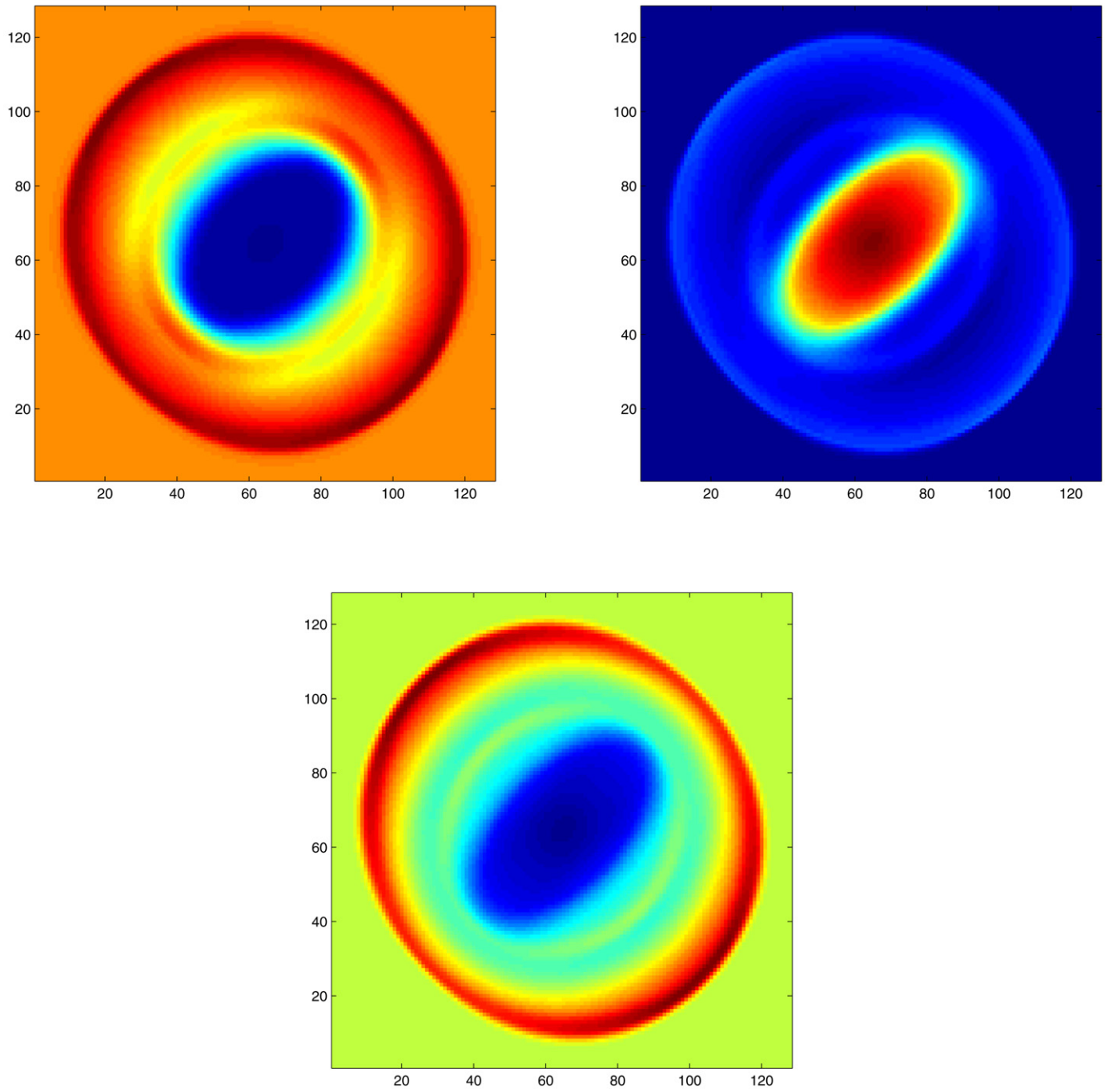

Fig. 15. Results of the density (top-left), gas pressure (top-right) and magnetic pressure (bottom) of 3D blast wave test at $t = 0.2L$ sliced along the x–y plane at $z = 0.5$ and computed with $128 \times 128 \times 128$ grid points.

since the number of grid points is small. It should be realized that the amount of speedup is increased as long as the resolution is increased but dropped when the resolution reaches 512. It is because the "max threads per block" of the GTX 295 is 512, all the computations are handled within one block and a very high processing speed can be archived. On the GTX 480, there is about 80 times speedup ($4096 \times 1 \times 1$ case) and the amount of speedup is increased linearly thanks to the improvement of the floating point arithmetic in the GTX 480 [32] (the new IEEE 754-2008 floating-point standard which provides the fused multiply-add (FMA) instruction for both single (512 FMA ops/clock) and double precision arithmetic (256 FMA ops/clock)). In the double precision mode, around 10 times and 60 times speedup ($4096 \times 1 \times 1$ case) is achieved on the GTX 295 and the GTX 480, respectively.

Table 1 gives the comparison of the *GPU-MHD* using single precision and double precision of 1D test with different numbers of grid points. On the GTX 295, a similar speed drop happened in both the single and double precision modes, but it occurred at different resolutions: 512 in single precision and 256 in double precision. This is not strange and it is not difficult to understand since the double precision has double the size of data to be handled by the processors. Except for the special case of 512 resolution, the processing speed in both modes are very closed to one another.

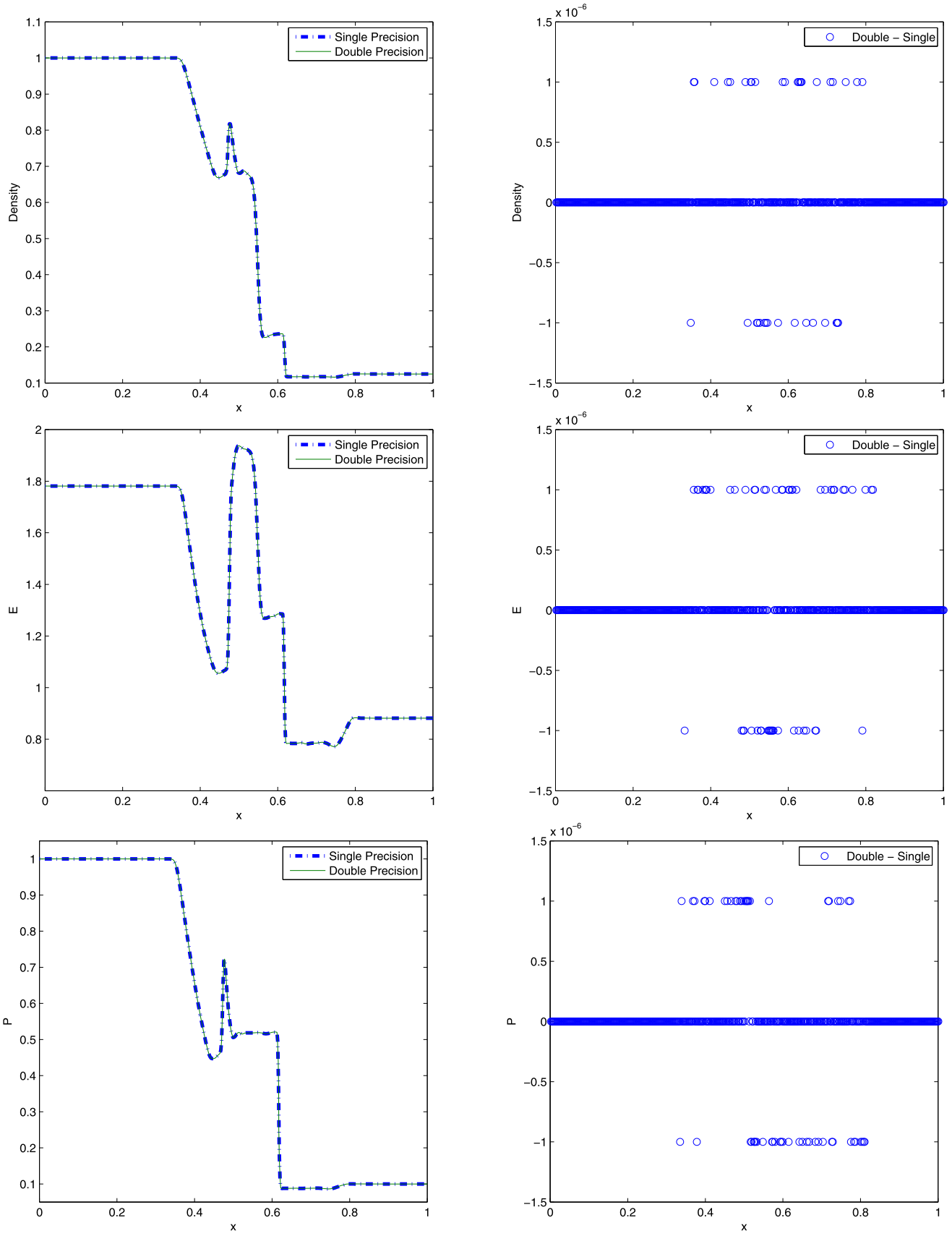

Fig. 16. Result of $\rho_{double} - \rho_{single}$ (top), $E_{double} - E_{single}$ (middle) and $p_{double} - p_{single}$ (bottom) of the 1D Brio–Wu shock tube problem at $t = 0.08L$ with 512 grid points.

