BULLETIN OF THE POLISH ACADEMY OF SCIENCES
TECHNICAL SCIENCES, Vol. 61, No. 1, 2013
DOI: 10.2478/bpasts-2013-0023

VARIA

A new CUDA-based GPU implementation of the two-dimensional Athena code

A. WASILJEW and K. MURAWSKI*

Faculty of Physics, Mathematics and Informatics, University of Maria Curie-Skłodowska,
1 M. Curie-Skłodowskiej Sq., 20-031 Lublin, Poland

Abstract. We present a new version of the Athena code, which solves magnetohydrodynamic equations in two-dimensional space. This new implementation, which we have named Athena-GPU, uses CUDA architecture to allow the code execution on Graphical Processor Unit (GPU). The Athena-GPU code is an unofficial, modified version of the Athena code which was originally designed for Central Processor Unit (CPU) architecture.

We perform numerical tests based on the original Athena-CPU code and its GPU counterpart to make a performance analysis, which includes execution time, precision differences and accuracy. We narrowed our tests and analysis only to double precision floating point operations and two-dimensional test cases. Our comparison shows that results are similar for both two versions of the code, which confirms correctness of our CUDA-based implementation. Our tests reveal that the Athena-GPU code can be 2 to 15-times faster than the Athena-CPU code, depending on test cases, the size of a problem and hardware configuration.

Key words: CUDA-based GPU implementation, two-dimensional Athena code, magnetohydrodynamic equations.

1. Introduction

High resolution numerical methods for solving equations of magnetohydrodynamics (MHD) are computationally expensive due to complexity of these equations. Some of the MHD problems may even require more sophisticated approach as for example with two-component plasma [1]. Despite these difficulties there are a number of implementations for solving MHD equations, such as in Zeus [2], Flash [3], Pluto [4], Nirvana [5], Surya [6], and Athena [7] codes. Most of them use MPI standard [8] to make parallel execution feasible. In such case a simulation domain is divided into patches, each of which is treated separately, being assigned to a different node (processor). Each node performs computations on its own domain. One of the nodes synchronizes execution and gather results from other nodes. This common approach is often used by multiprocessor platforms, supercomputers or clusters. Although these computing systems are very powerful, they are also very expensive.

With the introduction of NVIDIA's Compute Unified Device Architecture (CUDA) graphic processors became good alternatives for expensive general purpose computations. CUDA allows to perform computations on graphic processors without using any graphic pipeline. This approach brings graphic processor unit (GPU) computational power to the scientific community who is not familiar with graphic pipeline programming. Over the past few years GPUs became very fast as far as their theoretical performance in arithmetic operations per second is concerned. It is showed by NVIDIA that a modern GPU is much more powerful (over 1 TFlops/s) than a CPU (about 130 GFlops/s), with overwhelming theoretical memory bandwidth (over 160 GB/s) [9].

Graphic processors are highly specialized in processing large amount of data in parallel, which requires that the task for a GPU must be highly parallelized to optimize its performance. As a result, not each problem can be efficiently implemented on graphic processors. Finite-volume numerical schemes, devoted to solving fluid equations, seem to be good candidates for a GPU task [10]. These schemes often use large amount of data at the grid cells, and values in each cell can be updated in time. There are a few known examples of MHD codes which use GPU/CUDA [11, 12].

We aim to implement a GPU support with C for CUDA into the existing Athena-CPU code [7]. Our paper is structured as follows. In the next section we overview the existing Athena-CPU code [13]. In Sec. 3 we describe a new version of the code that is implemented in C for CUDA. We conclude this paper by a presentation of the main results in Sec. 4.

2. General characteristic of the existing Athena-CPU code

The Athena-CPU code [14] solves MHD equations which we write here for the ideal case in the following form:

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho\mathbf{v}\\ \mathbf{B}\\ E\end{bmatrix}+\nabla\cdot\begin{bmatrix}\rho\mathbf{v}\\ \rho\mathbf{v}\mathbf{v}+I\left(\left(p+\dfrac{\mathbf{B}^2}{2}\right)-\mathbf{B}\mathbf{B}\right)\\ \mathbf{v}\mathbf{B}-\mathbf{B}\mathbf{v}\\ \left(E+p+\dfrac{\mathbf{B}^2}{2}\right)\mathbf{v}-\dfrac{1}{\mu}\mathbf{B}\left(\mathbf{v}\cdot\mathbf{B}\right)\end{bmatrix}=0. \tag{1}$$

*e-mail: kmur@kft.umcs.lublin.pl

Here ρ is mass density, $\mathbf{v}$ denotes the flow velocity, p is the gas pressure, $\mathbf{B}$ is the magnetic field which must satisfy the solenoidal constraint, $\nabla \cdot \mathbf{B} = 0$, μ is magnetic field permeability and

$$E = \frac{p}{\gamma - 1} + \frac{\rho \mathbf{v}^2}{2} + \frac{\mathbf{B}^2}{2\mu} \tag{2}$$

is a total energy density, with γ denoting the specific heat ratio. We set and hold fixed $\gamma = 5/3$.

Note that electric field $\mathcal{E} = -\mathbf{v} \times \mathbf{B}$, enters the induction equation as

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \times \mathcal{E} = 0. \tag{3}$$

The Athena-CPU code adopts combination of Corner Transport Upwind (CTU) algorithm with Constrained Transport (CT) [14]. The former is used to update cell averaged values while the latter is implied to correct and update magnetic field components to satisfy the solenoidal condition. This condition is crucial for all MHD numerical schemes [15]. The CTU algorithm was devised as a two-dimensional (2D) variant of Piecewise Parabolic Method (PPM) of Collela and Woodward [16]. Conserved values in each cell are first updated to a half time-step. Then these new values are used to calculate the corrected numerical fluxes, which are adopted to update cell averaged components to a full time-step. For simplicity reasons we will further refer to cell averaged components as the cell centered values. This scheme is well documented by the authors of the original Athena code [13]. As we were porting the existing code for CUDA architecture, we briefly report here on the CPU implementation.

The Athena-CPU code is organized as shown in Fig. 1. The core of the code is the main loop, which is marked by the box with dashed line. As we are adopting a 2D version of the Athena 3.1 code, the most important is a one time-step integration function, called `integrate2d()`. This is the most computationally expensive function which must be used for iteration over all cells within the computational box. The block in Fig. 1, containing the `integrate2d()` function, is named "Main integration". The input data for this block is the actual state of conserved variables in each cell at a given time-step and the output specifies the updated state for each cell at a new time-step.

The initial data is prepared and stored in the `Grid` structure. Cell centered conserved values are stored in the array of the `Cons1D` structure. This array is located under address pointed by `U` component of `Grid`. Values for magnetic field components located on interfaces between two neighbouring grid cells are stored in separate arrays pointed by `B1i` and `B2i` variables within the `Grid` structure.