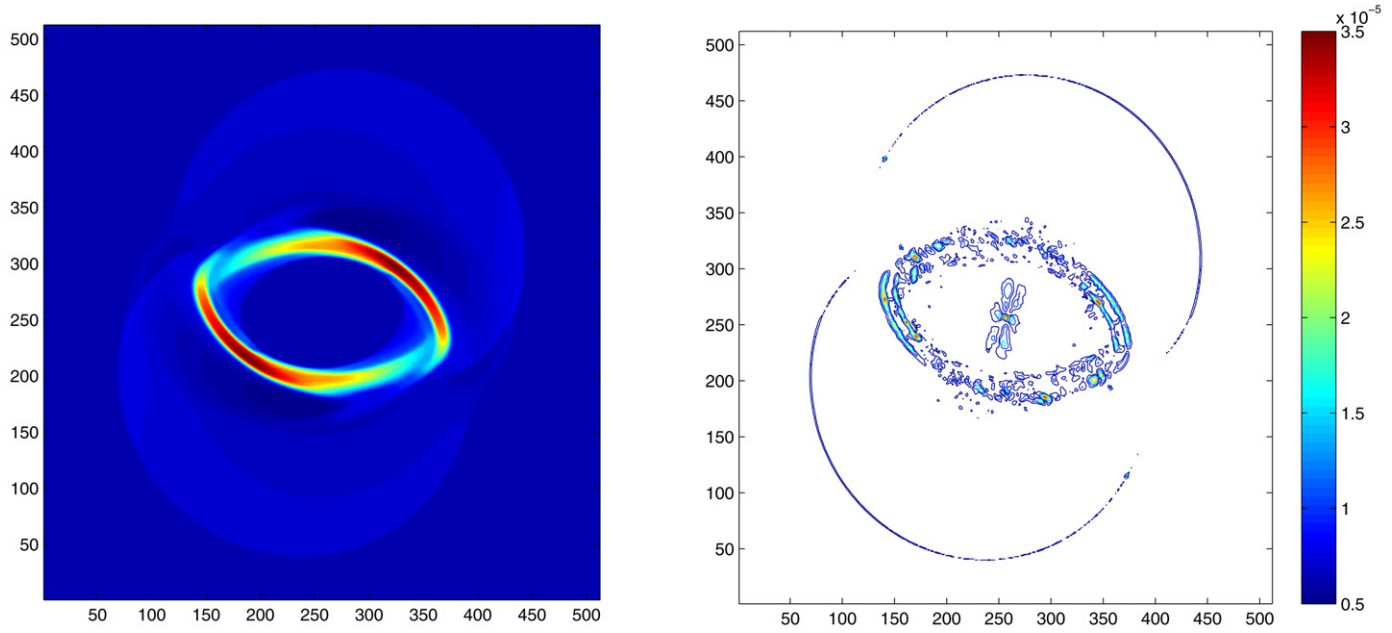

Fig. 17. Results of ρ_{double} (left) and $|\rho_{double} - \rho_{single}|$ (right) of the MHD rotor problem at $t = 0.15L$ with 512^2 grid points.

On the GTX 480, the performance between single and double precision is quite close.

The comparison of the *GPU-MHD* and the FORTRAN/CPU code of the 2D test with different numbers of grid points in single and double precisions is presented in Fig. 22. In the 2D case, a significant performance improvement is observed, especially when the numbers of grid points are 512^2 and 1024^2, respectively, a speedup of around 150 and around 200 are achieved on the GTX 295. On the GTX 480, a speedup of around 320 and around 600 is achieved, respectively. This is due to the significant improvement of the performance in double precision on the GTX 480 (with double precision floating point capability of 256 FMA ops/clock) is better than that of the GTX 295 (30 FMA ops/clock).

Table 2 presents the comparison of the *GPU-MHD* using single precision and double precision of 2D test with different numbers of grid points. The significant performance difference is noticeable when the number of grid points is increased. However, it still keeps a ratio increasing slowly from 1.118 to 1.6218 while the resolution increases from 128^2 to 1024^2. The double/single precision ratios on the GTX 480 vary between 1.2804 and 1.7300.

Fig. 23 shows the comparison of the *GPU-MHD* and the FORTRAN/CPU code of the 3D test with different numbers of grid points in single and double precisions. The performance of the *GPU-MHD* is faster than the FORTRAN/CPU code about 60 times and 84 times when the numbers of grid points are 64^3 and 128^3 on the GTX 295, respectively. The corresponding speedups on the GTX 480 are about 260 and 155, respectively.

Table 3 shows the comparison of the *GPU-MHD* using single precision and double precision of the 3D test with different numbers of grid points. The ratio is 1.6020 when the number of grid points is 64^3, and is 1.7389 when the number of grid points is 128^3 on the GTX 295. The corresponding ratio on the GTX 480 are 1.3973 and 1.7133, respectively.

The performance tests show that when the number of grid points of the test problems is small, such as those in the 1D case, the *GPU-MHD* can give a significant performance improvement over that of the single CPU case. When the number of grid points increases, an obvious disparity of performance becomes clear, especially for the multidimensional cases (see Figs. 22 and 23). Computation using double precision on GPUs prior to Fermi is known to have a very low performance compared to the single precision. However, in the performance comparison between single precision and double precision modes in the *GPU-MHD*, the ratios of the processing speed between two modes show that the *GPU-MHD* is efficient enough in the double precision computations. The performance results show that CUDA is an attractive parallel computing environment for MHD simulations.

Tables 4, 5, and 6 show the performance comparisons between the FORTRAN/CPU and the *GPU-MHD* of major calculations in single precision at different resolutions of the 1D, 2D and 3D tests, respectively. These tables provide the information of how the computational effort spent on which step in the 1D, 2D, and 3D problems. From these tables we observe that the calculations of fluids ($fluid_x$, $fluid_y$, and $fluid_z$) spend most of the computational times in all kinds of problems. Even though the data accesses of L_x and L_y are not coalesced (see Section 4.2), the computational times spent on each part of L_x, L_y, and L_z in the same problem are very close. It shows that the GT200 and the Fermi architectures handle the data accesses that are not perfectly coalesced quite well in our study.

8. Visualization of the simulation results

There is a need to visualize the MHD simulation data, for examples, Daum [7] developed a toolbox called *VisAn MHD* in MATLAB for MHD simulation data visualization and analysis. With the help of GPUs, Stantchev et al. [47] used GPUs for computation and visualization of plasma turbulence. In *GPU-MHD*, using the parallel computation power of GPUs and CUDA, the simulation results of one time step can be computed in dozens or hundreds milliseconds. According to the efficiency of the *GPU-MHD*, near real-time visualization is obtained for 1D and 2D problems. The motion or attributes of the magnetic fluid can be computed and rendered on the fly. So the changes of the magnetic fluid during the simulation can be observed in real-time.

By adding the real-time visualization, the flow of the *GPU-MHD*, Fig. 3 is extended as Fig. 24.

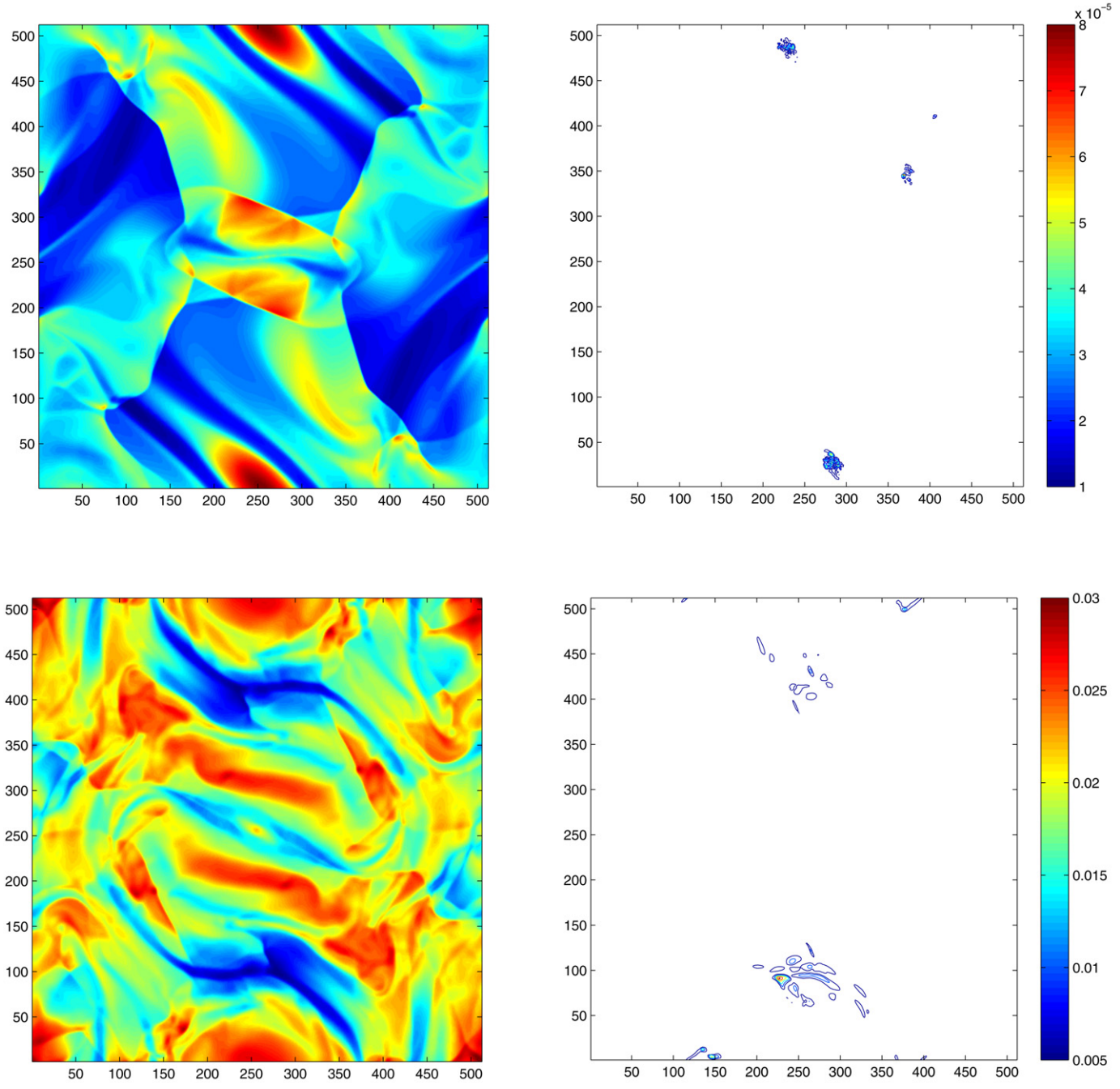

Fig. 18. Results of ρ_{double} (left) and $|\rho_{double} - \rho_{single}|$ (right) of the Orszag–Tang problem at $t = 0.5L$ (top) and $t = 1.0L$ (bottom) with 512^2 grid points.

The *GPU-MHD* provides different visualization methods for one-dimensional, two-dimensional and three-dimensional problems.

To visualize one-dimensional problems for each time step, the simulation results are copied to the CUDA global memory that mapped to the Vertex Buffer Object (VBO) [57]. For all grid points, one grid point is mapped to one vertex. The position of each grid point is mapped as the x-position of the vertex and the selected physical value (ρ, p, etc.) is mapped as the y-position of the vertex. Then a curve of these vertices is drawn. Since the VBO is mapped to CUDA global memory and the simulation results are stored in the GRAM, the copying and mapping operations are fast. Experimental result shows that *GPU-MHD* with real-time visualization can achieve 60 frame per second (FPS) in single precision mode and 30 FPS in double precision mode on the GTX 295. On the GTX 480, around 60 FPS in both single and double precisions can be achieved. Fig. 25 shows two example images of 1D visualizations using the *GPU-MHD*.

The operational flow of visualization of 2D problems is similar to that in the 1D visualization. However, instead of Vertex Buffer Object (VBO), Pixel Buffer Object (PBO) [57] is used. For each time step, the simulation results are copied to the CUDA global memory that are then mapped to the PBO. For all grid points, one grid point is mapped to one pixel. The x- and y-position of each grid point are mapped as the corresponding x-position and the y-position of the vertex and the selected physical value (ρ, p, etc.) is mapped as the color of the pixel to form a color image. To render this color image, a transfer function is set to map the physical value to the color of the pixel and then the resulting image is drawn. Similar to VBO, PBO is also mapped to the CUDA global memory and the simulation results are stored in the GRAM, so the copy-

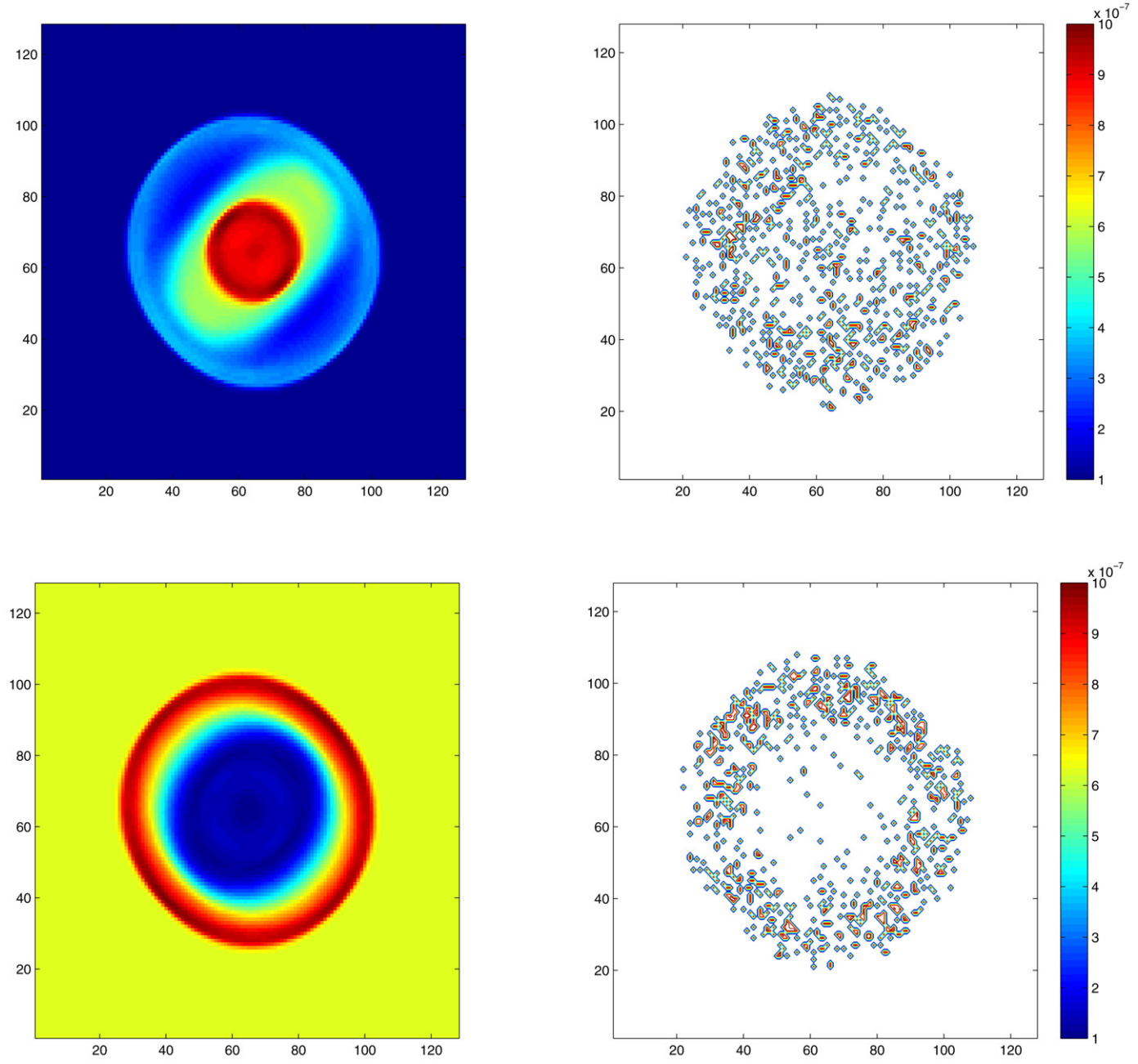

Fig. 19. Results of p_{double} (top-left), ρ_{double} (bottom-left) and $|p_{double} - p_{single}|$ (top-right), $|\rho_{double} - \rho_{single}|$ (bottom-right) of the 3D blast wave problem at $t = 0.1L$ with 128^3 grid points.

ing and mapping operations are also fast and do not affect too much of the performance. Although the number of grid points in the 2D problem is much larger than that in the one-dimension problem, the FPS still reaches 10 in the single precision mode and 6 in the double precision mode on the GTX 295 when the number of grid points is 512^2, still giving acceptable performance to the user. On the GTX 480, 22 FPS in single precision and 17 FPS in double precision are achieved and thus interactive rates are possible. Fig. 26 shows two example images of 2D visualizations using *GPU-MHD*.