In the first two steps of the `integrate2d()` function left and right states of the conserved state are computed at interfaces. These states are stored in 2D arrays named `Ul_x1Face`, `Ur_x1Face`, `Ul_x2Face`, and `Ur_x2Face`. There are few methods available for evaluating the interface states in the Athena-CPU code. We mention the second-order piecewise linear method (PLM) as it is the only choice in the Athena-GPU code. The PLM method uses linear interpolation in the primitive variables for spatial reconstruction. The magnetic field components, located at interfaces, are stored in temporary arrays pointed by `B1_x1Face` and `B2_x2Face`. Left and right states as well as magnetic field components are used to compute the one-dimensional fluxes in the x- and y-directions. These fluxes are stored in 2D arrays `x1Flux` and `x2Flux`. Integration over domain cells is performed twice for each flux, which has negative impact on a performance of the code. Additionally, this code requires a lot of intermediate data allocated in 2D arrays, which may result in some problems while porting this code for a GPU.

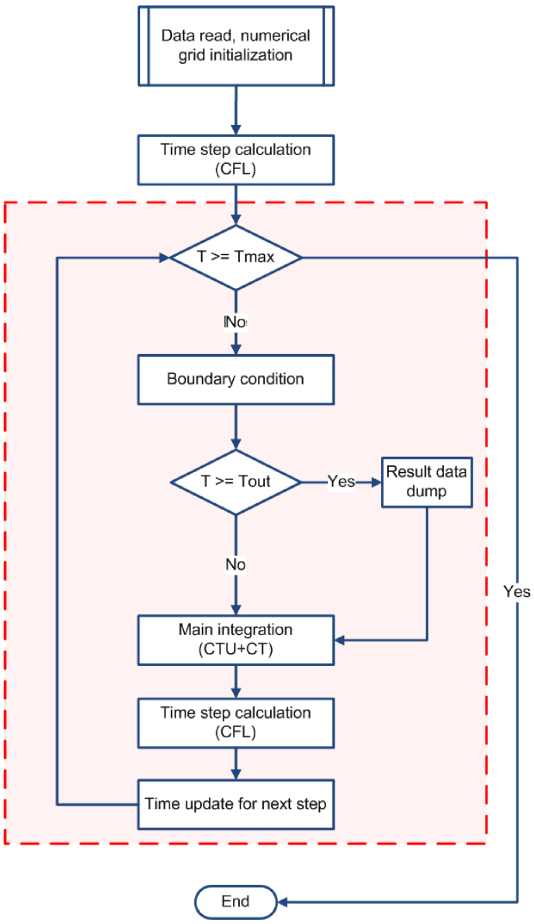

Fig. 1. Simplified flow chart of the Athena-CPU code

In the next two steps the z-component of electromagnetic field is evaluated and then the magnetic fields components `B1_x1Face` and `B2_x2Face` are updated, using the CT scheme for a half time-step [13]. This needs the additional intermediate step, which integrates $\mathcal{E}_z$ from cell-centered to corner-centered quantities. This step requires two additional 2D temporary arrays pointed by `emf3_cc` (cell-centered) and `emf3` (corner-centered) variables. These steps are implemented by 3 separate loops, iterating through all numerical cells.

Steps 5 and 6 of the main integration method advance left and right states of the conserved variables, stored in arrays `Ul_x1Face`, `Ur_x1Face`, `Ul_x2Face`, and `Ur_x2Face`, by a half time-step, using the `x1Flux` and `x2Flux` fluxes, which

we were already evaluated in the first two steps. In each step there is also the additional loop which adds source terms to conservative fluxes along the y- and x-directions. Note that we do not treat the gravity source term in the Athena-GPU.

Step 7 is only needed by the CT algorithm to integrate emf to a corner in one of the final steps. It is implemented as one separate loop over numerical grid, which evaluates a cell centered value of $\mathcal{E}_z$, stored back in the `emf3_cc` array.

Step 8 recomputes the fluxes `x1Flux` and `x2Flux` for which corrected left- and right-states of the conserved quantities are used. These quantities are evaluated during steps 5 and 6. This part of the code is implemented by two separate loops over all cells. In the original Athena-CPU code, there are a few different solvers for computing the fluxes. However, in the Athena-GPU code we limit ourselves to the Roe solver [17]. When the Roe solver fails, the fluxes are computed using HLLE solver [18].

Step 9 is simply implemented by two loops iterating over each numerical cell, integrating electromagnetic field to cell corners, and then updating magnetic field components for a full time-step with the use of the CT scheme.

The following step is used to update cell centered conserved variables using fluxes evaluated in step 8. These values are stored back in the `Grid` structure. Last step is adopted to cell centered magnetic field components, using updated values from step 9. Final steps are implemented by 3 separate loops, iterating over the whole numerical grid.

After updating the conserved quantities, a time-step is evaluated from the CFL condition (Fig. 1). Before the integration function is called, the code sets boundary conditions along all edges of the simulation region. These conditions are realized by setting values of $\rho, \mathbf{v}, p, \mathbf{B}$ within ghost cells. A user can implement open, periodic and reflected conditions as well as specifies his/her own boundary conditions.

Note that there are many temporary arrays which must be allocated before main integration loop starts. These arrays need to have the same dimensions as numerical grid which may result in large memory consumption. There are also a lot of intermediate steps, which are implemented in separate loops. Each loop iterates over a numerical grid, next loop could not start until previous is working. In the next section we show that this consists a challenge while porting a code for CUDA architecture.

3. GPU implementation into the Athena code

Athena-GPU code is not a part of official Athena code release. That is why some functionalities such as different types of spatial reconstruction, switches for floating point precision in the code and gravity are not used. We selected basic code configuration which simplified porting the Athena-CPU code for a GPU and it included:

1. double precision floating point arithmetic
2. disabled H-correction
3. only MHD is dealt with
4. adiabatic equation of state
5. Roe's and HLLE Riemann solvers
6. second order spatial reconstruction (PLM)
7. gravity-free case
8. 2D numerical grid
9. no MPI is used

The very first step when implementing a grid-based code for a GPU is to look at structures which preserve all the data required for the computation. One of the most important task is to prepare such C structures that can be allocated in graphic card memory. We also need to copy data between CPU host and GPU device memories. The Athena-CPU code uses extensively multi-dimensional arrays within data structures, while the CUDA architecture adopts only one-dimensional arrays. To solve this issue we implemented new data structure, called `Grid_gpu`. This structure contains only 1D arrays pointed by its components `B1i`, `B2i`, `B3i`, and `U`. These components represent the same data as in the `Grid` structure, but the memory alignment was flattened to one dimension. Thus the `Grid_gpu` structure can be stored in the graphic card memory. We implemented new methods to copy data between `Grid` and `Grid_gpu`, which convert multi-dimensional arrays from `Grid` to one-dimensional arrays in `Grid_gpu`. We also introduced new functions which can allocate device memory for a numerical grid and other working variables.

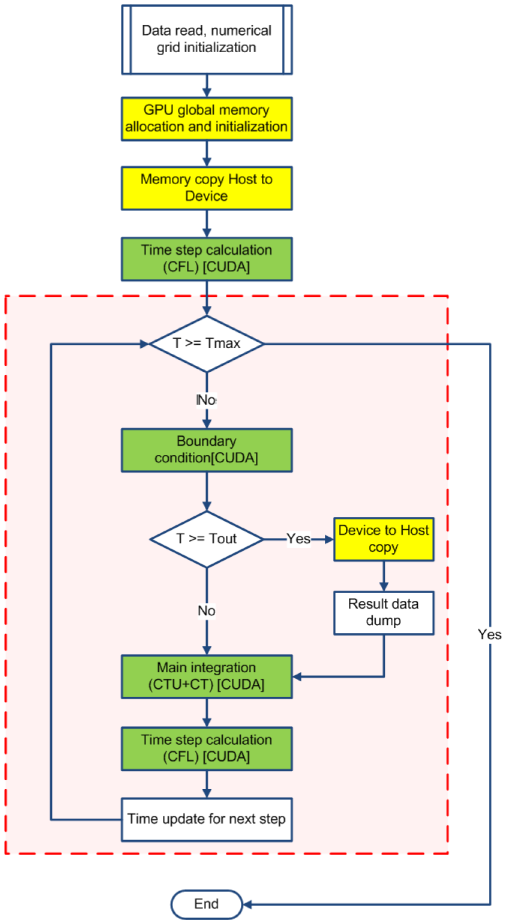

Fig. 2. A simplified flow chart of the Athena-GPU code. The main loop of the code is denoted by dashed line. Green blocks correspond to functions which are executed on GPUs

The next step, while porting any existing CPU code for a GPU architecture, is to identify parts of the CPU code, which are computationally expensive and should be reimplemented for a GPU. Figure 2 illustrates a simplified flow chart of the Athena-GPU code in which some computations are mapped on a GPU. The operations performed on a GPU and the new function which evaluates a time-step, written in C for CUDA, are marked by green blocks. Some parts of the code which are denoted by yellow blocks are GPU specific operations which do not exist in the Athena-CPU code. The remnant parts of the Athena-CPU code were left intact. The `Grid` structure is taken from the Athena-CPU implementation and is used for a computational domain setting.

At the beginning of the Athena-GPU code we set two additional steps. After setting up a numerical grid and initial data stored in host memory, we allocate the device memory and make a copy of `Grid`. From this stage all calculations are performed by a GPU. Because of high latencies, which exert a strong impact on the code performance, it is essential to make as few memory transfer operations between the host and device as possible. That is why we only transfer data from the device to host, when we need to store intermediate simulation results at given time intervals.

The Athena-GPU implementation is based on a set of different kernel functions. Those functions are enclosed by ordinary C functions, which are used in parts of the Athena-CPU code. To launch the kernel functions we used one-dimensional threads blocks of constant size `BLOCK_SIZE` that is specified at the compilation stage. The total number of blocks is evaluated with the use of a grid size. In an algorithm which is based on a finite-volume method the plasma quantities can be updated independently in every numerical cell. We make each thread in a given kernel working in a particular numerical cell. Some kernels work on entire 2D grid and some of them work only on a single row or column. This requires the index of a grid element on which each thread is working on. Depending on the dimension of processed data we implemented two types of indexes which are calculated according to listing 1.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
if(i < is || i > ie) return;
```

```
int ind = (blockIdx.x * blockDim.x) + threadIdx.x;
int j = ind / sizex;
int i = ind % sizex;
if(i < is || i > ie || j < js || j > je) return;
```

Listing 1: Index evaluation for 1D (top) and 2D (bottom) cases

Variables `blockIdx.x`, `blockDim.x` and `threadIdx.x` are predefined in C for CUDA and `sizex` is the number of cells in the x-direction. Variables `is`, `ie`, `js` and `je` are used for setting limits of cell indexes, which are necessary to prevent accessing memory from outside of the allocated computational grid. Index evaluation is fairly simple, but accessing global memory in a non-coalesced way decreases the performance. Our implementation stands as the simplest solution and it could be further optimized.

The core of the Athena-CPU code is the `integrate2d()` function which consumes most of CPU time. In the GPU implementation we renamed this function by `integrate_2d_cuda()` and mapped it on a GPU. As in the original code, the first two steps evaluate left- and right-states at cell interfaces. These states are stored in 2D arrays named `Ul_x1Face_dev`, `Ur_x1Face_dev`, `Ul_x2Face_dev` and `Ur_x2Face_dev`. Suffix `_dev` indicates that these arrays are allocated in device memory. They have the same size as the computational grid and are stored in memory as one-dimensional memory blocks. The most important function at this stage is `lr_states_cu_1b_dev()` (listing 2). This is the kernel function enclosed within a loop iterating over all rows (columns) of the numerical grid. Each thread in this kernel evaluates left- and right-states in primitive variables at a given cell. Cell index is evaluated based on kernel execution configuration. The main body of this function is essentially the same as in the Athena-CPU code, starting from step 2 to 9. In step 2, the eigenvalues in primitive variables must be calculated by device function named `esys_prim_adb_mhd_cu_dev()` (Listing 3). This function has the same implementation as in the Athena-CPU code, but has slightly different parameters list in declaration.

```
__global__ void lr_states_cu_1b_dev(Prim1D* W, Real* Bxc,
              Real dt, Real dtodx, int is, int ie, int j↩
                  , int sizex,
              Prim1D* Wl, Prim1D* Wr, Real Gamma)
{
  //Cell index evaluation
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if((i < is) || (i > ie)) return;
  i = j*sizex+i;
  ...
  // Define working pointers used further in calculations
  Real* pW_dev = (Real*)&(W[i]); //W[i]
  Real* pW_dev_1 = (Real*)&(W[i-1]); //W[i-1]
  Real* pW_dev_2 = (Real*)&(W[i+1]); //W[i+1]

  /*---- Step 1. ↩
       ----------------------------------------------
   * Compute eigensystem in primitive variables.
   * Below is the device function, compiled as inline! ↩
       */

  esys_prim_adb_mhd_cu_dev(W[i].d,W[i].Vx,W[i].P,
    Bxc[i],W[i].By,W[i].Bz,ev,rem,lem, Gamma);

  /*---- Step 2. to Step 9. as in the Athena-CPU code ↩
       ----------
   */
}
```

Listing 2: The `lr_states_cu_1b_dev()` function

```
__device__ void esys_prim_adb_mhd_cu_dev(const Real d,
  const Real v1, const Real p,
  const Real b1, const Real b2, const Real b3,
  Real *eigenvalues,
  Real *rem, Real *lem, Real Gamma)
```

Listing 3: Device function declaration, `esys_prim_adb_mhd_cu_dev()`

Steps 3 and 4 of the original algorithm are implemented by a set of four kernel functions. The first is `emf_3_dev()` which calculates cell centered values of the z-component of electromagnetic field and stores the result in the `emf3_cc_dev`

array. Here we used shared memory for transferring the Gas data from the global memory (listing 4). Then each access to U_shared[threadIdx.x] is faster and at the end the result can be written back to the global memory.