However, visualization of 3D problem is different to that in the 1D and 2D problems. GPU-based volume visualization method [15] and texture memory (or video memory) are used. Unfortunately, the current version (Version 3.2) of CUDA does not provide the feature to copy the data from the CUDA global memory to texture memory directly, even though both of them are in the GRAM. On the other hand, texture memory is readable but is not rewritable in CUDA. So the simulation results have to be copied to the main memory first, and then be copied to texture memory. In addition, the number of grid points is usually large compared to that of the 2D problems and volume visualization techniques are somewhat time-consuming. As a result, on the GTX 295, *GPU-MHD* only achieves 2 FPS in single precision mode and 1 FPS in double precision mode when the number of grid points is 128^3, and it is far from real-time. Nevertheless, we still obtain 10 FPS (single precision mode) and 6 FPS (double precision mode) for performing the simulation of problems with a resolution of 64^3 and about 20 FPS (single and double precision modes) for problems with a resolu-

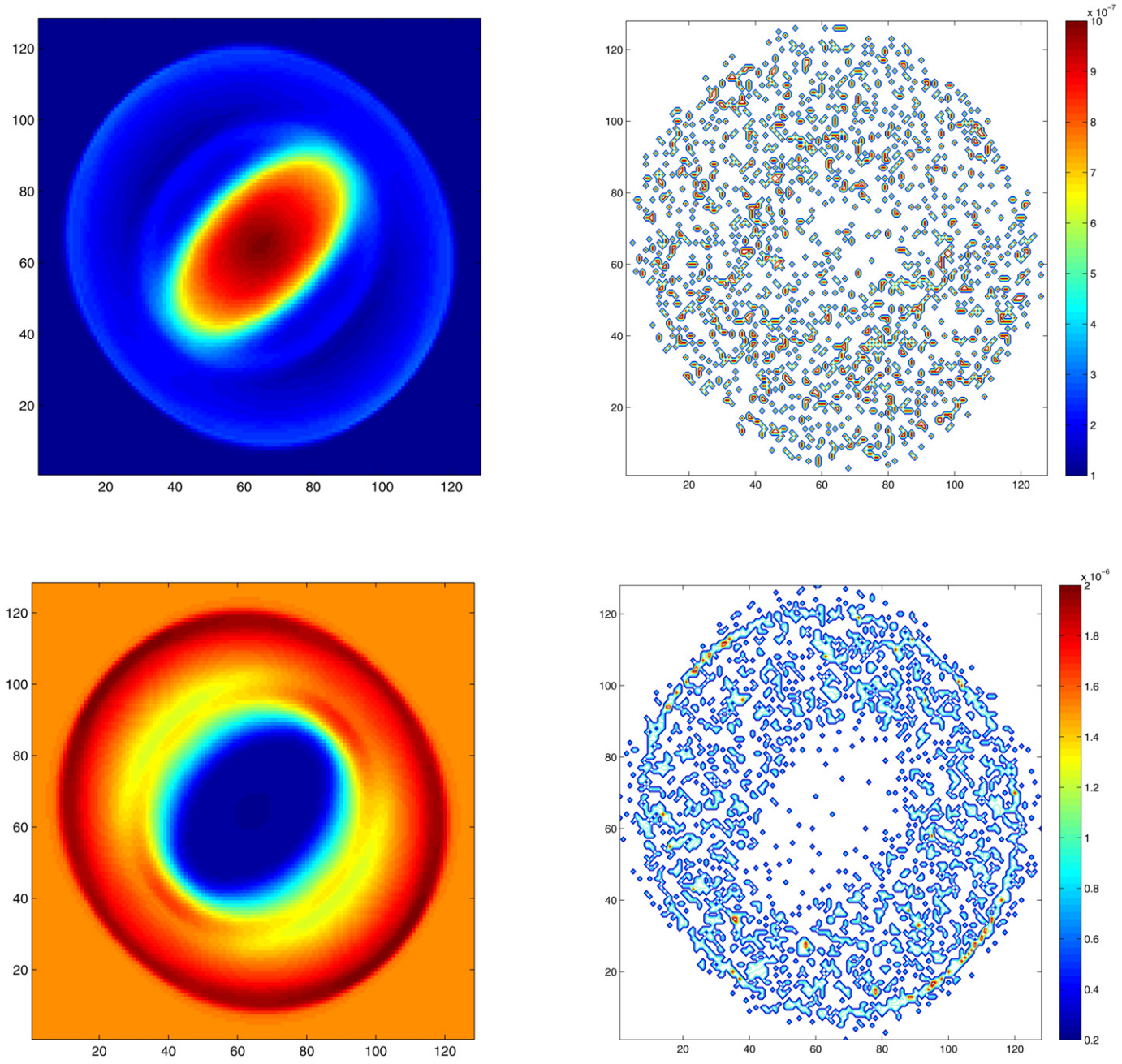

Fig. 20. Result of p_{double} (top-left), ρ_{double} (bottom-left) and $|p_{double} - p_{single}|$ (top-right), $|\rho_{double} - \rho_{single}|$ (bottom-right) of the 3D blast wave problem at $t = 0.2L$ with 128^3 grid points.

tion of 32^3. On GTX 480, we can obtain 60 FPS for both single and double precision for 32^3 grid points, 20 FPS (single) and 16 FPS (double) for 64^3 grid points, and 6.1 FPS (single) and 3.6 FPS (double) for 128^3 grid points. Fig. 27 shows two example images of 2D visualizations using *GPU-MHD*.

9. Conclusion and future work

In this paper we present, to the best of the author's knowledge, the first implementation of MHD simulations entirely on GPUs with CUDA, named *GPU-MHD*, to accelerate the simulation process. The aim of this paper is to present a GPU implementation in detail, demonstrating how a TVD based MHD simulations can be implemented efficiently for NVIDIA GPUs with CUDA. A series of numerical tests have been performed to validate the correctness of our code. Accuracy evaluation by comparing single and double precision computation results is also given, indicating that double precision support on GPUs is a necessity for long-term MHD simulation. Performance measurements of both single and double precision modes of *GPU-MHD* are conducted on the GT200 architecture (GTX 295) and the Fermi architecture (GTX 480). These measurements show that our GPU-based implementation achieves between one and two orders of magnitude improvement depending on the graphics card used, the problem size, and the precision when comparing to the original serial CPU MHD implementation. In order to provide the user better understanding of the problems being investigated during the simulation process, we have extended *GPU-MHD* to support visualization of the simulation results. With *GPU-MHD*,

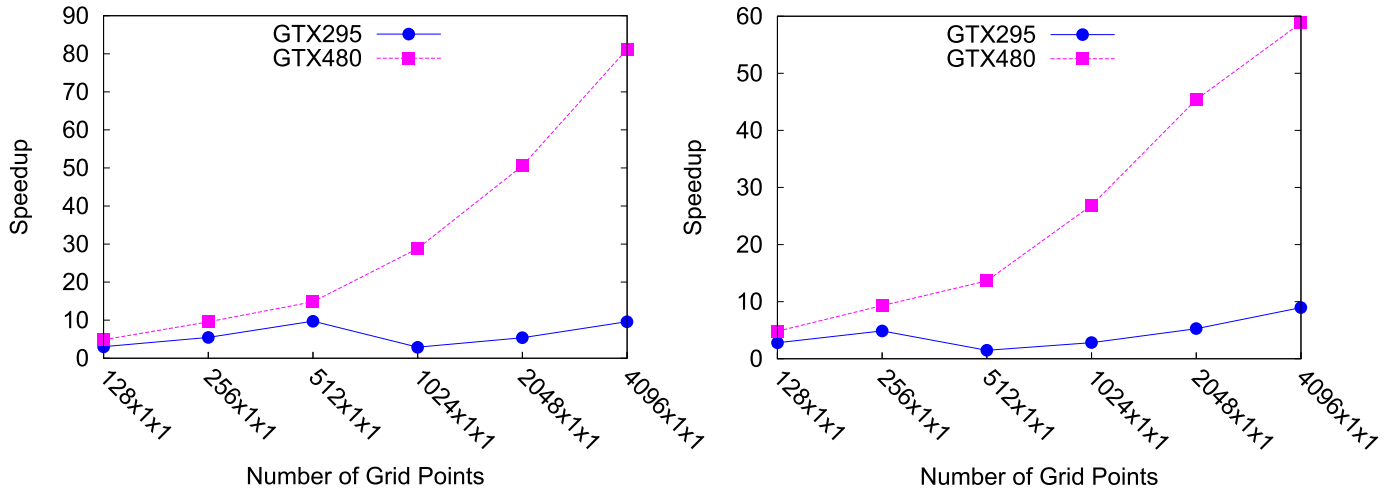

Fig. 21. The speedup of the 1D test using the *GPU-MHD* (left: single precision; right: double precision), compared with the FORTRAN/CPU code at different resolutions.

Table 1
The performance results of the 1D test between single precision and double precision of *GPU-MHD* at different resolutions.

Number of grid points	GPU	Double precision (ms/step)	Single precision (ms/step)	Ratio
$128 \times 1 \times 1$	GTX 295	3.9	3.6	1.0833
	GTX 480	2.26	2.24	1.0089
$256 \times 1 \times 1$	GTX 295	4.5	4.0	1.1250
	GTX 480	2.35	2.29	1.0262
$512 \times 1 \times 1$	GTX 295	29.5	4.5	6.5555
	GTX 480	3.21	2.95	1.0881
$1024 \times 1 \times 1$	GTX 295	31.0	30.0	1.0333
	GTX 480	3.25	3.03	1.0726
$2048 \times 1 \times 1$	GTX 295	33.0	32.2	1.0248
	GTX 480	3.82	3.43	1.1137
$4096 \times 1 \times 1$	GTX 295	39.1	36.4	1.0742
	GTX 480	5.95	4.31	1.3805

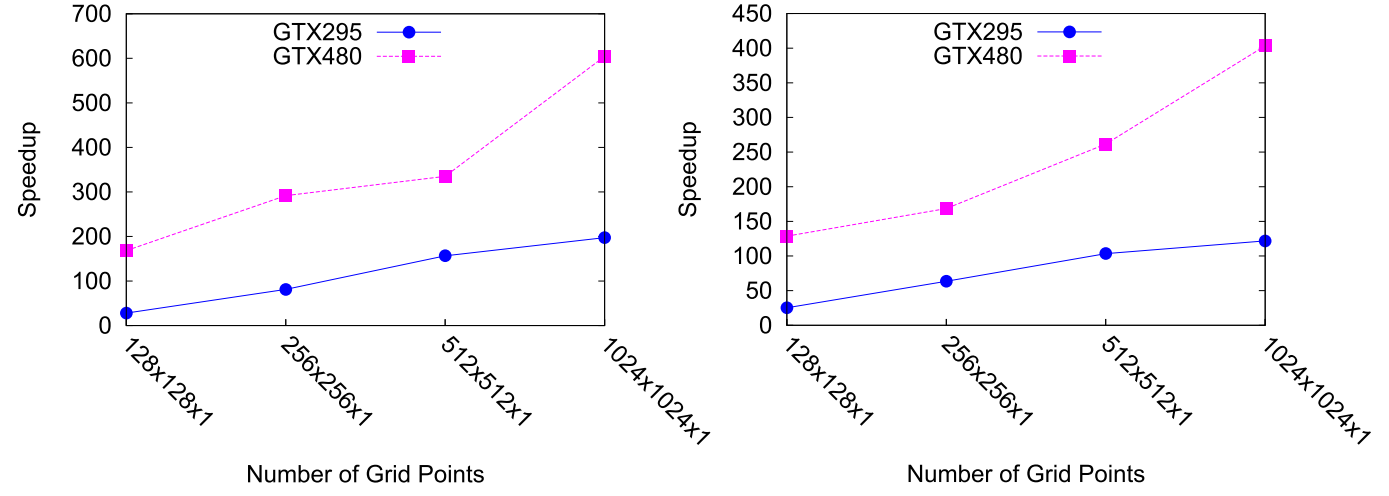

Fig. 22. The speedup of the 2D test using the *GPU-MHD* (left: single precision; right: double precision), compared with the FORTRAN/CPU code at different resolutions.

the whole MHD simulation and visualization process can be performed entirely on GPUs.

There are two possible future research directions in our future work, firstly, we wish to extend *GPU-MHD* for multiple GPUs and GPU cluster [42] to fully exploit the power of GPUs. Secondly, we will investigate implementing other recent high-order Godunov MHD algorithms such as [23] and [50] on GPUs. These GPU-based algorithms will serve as the base of our GPU framework for simulating large-scale MHD problems in space weather modeling.

Table 2
The performance results of the 2D test between single precision and double precision of *GPU-MHD* at different resolutions.

Number of grid points	GPU	Double precision (ms/step)	Single precision (ms/step)	Ratio
$128 \times 128 \times 1$	GTX 295	35.8	32.2	1.1118
	GTX 480	7.01	5.36	1.3078
$256 \times 256 \times 1$	GTX 295	57.3	44.8	1.2790
	GTX 480	21.59	12.48	1.7300
$512 \times 512 \times 1$	GTX 295	142.3	94.0	1.5138
	GTX 480	56.26	43.94	1.2804
$1024 \times 1024 \times 1$	GTX 295	478.6	295.1	1.6218
	GTX 480	144.48	96.43	1.4983

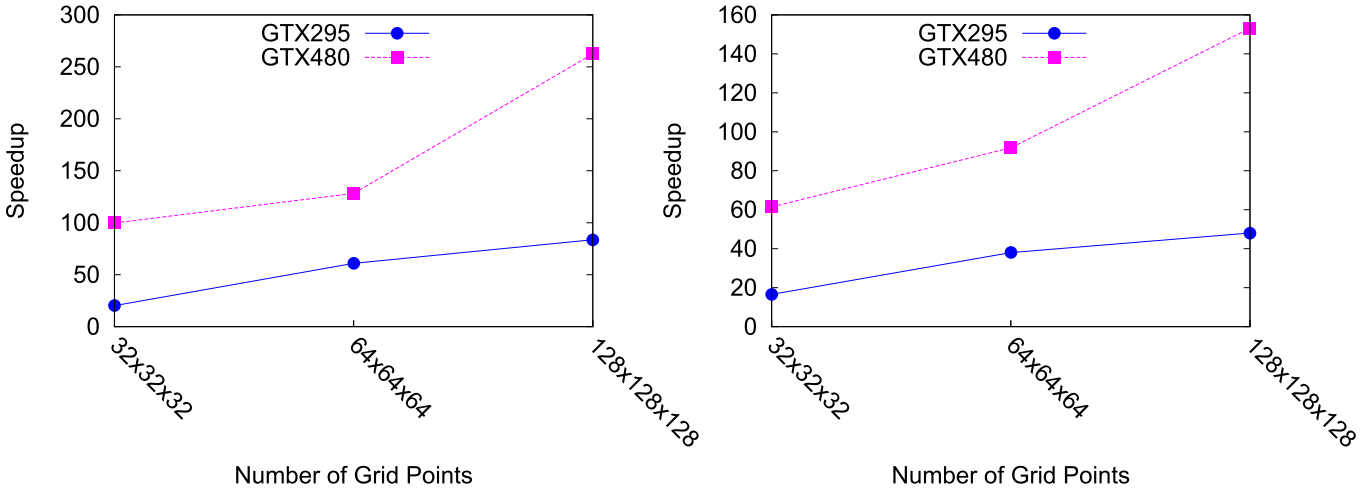

Fig. 23. The speedup of the 3D test using the *GPU-MHD* (left: single precision; right: double precision), compared with the FORTRAN/CPU code at different resolutions.

Table 3
The performance results of the 3D test between single precision and double precision of *GPU-MHD* at different resolutions.

Number of grid points	GPU	Double precision (ms/step)	Single precision (ms/step)	Ratio
$32 \times 32 \times 32$	GTX 295	44.6	36.6	1.2186
	GTX 480	12.03	7.41	1.6235
$64 \times 64 \times 64$	GTX 295	145.3	90.7	1.6020
	GTX 480	60.18	43.07	1.3973
$128 \times 128 \times 128$	GTX 295	880.6	506.4	1.7389
	GTX 480	276.40	161.33	1.7133

Table 4
Performance comparison (ms/step) between the FORTRAN/CPU and the *GPU-MHD* of major calculations in single precision at different resolutions of the 1D test.