```
__global__ void emf_3_dev(Real *emf3_cc_dev, Gas *U, int
    is, int ie, int js, int je, int sizex)
{
  //Evaluate index
  int ind = (blockIdx.x * blockDim.x) + threadIdx.x;
  int j = ind / sizex;
  int i = ind % sizex;

  if(i < is || i > ie || j < js || j > je) return;

  __shared__ Gas U_shared[BLOCK_SIZE];
  U_shared[threadIdx.x] = U[ind];

  // Use U_shared[threadIdx.x] for evaluation of
      emf3_cc_dev[j*sizex+i]
  ...
}
```

Listing 4: The emf_3_dev() function – shared memory usage

Step 4 in the Athena-CPU code is implemented by the integrate_emf3_corner() function and another three loops, which integrate electromagnetic field to the grid cell corners and then update the interface magnetic fields, using CT for a half time-step. In the Athena-GPU code we implemented respectively the following functions: integrate_emf3_corner_dev(), updateMagneticField_4a_dev, updateMagneticField_4b_dev, updateMagneticField_4c_dev, which evaluate B1_x1Face_dev and B2_x2Face_dev. Their implementation is straightforward and requires only appropriate cells indexing.

The next two steps correct transverse flux gradients and add MHD source terms, one for the x-direction and the other for the y-direction. Kernel functions are basically similar in each direction. The function correctTransverseFluxGradients_dev() corrects fluxes and uses shared memory optimization. We limit the number of read operation from the global memory by using shared memory blocks as presented in Listing 5. Each block has its own shared memory block and each thread uses one cell of the shared memory, indexed by threadIdx.x.

```
__global__ void correctTransverseFluxGradients_dev(Cons1D
    *Ul_x1Face_dev,
  Cons1D *Ur_x1Face_dev, Cons1D *x2Flux_dev,
  int is, int ie, int js, int je,
  int sizex, Real hdtodx2, Real hdtodx1)
{
  // Index evaluation
  int ind = (blockIdx.x * blockDim.x) + threadIdx.x;
  int j = ind / sizex;
  int i = ind % sizex;
  if(i < is || i > ie || j < js || j > je) return;

  __shared__ Cons1D x2Flux_dev_shared[BLOCK_SIZE];
  __shared__ Cons1D x2Flux_dev_shared_1[BLOCK_SIZE];
  __shared__ Cons1D x2Flux_dev_shared_2[BLOCK_SIZE];
  __shared__ Cons1D x2Flux_dev_shared_3[BLOCK_SIZE];
  __shared__ Cons1D Ul_x1Face_dev_shared[BLOCK_SIZE];
  __shared__ Cons1D Ur_x1Face_dev_shared[BLOCK_SIZE];

  // Load data to shared memory
  x2Flux_dev_shared[threadIdx.x] = x2Flux_dev[ind+sizex
      -1];
  x2Flux_dev_shared_1[threadIdx.x] = x2Flux_dev[ind-1];
  x2Flux_dev_shared_2[threadIdx.x] = x2Flux_dev[ind+sizex
      ];
  x2Flux_dev_shared_3[threadIdx.x] = x2Flux_dev[ind];
  Ul_x1Face_dev_shared[threadIdx.x] = Ul_x1Face_dev[ind];
  Ur_x1Face_dev_shared[threadIdx.x] = Ur_x1Face_dev[ind];
  ...
  // Perform calculations using index threadIdx.x in
      shared block
  ...
  // Store back to global memory
  Ul_x1Face_dev[ind] = Ul_x1Face_dev_shared[threadIdx.x];
  Ur_x1Face_dev[ind] = Ur_x1Face_dev_shared[threadIdx.x];
}
```

Listing 5: The correctTransverseFluxGradients_dev() function – shared memory usage

MHD source terms are added in the addMHDSourceTerms_dev() function, where we also used shared memory defined in Listing 6. Because there is a limited shared memory amount for each thread block (depending on hardware capabilities) we need to set BLOCK_SIZE carefully, not to exceed physical limits for the target graphic card.

```
__global__ void addMHDSourceTerms_dev(Cons1D *
    Ul_x1Face_dev,
    Cons1D *Ur_x1Face_dev,
    Gas *U, Real *B1i, int is, int ie, int js, int je,
    int sizex, Real hdtodx2, Real hdtodx1)
{
  // Evaluate index
  int ind = (blockIdx.x * blockDim.x) + threadIdx.x;
  int j = ind / sizex;
  int i = ind % sizex;

  if(i < is || i > ie || j < js || j > je) return;

  //Declare shared block
  __shared__ Real dbx[BLOCK_SIZE];
  __shared__ Real B1[BLOCK_SIZE];
  __shared__ Real B2[BLOCK_SIZE];
  __shared__ Real B3[BLOCK_SIZE];
  __shared__ Real V3[BLOCK_SIZE];
  __shared__ Gas U_shared[BLOCK_SIZE];
  __shared__ Gas U_shared_1[BLOCK_SIZE];
  __shared__ Cons1D Ul_x1Face_dev_shared[BLOCK_SIZE];
  __shared__ Cons1D Ur_x1Face_dev_shared[BLOCK_SIZE];

  //Load shared data
  U_shared[threadIdx.x] = U[ind-1];
  U_shared_1[threadIdx.x] = U[ind];
  Ul_x1Face_dev_shared[threadIdx.x] = Ul_x1Face_dev[ind];
  Ur_x1Face_dev_shared[threadIdx.x] = Ur_x1Face_dev[ind];

  // Perform calculations using shared data blocks
  ...

  // Store data to global memory
  Ul_x1Face_dev[ind] = Ul_x1Face_dev_shared[threadIdx.x];
  Ur_x1Face_dev[ind] = Ur_x1Face_dev_shared[threadIdx.x];
}
```

Listing 6: Shared memory blocks in the addMHDSource Terms_dev() kernel function – shared memory usage

Step 7 is implemented in the Athena-GPU by the two separate kernel functions named dhalf_init_dev() and cc_emf3_dev() which replace four loops from the Athena-CPU code. Both of them evaluate emf3_cc_dev just as in the Athena-CPU is evaluated emf3_cc. It is noteworthy that in case of the cc_emf3_dev() function we use device shared memory (listing 7).

```
__global__ void cc_emf3_dev(Real* dhalf_dev, Cons1D *↩
    x1Flux_dev,
    Cons1D *x2Flux_dev, Real *B1_x1Face_dev, Real *↩
        B2_x2Face_dev,
    Real *emf3_cc_dev, Grid_gpu *pG, Gas *U,
    int is, int ie, int js, int je, int sizex,
    Real hdtodx1, Real hdtodx2)
{
  // Evaluate index
  int ind = (blockIdx.x * blockDim.x) + threadIdx.x;
  int j = ind / sizex;
  int i = ind % sizex;

  /* Check bounds */
  if(i < is || i > ie || j < js || j > je) return;

  // Define shared blocks
  __shared__ Real d[BLOCK_SIZE];
  __shared__ Real M1[BLOCK_SIZE];
  __shared__ Real M2[BLOCK_SIZE];
  __shared__ Real B1c[BLOCK_SIZE];
  __shared__ Real B2c[BLOCK_SIZE];
  __shared__ Cons1D x1Flux_dev_shared[BLOCK_SIZE];
  __shared__ Cons1D x1Flux_dev_shared_1[BLOCK_SIZE];
  __shared__ Cons1D x2Flux_dev_shared[BLOCK_SIZE];
  __shared__ Cons1D x2Flux_dev_shared_1[BLOCK_SIZE];
  __shared__ Gas U_shared[BLOCK_SIZE];

  // Load shared memory
  x1Flux_dev_shared[threadIdx.x] = x1Flux_dev[ind];
  x1Flux_dev_shared_1[threadIdx.x] = x1Flux_dev[ind+1];
  x2Flux_dev_shared[threadIdx.x] = x2Flux_dev[ind];
  x2Flux_dev_shared_1[threadIdx.x] = x2Flux_dev[ind+sizex↩
      ];
  U_shared[threadIdx.x] = U[ind];
  d[threadIdx.x] = dhalf_dev[ind];

  // Perform calculations
  ...