Number of grid points	Operations	FORTRAN/CPU	GTX 295	GTX 480
$512 \times 1 \times 1$	CFL	0.18	0.14	0.28
	$fluid_x$	1.07	0.93	0.61
	$B_{x\to y}$ and $B_{x\to z}$	0.36	0.18	0.11
	$fluid_y$	5.85	0.40	0.27
	$B_{y\to x}$ and $B_{y\to z}$	3.80	0.19	0.10
	$fluid_z$	5.88	0.40	0.27
	$B_{z\to x}$ and $B_{z\to y}$	3.75	0.18	0.10
	Transposition	1.96	–	–
$1024 \times 1 \times 1$	CFL	0.44	0.61	0.24
	$fluid_x$	2.16	4.90	0.58
	$B_{x\to y}$ and $B_{x\to z}$	0.53	0.69	0.10
	$fluid_y$	11.81	4.09	0.26
	$B_{y\to x}$ and $B_{y\to z}$	7.57	0.66	0.10
	$fluid_z$	11.70	4.03	0.26
	$B_{z\to x}$ and $B_{z\to y}$	7.55	0.80	0.10
	Transposition	4.10	–	–
$2048 \times 1 \times 1$	CFL	0.90	0.62	0.24
	$fluid_x$	4.31	6.09	0.77

Table 4 (*Continued*)

Number of grid points	Operations	FORTRAN/CPU	GTX 295	GTX 480
	$B_{x\to y}$ and $B_{x\to z}$	1.07	0.67	0.10
	$fluid_y$	23.31	4.10	0.26
	$B_{y\to x}$ and $B_{y\to z}$	15.09	0.67	0.10
	$fluid_z$	23.26	4.04	0.26
	$B_{z\to x}$ and $B_{z\to y}$	14.96	0.67	0.10
	Transposition	8.22	–	–
$4096 \times 1 \times 1$	CFL	1.24	0.63	0.28
	$fluid_x$	8.92	7.93	1.19
	$B_{x\to y}$ and $B_{x\to z}$	1.59	0.66	0.10
	$fluid_y$	46.83	4.01	0.26
	$B_{y\to x}$ and $B_{y\to z}$	30.41	0.67	0.10
	$fluid_z$	46.74	3.98	0.26
	$B_{z\to x}$ and $B_{z\to y}$	30.15	0.67	0.10
	Transposition	16.50	–	–

Table 5
Performance comparison (ms/step) between the FORTRAN/CPU and the *GPU-MHD* of major calculations in single precision at different resolutions of the 2D test.

Number of grid points	Operations	FORTRAN/CPU	GTX 295	GTX 480
$128 \times 128 \times 1$	CFL	9.44	0.69	0.30
	$fluid_x$	33.30	4.43	0.79
	$B_{x\to y}$ and $B_{x\to z}$	8.54	0.77	0.11
	$fluid_y$	33.28	4.48	0.83
	$B_{y\to x}$ and $B_{y\to z}$	7.15	0.83	0.12
	$fluid_z$	179.00	4.37	0.57
	$B_{z\to x}$ and $B_{z\to y}$	116.17	0.78	0.11
	Transposition	105.22	–	–
$256 \times 256 \times 1$	CFL	28.26	1.01	0.56
	$fluid_x$	136.87	6.05	1.73
	$B_{x\to y}$ and $B_{x\to z}$	33.41	1.42	0.37
	$fluid_y$	133.76	6.32	1.77
	$B_{y\to x}$ and $B_{y\to z}$	32.61	1.41	0.41
	$fluid_z$	713.28	5.94	1.33
	$B_{z\to x}$ and $B_{z\to y}$	467.13	1.24	0.36
	Transposition	406.36	–	–
$512 \times 512 \times 1$	CFL	108.00	1.90	1.46
	$fluid_x$	513.75	12.21	6.11
	$B_{x\to y}$ and $B_{x\to z}$	121.67	2.86	1.18
	$fluid_y$	503.29	12.73	6.12
	$B_{y\to x}$ and $B_{y\to z}$	120.46	3.53	1.33
	$fluid_z$	2747.38	11.78	5.33
	$B_{z\to x}$ and $B_{z\to y}$	1796.88	2.94	1.16
	Transposition	1554.66	–	–
$1024 \times 1024 \times 1$	CFL	459.22	5.28	3.18
	$fluid_x$	2113.20	36.24	12.80
	$B_{x\to y}$ and $B_{x\to z}$	532.89	9.37	3.08
	$fluid_y$	2078.98	40.44	13.27
	$B_{y\to x}$ and $B_{y\to z}$	534.93	11.89	3.70
	$fluid_z$	11611.11	37.14	10.72
	$B_{z\to x}$ and $B_{z\to y}$	7538.00	9.70	3.05
	Transposition	6537.16	–	–

Table 6
Performance comparison (ms/step) between the FORTRAN/CPU and the *GPU-MHD* of major calculations in single precision at different resolutions of the 3D test.

Number of grid points	Operations	FORTRAN/CPU	GTX 295	GTX 480
$32 \times 32 \times 32$	CFL	14.30	0.78	0.35
	$fluid_x$	76.74	4.90	1.00
	$B_{x\to y}$ and $B_{x\to z}$	23.09	0.97	0.16
	$fluid_y$	74.35	4.95	0.99
	$B_{y\to x}$ and $B_{y\to z}$	22.09	1.04	0.18
	$fluid_z$	74.02	4.97	1.04
	$B_{z\to x}$ and $B_{z\to y}$	22.44	1.00	0.17
	Transposition	125.12	–	–
$64 \times 64 \times 64$	CFL	111.92	1.90	1.43
	$fluid_x$	545.58	11.82	5.57
	$B_{x\to y}$ and $B_{x\to z}$	144.58	2.94	1.18
	$fluid_y$	533.83	12.11	5.59
	$B_{y\to x}$ and $B_{y\to z}$	143.21	3.61	1.31

(*continued on next page*)

Table 6 (*Continued*)

Number of grid points	Operations	FORTRAN/CPU	GTX 295	GTX 480
	$fluid_z$	537.75	12.37	5.90
	$B_{z\to x}$ and $B_{z\to y}$	142.58	3.25	1.27
	Transposition	963.6	–	–
$128 \times 128 \times 128$	CFL	871.00	10.98	5.50
	$fluid_x$	4231.83	63.83	19.64
	$B_{x\to y}$ and $B_{x\to z}$	1047.42	17.30	5.47
	$fluid_y$	4110.67	65.29	18.63
	$B_{y\to x}$ and $B_{y\to z}$	1029.96	22.42	6.45
	$fluid_z$	4096.88	70.65	21.56
	$B_{z\to x}$ and $B_{z\to y}$	1031.92	19.52	6.18
	Transposition	7686.58	–	–

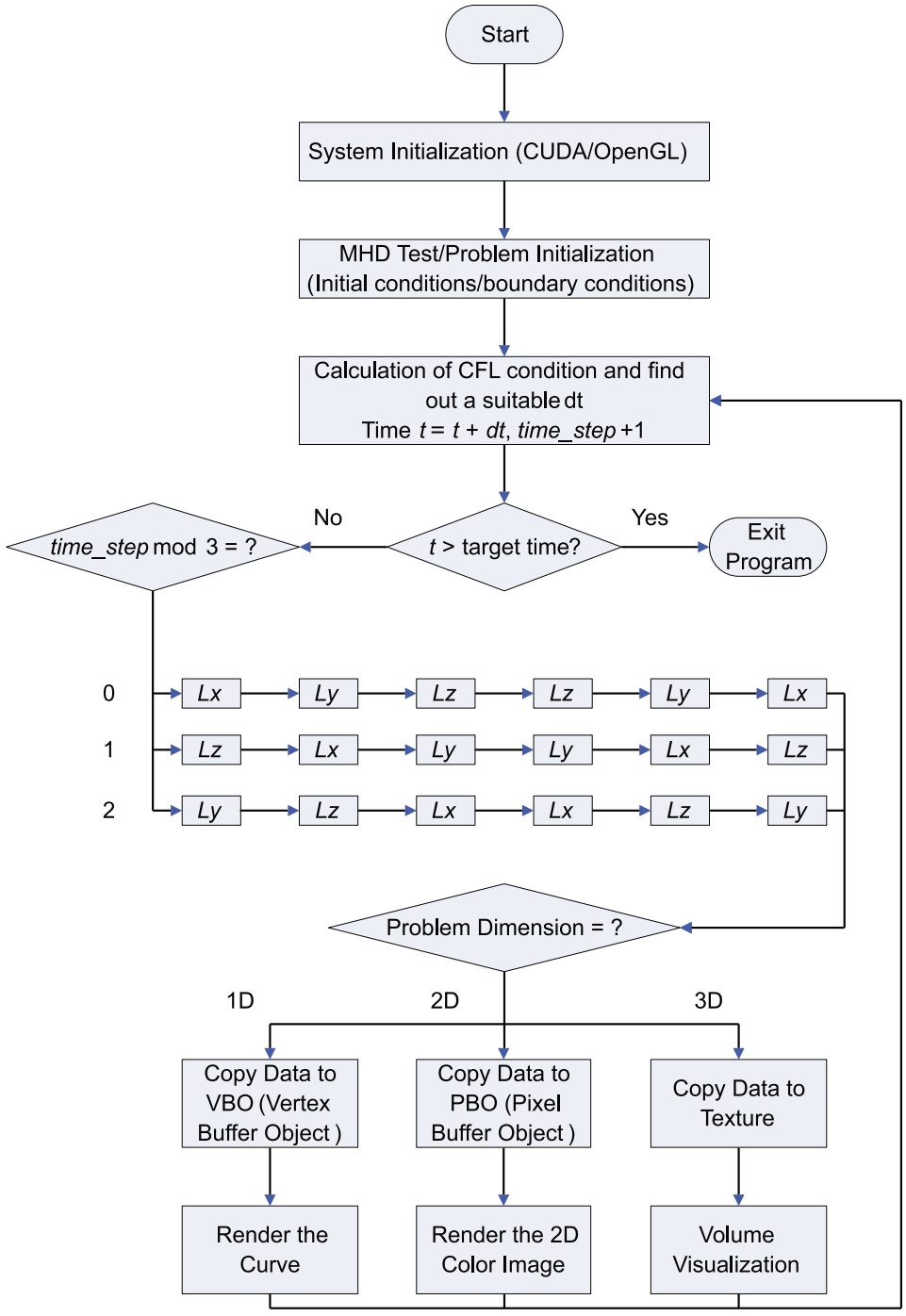

Fig. 24. The flow chat of the extended *GPU-MHD*.