  // Store back result to global memory emf3_cc_dev[ind]
  ...
}
```

Listing 7: Shared memory blocks in the `cc_emf3_dev()` kernel function

In step 8 we calculate fluxes from corrected left- and right-states from previous steps. We implemented two similar kernel functions `Cons1D_to_Prim1D_Slice1D_8b()` and `Cons1D_to_Prim1D_Slice1D_8c()` to perform this task (listing 8). Both of them use the device function, `flux_roe_cu_dev()`, to perform flux evaluation. The results are stored in `x1Flux_dev` and `x2Flux_dev`.

```
__global__ void Cons1D_to_Prim1D_Slice1D_8b(Cons1D *↩
    Ul_x1Face_dev,
    Cons1D *Ur_x1Face_dev, Prim1D *Wl_dev, Prim1D *Wr_dev↩
        ,
    Real *B1_x1Face_dev, Cons1D *x1Flux_dev,
    int is, int ie, int js, int je, int sizex,
    Real Gamma_1, Real Gamma_2) {

  //Index evaluation
  ...

  /* Main algorithm */
  Cons1D_to_Prim1D_cu_dev(&Ul_x1Face_dev[ind],&Wl_dev[ind↩
      ],
    &B1_x1Face_dev[ind], Gamma_1);
  Cons1D_to_Prim1D_cu_dev(&Ur_x1Face_dev[ind],&Wr_dev[ind↩
      ],
    &B1_x1Face_dev[ind], Gamma_1);

  flux_roe_cu_dev(Ul_x1Face_dev[ind],Ur_x1Face_dev[ind],↩
      Wl_dev[ind],
    Wr_dev[ind],B1_x1Face_dev[ind],&x1Flux_dev[ind], ↩
        Gamma_1,
    Gamma_2);
}
```

```
__global__ void Cons1D_to_Prim1D_Slice1D_8c(Cons1D *↩
    Ul_x2Face_dev,
    Cons1D *Ur_x2Face_dev, Prim1D *Wl_dev, Prim1D *Wr_dev↩
        ,
    Real *B2_x2Face_dev, Cons1D *x2Flux_dev,
    int is, int ie, int js, int je, int sizex,
    Real Gamma_1, Real Gamma_2) {
  int i = (blockIdx.x * blockDim.x) + threadIdx.x;
  int j;
  calculateIndexes2D(&i, &j, sizex);

  //Index evaluation
  ...

  /* Main algorithm */
  Cons1D_to_Prim1D_cu_dev(&Ul_x2Face_dev[ind],&Wl_dev[ind↩
      ],
    &B2_x2Face_dev[ind], Gamma_1);
  Cons1D_to_Prim1D_cu_dev(&Ur_x2Face_dev[ind],&Wr_dev[ind↩
      ],
    &B2_x2Face_dev[ind], Gamma_1);

  flux_roe_cu_dev(Ul_x2Face_dev[ind],Ur_x2Face_dev[ind],↩
      Wl_dev[ind],
    Wr_dev[ind],B2_x2Face_dev[ind],x2Flux_dev+ind,
    Gamma_1, Gamma_2);
}
```

Listing 8: Th kernel functions for evaluating fluxes in step 8 of the main algorithm

The `flux_roe_cu_dev()` function is based on `flux_roe()` from the Athena-CPU code, but it is defined as the device function, thus can be used within the kernel functions. The same way we implemented the device function, `flux_hlle_cu_dev()`, which is used when Roe's flux evaluation fails.

Step 9 of the main integration function is very similar to step 4. We used the four separate kernel functions as in step 4 to update the interface magnetic fields using CT for a full time-step.

The final two steps update cell centered conserved variables for a full time-step (using `x1Flux_dev` and `x2Flux_dev` fluxes) and cell centered magnetic field (using updated face centered fields). This is performed by the three separate kernel functions: `update_cc_x1_Flux()`, `update_cc_x2_Flux()`, and `update_cc_mf()`. Their implementation is based on the Athena-CPU source code. We converted each loop from original code to the corresponding kernel functions. Each thread within kernel works on a single grid cell and perform single update.

The main integration function, `integrate_2d_cuda()`, which is built by above sequence of the kernel functions executions, can be called from the host code. As a result we get `Grid_gpu` structure updated for a full time-step.

After evaluating conserved values at a new time-step, the Athena-GPU code calculates next time-step from the CFL condition. To avoid unnecessary memory copies between the host and device memories, we implemented time-step calculation on a GPU (Fig. 2). The main scheme from the Athena-CPU code is based on finding a minimum of the time-step, Δt, based on wave speeds in each numerical cell. The `new_dt_cuda()` function uses the two kernel

functions to find the time-step, Δt. The first kernel function, `new_dt_1Step_cuda_kernel()`, evaluates $1/\Delta t_i$ in each grid cell. The second kernel function, `get_max_dti()` searches for a maximum value of $1/\Delta t_i$ (listing 9). It is noteworthy that we need to copy only one single double precision value from the device memory located in the first element of the `max_dti_array` array. Thus time-step evaluation performed on a GPU can be very fast and next Δt is available for the CPU code.

```
__global__ void get_max_dti(int N, int n, Real *↩
     max_dti_array) {
    int i = (blockIdx.x * blockDim.x) + threadIdx.x;
    int i1 = (1 << n) * i;
    int i2 = i1 + ( 1 << (n -1 ));
    if(i2 < N) {
          max_dti_array[i1] = MAX(max_dti_array[i1], ↩
               max_dti_array[i2]);
    }
}

extern "C" void new_dt_cuda(Grid_gpu *pGrid, Real Gamma, ↩
     Real Gamma_1, Real CourNo)
{
        int sizex = pGrid->Nx1+2*nghost;
        int sizey = pGrid->Nx2+2*nghost;
        int nnBlocks = (sizex*sizey)/(BLOCK_SIZE) +
            ((sizex*sizey) % (BLOCK_SIZE) ? 1 : 0);

    // Calculate initial data
    new_dt_1Step_cuda_kernel<<<nnBlocks,BLOCK_SIZE>>>(↩
         pGrid->U, pGrid->B1i, pGrid->B2i,
                pGrid->B3i, pGrid->is, pGrid->ie, pGrid->↩
                     js, pGrid->je, sizex, max_dti_array,
                Gamma, Gamma_1, pGrid->dx1, pGrid->dx2, ↩
                     pGrid->Nx1, pGrid->Nx2);

    int n = 1;
    int N = sizex*sizey / (1 << n) + (sizex*sizey) % (1 ↩
         << n); //How many threads
    nnBlocks = N/BLOCK_SIZE + (N % BLOCK_SIZE ? 1 : 0);