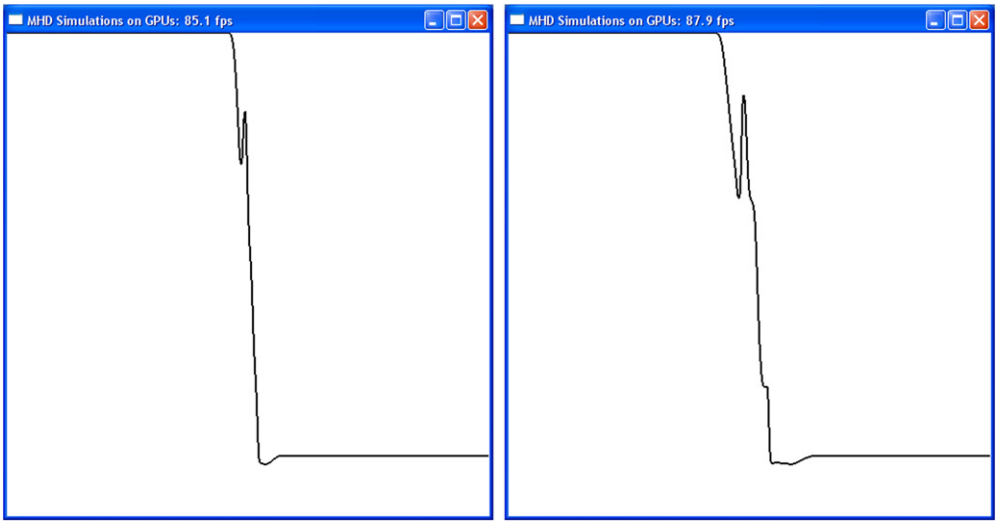

Fig. 25. 1D real-time visualizations of the density (ρ) of the Brio–Wu shock tube problem with 512 grid points using *GPU-MHD*.

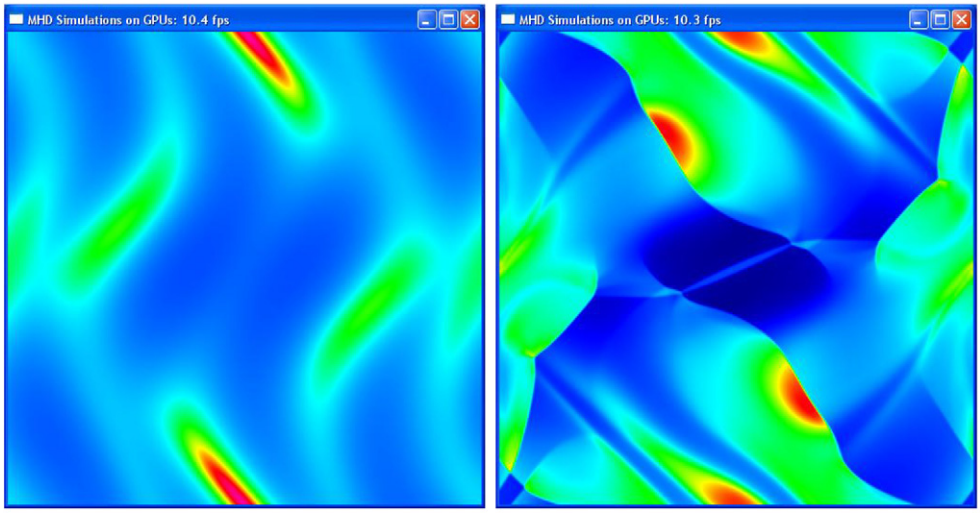

Fig. 26. 2D visualizations of the density (ρ) of the Orszag–Tang vortex problem with 512^2 grid points using *GPU-MHD*.

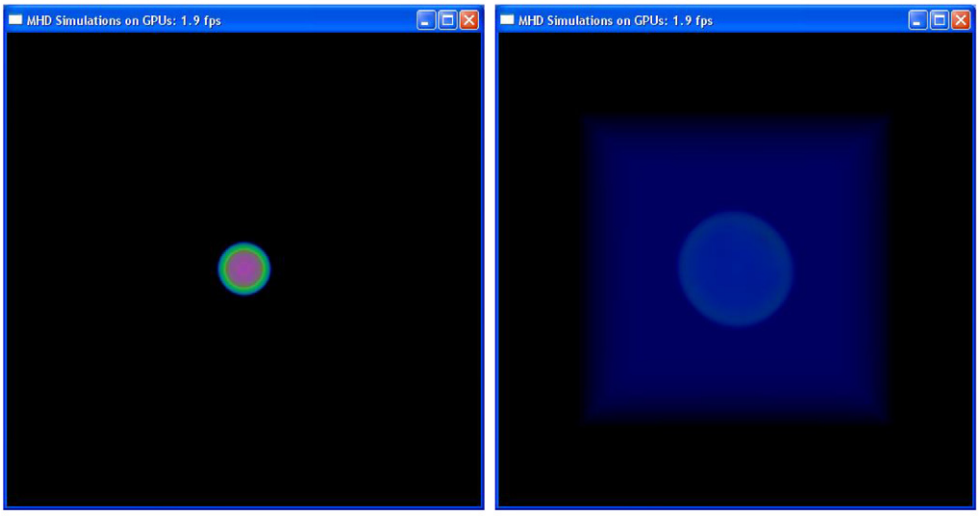

Fig. 27. 3D visualizations of the density (E) of the 3D blast wave problem with 128^3 grid points using *GPU-MHD*.

Acknowledgements

This work is supported by the Science and Technology Development Fund of Macao SAR (03/2008/A1) and the National High-Technology Research and Development Program of China (2010AA122205). Xueshang Feng is supported by the National Natural Science Foundation of China (40874091 and 40890162). The authors would like to thank Dr. Ue-Li Pen and Bijia Pang at the Canadian Institute for Theoretical Astrophysics, University of Toronto for providing the FORTRAN MHD code. Thanks also to Professor Ah Chung Tsoi and Dr. Yuet Ming Lam for their suggestions on the revision of the paper. Special thanks to anonymous reviewers for their constructive and valuable comments that helped us to improve the paper.