    // Evaluate maximum
    while(N > 1) {
        get_max_dti<<<nnBlocks,BLOCK_SIZE>>>(sizex*sizey,↩
             n, max_dti_array);
        n++;
        N = sizex*sizey / (1 << n) + ((sizex*sizey) % (1 ↩
             << n) ? 1 : 0); //How many threads
        nnBlocks = N/BLOCK_SIZE + (N % BLOCK_SIZE ? 1 : ↩
             0);
    }
    /* Copy max_dti_array[0] as maximum */
    Real max_dti;
    cudaMemcpy(&max_dti, max_dti_array, sizeof(Real), ↩
         cudaMemcpyDeviceToHost);
    // Calculate new dt based on CourNo and max_dti
    ...
}
```

Listing 9: New time-step calculation `new_dt_cuda()` and its kernel function, `get_max_dti()`

The last part of the Athena-CPU code, which we reimplemented for a GPU, concerns boundary conditions. The Athena-GPU implementation of the `set_bvals_cu()` function is very simple. We implemented only periodic boundary conditions as a set of the kernel functions which operate on the `Grid_gpu` structure. The example of one of those kernel functions is presented in listing 10. The main idea is to copy appropriate cell values from the edges to adjacent ghost cells.

```
__global__ void periodic_ix1_cu_step1(Gas *U, Real *B1i, ↩
     int js,
        int je, int is, int ie, int sizex)