References

[1] D.S. Balsara, D.S. Spicer, A staggered mesh algorithm using high order Godunov fluxes to ensure solenoidal magnetic fields in magnetohydrodynamic simulations, J. Comput. Phys. 149 (1999) 270–292.
[2] R.G. Belleman, J. Bédorf, S.F. Portegies Zwart, High performance direct gravitational N-body simulations on graphics processing units II: an implementation in CUDA, New Astronomy 13 (2008) 103–112.
[3] M. Brio, C.C. Wu, An upwind differencing scheme for the equations of ideal magnetohydrodynamics, J. Comput. Phys. 75 (1988) 400–422.
[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, J. Parallel Distrib. Comput. 68 (2008) 1370–1380.
[5] A. Ciardi, S. Lebedev, A. Frank, E. Blackman, D. Ampleford, C. Jennings, J. Chittenden, T. Lery, S. Bland, S. Bott, G. Hall, J. Rapley, F. Vidal, A. Marocchino, 3D MHD simulations of laboratory plasma jets, Astrophysics and Space Science 307 (2007) 17–22.
[6] W. Dai, P.R. Woodward, An approximate Riemann solver for ideal magnetohydrodynamics, J. Comput. Phys. 111 (1994) 354–372.
[7] P. Daum, VisAn MHD: a toolbox in Matlab for MHD computer model data visualization and analysis, Ann. Geophys. 25 (2007) 779–784.
[8] M. Dryer, Multi-dimensional MHD simulation of solar-generated disturbances: space weather forecasting of geomagnetic storms, AIAA J. 36 (1998) 365–370.
[9] M. Dryer, Space weather simulations in 3D MHD from the Sun to Earth and beyond to 100 AU: a modeler's perspective of the present state of the art (Invited Review), Asia J. of Physics 16 (2007) 97–121.
[10] C.R. Evans, J.F. Hawley, Simulation of magnetohydrodynamic flow: a constrained transport method, Astrophys. J. 332 (1988) 659–677.
[11] X. Feng, Y. Zhou, S.T. Wu, A novel numerical implementation for solar wind modeling by the modified conservation element/solution element method, Astrophys. J. 655 (2007) 1110–1126.
[12] T.A. Gardiner, J.M. Stone, An unsplit Godunov method for ideal MHD via constrained transport, J. Comput. Phys. 205 (2005) 509–539.
[13] E. Gaburov, S. Harfst, S.P. Zwart, SAPPORO: a way to turn your graphics cards into a GRAPE-6, New Astronomy 14 (2009) 630–637.
[14] J.P. Goedbloed, S. Poedts, Principles of Magnetohydrodynamics: With Applications to Laboratory and Astrophysical Plasmas, Cambridge University Press, 2004.
[15] M. Hadwiger, J.M. Kniss, C. Rezk-Salama, D. Weiskopf, K. Engel, Real-Time Volume Graphics, A K Peters, 2006.
[16] J.C. Hayes, M.L. Norman, R.A. Fiedler, J.O. Bordner, P.S. Li, S.E. Clark, A. Ud-Doula, M.-M.M. Low, Simulating radiating and magnetized flows in multiple dimensions with ZEUS-MP, Astrophys. J. Supp. 165 (2006) 188–228.
[17] Z. Huang, C. Wang, Y. Hu, X. Guo, Parallel implementation of 3D global MHD simulations for Earth's magnetosphere, Comput. Math. Appl. 55 (2008) 419–425.
[18] S. Jin, Z. Xin, The relaxation schemes for systems of conservation laws in arbitrary space dimensions, Commun. Pure Appl. Math. 48 (1995) 235–276.
[19] R. Käppeli, S.C. Whitehouse, S. Scheidegger, U.-L. Pen, M. Liebendörfer, FISH: a 3D parallel MHD code for astrophysical applications, arXiv:0910.2854v1, 2009.
[20] D.B. Kirk, W.W. Hwu, Programming Massively Parallel Processors, Addison–Wesley, 2010.
[21] A.G. Kulikovskii, N.V. Pogorelov, A.Yu. Semenov, Mathematical Aspects of Numerical Solution of Hyperbolic Systems, CRC Press, 2001.
[22] D. Lee, A.E. Deane, A parallel unsplit staggered mesh algorithm for magnetohydrodynamics, in: A. Deane, et al. (Eds.), Parallel Computational Fluid Dynamics — Theory and Applications, Elsevier, 2006, pp. 243–250.
[23] D. Lee, A.E. Deane, An unsplit staggered mesh scheme for multidimensional magnetohydrodynamics, J. Comput. Phys. 228 (2009) 952–975.
[24] R.J. LeVeque, Numerical Methods for Conservation Laws, second edition, Birkhauser, 1992.
[25] R.J. LeVeque, Finite Volume Methods for Hyperbolic Problems, Cambridge University Press, 2002.
[26] S. Li, H. Li, A novel approach of divergence-free reconstruction for adaptive mesh refinement, J. Comput. Phys. 199 (2004) 1–15.
[27] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: a unified graphics and computing architecture, IEEE Micro 28 (2008) 39–55.
[28] W.R. Mark, R.S. Glanville, K. Akeley, M.J. Kilgard, Cg: a system for programming graphics hardware in a C-like language, ACM Trans. Graph. 22 (2003) 896–907.
[29] P.M. Meijer, S. Poedts, J.P. Goedbloed, Parallel magnetohydrodynamics on the Cray T3D, Future Generation Computer Systems 12 (1996) 307–323.
[30] NVIDIA Corporation, NVIDIA CUDA compute unified device architecture programming Guide 3.2, Oct. 2010, http://www.nvidia.com/object/cuda_develop.html.
[31] NVIDIA Corporation, CUDA C programming best practices guides 3.2, August 2010, http://www.nvidia.com/object/cuda_develop.html.
[32] NVIDIA Corporation, NVIDIA's next generation CUDA compute architecture: Fermi, Whitepaper, 2009, http://www.nvidia.com/object/cuda_develop.html.
[33] A. Orszag, C.M. Tang, Small-scale structure of two-dimensional magnetohydrodynamics turbulence, J. Fluid Mech. 90 (1979) 129–143.
[34] B. Pang, U.-L. Pen, E.T. Vishniac, Fast magnetic reconnection in three-dimensional magnetohydrodynamics simulations, Physics of Plasmas 17 (2010) 102302.
[35] U.-L. Pen, P. Arras, S. Wong, A free, fast, simple and efficient TVD MHD code, Astrophys. J. Supp. 149 (2003) 447–455.
[36] U.-L. Pen, C.D. Matzner, S. Wong, The fate of nonradiative magnetized accretion flows: Magnetically frustrated convection, Astrophys. J. 596 (2003) L207–L210.
[37] U.-L. Pen, et al., MHD code: http://www.cita.utoronto.ca/~pen/MHD/.
[38] D. Ryu, T.W. Jones, Numerical magnetohydrodynamics in astrophysics: algorithm and tests for one-dimensional flow, Astrophys. J. 442 (1995) 228–258.
[39] J. Sainio, CUDAEASY — a GPU accelerated cosmological lattice program, Comput. Phys. Commun. 181 (2010) 906–912.
[40] J. Sanders, E. Kandrot, CUDA by Example, Addison–Wesley, 2011.
[41] O. Schenk, M. Christen, H. Burkhart, Algorithmic performance studies on graphics processing units, J. Parallel Distrib. Comput. 68 (2008) 1360–1369.
[42] H.-Y. Schive, C.-H. Chien, S.-K. Wong, Y.-C. Tsai, T. Chiueh, Graphic-card cluster for astrophysics (GraCCA) — performance tests, New Astronomy 13 (2008) 418–435.
[43] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, Larrabee: a many-core x86 architecture for visual computing, ACM Trans. Graph. 27 (2008), Article 18.
[44] G. Sod, A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws, J. Comput. Phys. 27 (1978) 1–31.
[45] Spherical blast wave test page of Athena3D test suite: http://www.astro.virginia.edu/VITA/ATHENA/blast.html.
[46] G. Stantchev, W. Dorland, N. Gumerov, Fast parallel particle-to-grid interpolation for plasma PIC simulations, J. Parallel Distrib. Comput. 68 (2008) 1339–1349.
[47] G. Stantchev, D. Juba, W. Dorland, A. Varshney, Using graphics processors for high-performance computation and visualization of plasma turbulence, Computing in Science and Engineering 11 (2009) 52–59.
[48] J. Stone, T.A. Gardiner, Recent progress in astrophysical MHD, Comput. Phys. Commun. 177 (2007) 257–259.
[49] J. Stone, T.A. Gardiner, P. Teuben, J.F. Hawley, J.B. Simon, Athena a new code for astrophysical MHD, Astrophys. J. Supp. 178 (2008) 137–177.
[50] J. Stone, T. Gardiner, A simple unsplit Godunov method for multidimensional MHD, New Astronomy 14 (2009) 139–148.
[51] G. Strang, On the construction and comparison of difference schemes, SIAM J. Numer. Anal. 5 (1968) 506–517.
[52] G. Tóth, D. Odstrčil, Comparison of some flux corrected transport and total variation diminishing numerical schemes for hydrodynamic and magnetohydrodynamic problems, J. Comput. Phys. 128 (1996) 82–100.
[53] G. Tóth, The $\nabla \cdot \mathbf{B} = 0$ constraint in shock-capturing magnetohydrodynamics codes, J. Comput. Phys. 161 (2000) 605–656.
[54] H. Trac, U.-L. Pen, A primer on Eulerian computational fluid dynamics for astrophysics, Publications of the Astronomical Society of the Pacific 115 (2003) 303–321.
[55] B. van Leer, Towards the ultimate conservative difference scheme II. Monotonicity and conservation combined in a second order scheme, J. Comput. Phys. 14 (1974) 361–370.
[56] B. van Leer, Towards the ultimate conservative difference scheme V. A second-order sequel to Godunov's method, J. Comput. Phys. 32 (1979) 101–136.
[57] R.S. Wright Jr., B. Lipchak, N. Haemel, OpenGL Superbible: Comprehensive Tutorial and Reference, fourth edition, Addison–Wesley, 2007.
[58] A. Zachary, A. Malagoli, P. Colella, A higher-order Godunov method for multidimensional ideal magnetohydrodynamics, SIAM J. Sci. Comput. 15 (1994) 263–284.
[59] U. Ziegler, The NIRVANA code: parallel computational MHD with adaptive mesh refinement, Comput. Phys. Commun. 179 (2008) 227–244.