{

    /* Calculate and check index */
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    if (j < js || j > je) return;
    for (int i = 1; i <= nghost; i++) {
        U[j*sizex+(is - i)].d = U[j*sizex+(ie - (i - 1))↩
             ].d;
        U[j*sizex+(is - i)].M1 = U[j*sizex+(ie - (i - 1))↩
             ].M1;
        U[j*sizex+(is - i)].M2 = U[j*sizex+(ie - (i - 1))↩
             ].M2;
        U[j*sizex+(is - i)].M3 = U[j*sizex+(ie - (i - 1))↩
             ].M3;
        U[j*sizex+(is - i)].E = U[j*sizex+(ie - (i - 1))↩
             ].E;
        U[j*sizex+(is - i)].B1c = U[j*sizex+(ie - (i - 1)↩
             )].B1c;
        U[j*sizex+(is - i)].B2c = U[j*sizex+(ie - (i - 1)↩
             )].B2c;
        U[j*sizex+(is - i)].B3c = U[j*sizex+(ie - (i - 1)↩
             )].B3c;
        B1i[j*sizex+(is-i)] = B1i[j*sizex+(ie-(i-1))];
    }
}
```

Listing 10: The kernel function for evaluating periodic boundary conditions on the left side of the x-direction

The main loop of the Athena-GPU code, denoted by red dashed line in Fig. 2, is essentially executed on a GPU. Only at given time-steps we need to perform a copy from the `Grid_gpu` to `Grid` structures in order to save intermediate results of simulations for further analysis. Thus a GPU can handle the most expensive computationally calculations while a CPU is responsible only for writing results and scheduling execution of the kernel functions.

The Athena-GPU code is a simple port of the Athena-CPU code. We did not focus on optimization, but rather on a simplicity of implementation. Some of the device and kernel functions are almost exactly the same as in the original code for a CPU. The kernel function is executed as a group of simultaneously running threads, so the loops from a CPU code needed to be converted. In each kernel we evaluate cell index for each thread and we need to specify correct execution configuration for each kernel function call. Some of the kernels use shared memory, but the access to that memory could be further optimized. The Athena-GPU code was used for test cases we describe in details in the following section.

4. Numerical tests and performance analysis of the Athena-GPU code

We use four test platforms to compare performance of the Athena-GPU and Athena-GPU codes. Summary of these test platforms is presented in Table 1. In a parallel execution on GPUs the number of CUDA threads depends upon grid resolution and kernel execution configuration. For many core processors and parallel execution on a CPU we run MPI. For the Athena-GPU code we use CUDA Toolkit 3.1, which includes CUDA C/C++ compiler. We adopt default compiler options with the additional flag, `-arch=sm_13`, which enables double precision support.

Table 1
Test processors and parallel execution configuration

Processor	Model	Cores	Threads	Memory
GPU_1	GTX460 (GF114)	336 (SP) (1.55 GHz)	CUDA threads	1024 GDDR5 (128.0 GB/s)
GPU_2	GTX260 (GT200b)	216 (SP) (1.242 GHz)	CUDA threads	896 GDDR3 (111.9 GB/s)
CPU_1	Core i7 930	4 (2.8 GHz)	4 (MPI processes)	8GB DDR3
CPU_2	C2D E5200	2 (2.5 GHz)	1 (Single process)	4GB DDR2

The first test case is the magnetic field loop advection, originated from the Athena-CPU code [13]. The simulation region is defined as $(-1.0, 1.0) \times (-0.5, 0.5)$ along the $x-$ and $y-$directions. A numerical grid is chosen to satisfy geometry constraint $2N \times N$, where N is the number of cells along the $y-$direction. Numerical runs are performed for N varying from $N = 200$ to $N = 500$. The initial velocity components are $v_x = v_0 \cos(\theta)$, $v_y = v_0 \sin(\theta)$, $v_z = 0$, with the θ angle such as $\cos(\theta) = 2/\sqrt{5}$ and $\sin(\theta) = 1/\sqrt{5}$. The remaining initial parameters are mass density $\rho = 1$, gas pressure $p = 1$ and the base advection speed, $v_0 = \sqrt{5}$. Initial magnetic field components $[B_x,\ B_y] = \nabla \times A\hat{\mathbf{z}}$. Here $\hat{\mathbf{z}}$ is the unit vector along the z-axis and A is the magnetic flux function,

$$A = \begin{cases} A_0(R - r), & r \le R, \\ 0, & r > R, \end{cases} \tag{4}$$

with $r = \sqrt{x^2 + y^2}$. We set $A_0 = 10^{-3}$ and radius $R = 0.3$. For this test case we select periodic boundary conditions. Spatial profiles of the z-component of the current density, $j_z = \frac{1}{\mu}(\nabla \times \mathbf{B})_z$, are shown at $t = 0.8$ in Fig. 3. The magnetic field loop advection test verifies whether the code satisfies $\nabla \cdot \mathbf{B} = 0$ condition, which means that the loop shape should remain unaltered in time. Figure 3 reveals that the profiles of j_z look identically for the Athena-GPU and Athena-CPU codes.

The second test is associated with blast waves and is also taken from the Athena-CPU repositories [13]. The simulation domain spans over the domain $(-0.75, 0.75) \times (-0.5, 0.5)$, with the periodic boundary conditions at its four edges. This domain is covered by $1.5N \times N$ grid points, where N is the grid resolution along the $y-$direction. We perform runs for N varying from $N = 150$ to $N = 500$. Initially, at $t = 0$, we set $\rho = 1$, $\mathbf{v} = 0$, and $B_x = B_y = 10/\sqrt{2}$, $B_z = 0$. In the center of the simulation region we launch a Gaussian pulse in the gas pressure, $p = P \exp[-(x^2 + y^2)/2R^2]$ with radius $R = 0.125$ and $P = 100$. As a result of that the plasma β in the ambient medium is $\beta = 2\mu P/\mathbf{B}^2 = 2$, while at $(x = 0, y = 0)$ $\beta = 200$. The initial pulse triggers blast waves. Figure 4 illustrates mass density profiles at $t = 0.2$ obtained with the Athena-CPU (left) and Athena-GPU (right) codes. As these profiles look identical we infer that the GPU version of the code leads to same results as its CPU counterpart.

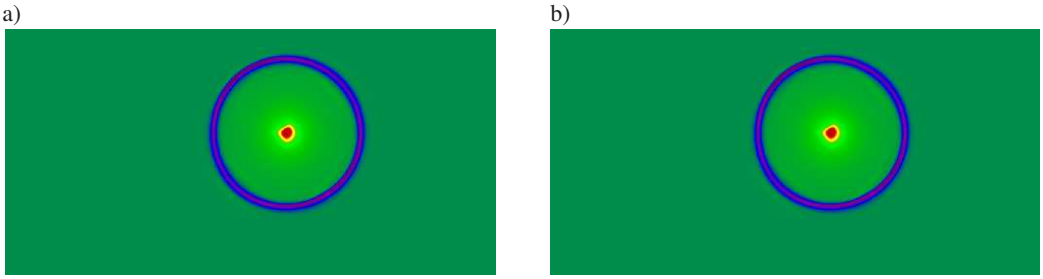

Fig. 3. The z-component of the current density, $j_z = \frac{1}{\mu}(\nabla \times \mathbf{B})_z$, for the field loop problem obtained with the Athena-CPU (left) and Athena-GPU (right) codes for the 500×250 grid points and the linear colour map $(-0.04,\ 0.08)$

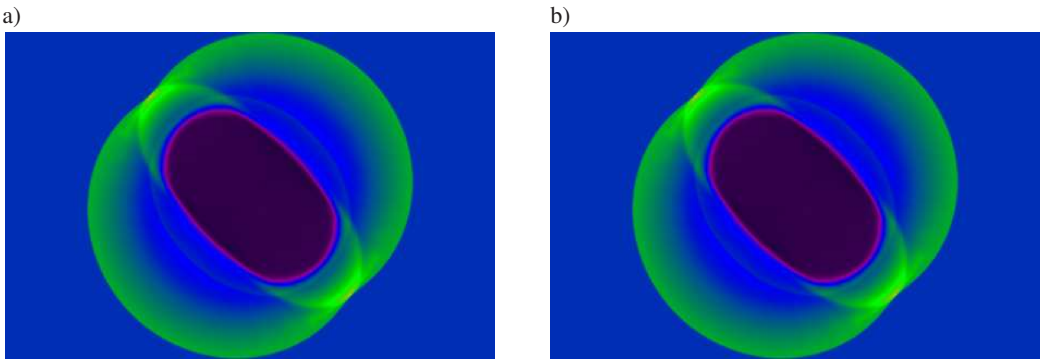

Fig. 4. Mass density profiles for the blast waves problem obtained with the Athena-CPU (left) and Athena-GPU (right) codes for 450×300 grid points and the linear colour map $(0.06,\ 4.4)$

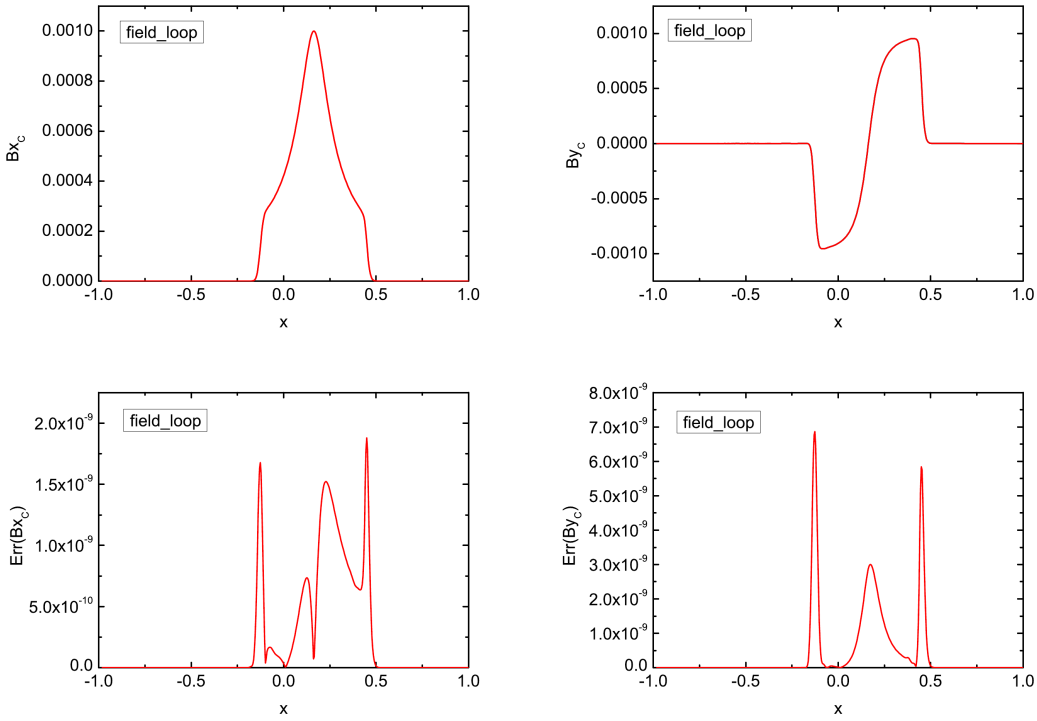

Fig. 5. Magnetic field B_x (B_y) component along $y = 0$ at $t = 0.8$ for the grid resolution 500×250 at top-left (top-right) panel and absolute difference between the CPU and GPU results at bottom-left (bottom-right) panel for the field loop problem

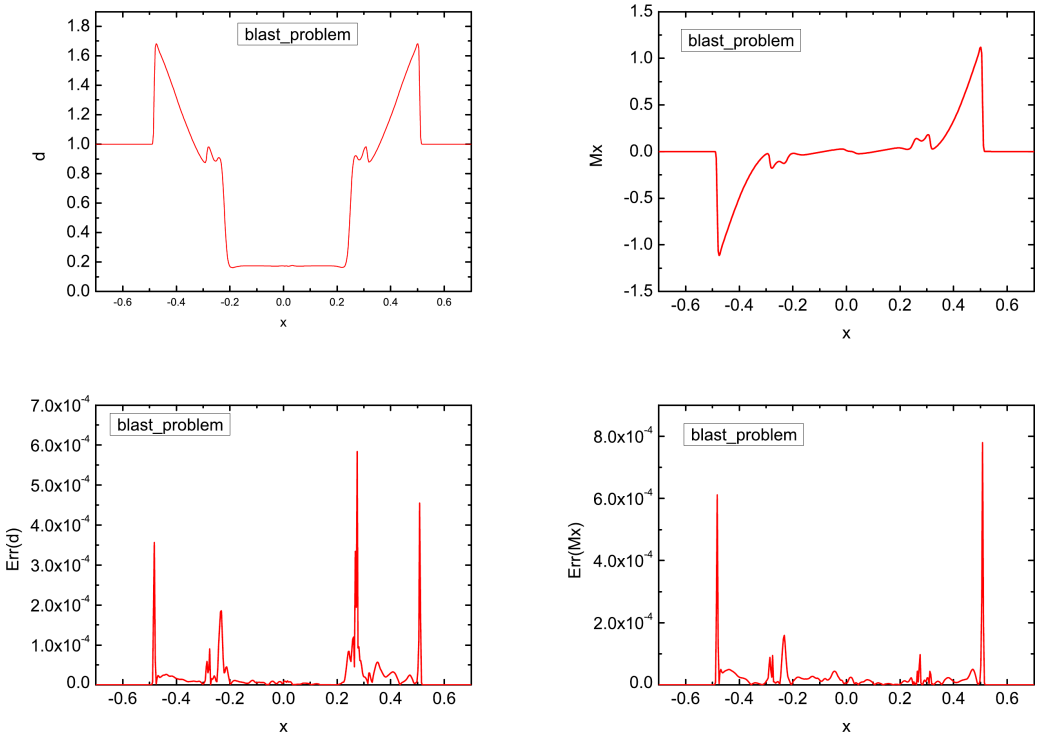

Fig. 6. Mass density (top-left) profiles and momentum density ρv_x (top-right) along $y = 0$ at $t = 0.2$ for grid resolution 450×300 for the blast waves problem. The corresponding absolute difference between the CPU and GPU results are displayed in bottom panels

Figures 5 and 6 compare the results in more details. In top panels slices of given profiles along $y = 0$ are presented. The results from CPU and GPU look identical and they are illustrated by single line. Bottom panels of Figs. 5 and 6 present absolute difference between the results obtained with the Athena-CPU and Athena-GPU codes. In the case of the field loop problem (Fig. 5) there is a difference of an order of 10^{-9} while for the blast wave (Fig. 6) the difference is about

10^{-4}. At steeper profiles the differences become larger, while in places where the profile is smoother, these differences become smaller.

These results, obtained with the Athena-CPU and Athena-GPU codes, look nearly identical as illustrated in Figs. 3 and 4. The more accurate analysis shown in Figs. 5 and 6 reveal small absolute differences between these results. These differences result from the fact that the GPU floating point arithmetic is not fully IEEE754 standard compliant and there are some deviations from this standard. Ordering of floating point math, which is not associative, is also important. The CUDA compiler can make optimisation for multiplication and addition operations and convert them into one multiply-add operation (FMAD) or fused multiply-add (FMA) which also change the precision of calculations. The use of double precision on a GPU make results more precise, but does not resolve the problem. Because of these facts, differences in results are present and they are more prominent for steep profiles. Indeed Figs. 5 and 6 shows that absolute difference between results is larger at steeper profiles.

We made the performance tests based on computer running times (Δt_r) of the Athena-GPU and Athena-CPU codes. Note that the Athena-GPU code includes device memory initialization and memory transfers between the host and device. We run the Athena-CPU code on 4 MPI processes, each working on a part of the whole domain. Running times are summarized in Tables 2 and 3. In both cases the shortest running times are for GPU_1 which is based on Fermi architecture [19]. For the Athena-CPU code Δt_r is shortest for the CPU_1. This is because it runs 4 parallel MPI processes and CPU_2 is theoretically slower in terms of Flops. However, Δt_r for CPU_1 is only about 3 times shorter than for CPU_2. Running times from columns GPU_2 and CPU_2 are nearly equal in case of low numerical grid resolution. This is true up to the 600×300 grid resolution for the magnetic field loop advection and up to 450×300 for the blast waves problem. Comparing in general Δt_r from the Athena-CPU and Athena-GPU codes, we infer that in both test cases GPUs performance is better for a finer grid.

Table 2

Simulation times for the magnetic field loop advection test

Grid	GPU_1 [s]	GPU_2 [s]	CPU_1 [s]	CPU_2 [s]
400x200	228.46	506,39	521,65	1693,90
500x250	374.23	823,79	1063,45	3303,53
600x300	589.73	1407,85	1789,13	5752,37
700x350	847.92	1773,98	2880,91	9067,04
800x400	1174.32	2441,64	4340,89	13576,60
900x450	1556.99	3228,03	6211,21	19450,82
1000x500	2012.64	4577,01	8601,99	27203,85

Table 3

Simulation times for the blast problem

Grid	GPU_1 [s]	GPU_2 [s]	CPU_1 [s]	CPU_2 [s]
225x150	48.25	111,38	89,59	337,06
300x200	95.81	208,72	223,27	797,09
375x250	155.84	342,5	440,08	1553,56
450x300	246.04	519,66	764,22	2701,72
525x350	352.09	745,1	1202,47	4300,76
600x400	486.14	1128,37	1800,08	6463,29
750x500	838.45	1751,91	3540,67	12703,60

In Fig. 7 we compare Δt_r. In both cases, the blast and field loop advection problems, Δt_r grows with the size of a numerical grid. In the case of a low resolution grid, Δt_r does not vary much with the test platforms. However, differences in Δt_r become more significant for a finer grid and Δt_r for the Athena-GPU code does not grow that much as for the Athena-CPU code.

We measure the ratio of code running times,

$$s = \Delta t_r(CPU_x)/\Delta t_r(GPU_y), \tag{5}$$

as a relative speed-up of the Athena-GPU code, where x and y denote a type of chip specified in Table 1. In both cases s grows with a grid resolution (Fig. 8). This growth is almost linear. Comparing the serial Athena code with the Athena-GPU code we infer that for the finest grid, the latter is almost 14 times faster in the loop problem and over 15 times faster in the blast problem. For the finest grid four MPI processes are about 4 times (for the blast problem) and 3 times (for the loop problem) slower compared to the Athena-GPU running on GTX460.

As it is shown in Figs. 7 and 8, Δt_r grows with a grid resolution. Obviously the Athena-GPU code performs better for a finer grid. The Athena-CPU works better with the use of MPI but its performance decreases faster than the Athena-GPU for a finer grid. This is because the number of MPI processes is relatively small, thus the simulation domain is divided into large sub domains. As a result, for each MPI process we are more likely to saturate maximum performance for each CPU core. The Athena-GPU makes use of a highly parallel architecture of a GPU which allows to run a large number of independent threads. This leads to a growth of performance with the size of the problem because when more grid points are needed to be updated more threads within the kernel function are used. A more powerful GPU is, more threads can be run simultaneously and the kernel function is executed more efficiently. We also infer that for problems with a low grid resolution, Δt_r remains essentially same for the Athena-GPU and Athena-CPU codes. In some cases Δt_r for the Athena-GPU code is larger than for the Athena-CPU code. This confirms that a GPU performs better for more complex problems with a larger simulation box and finer grid.

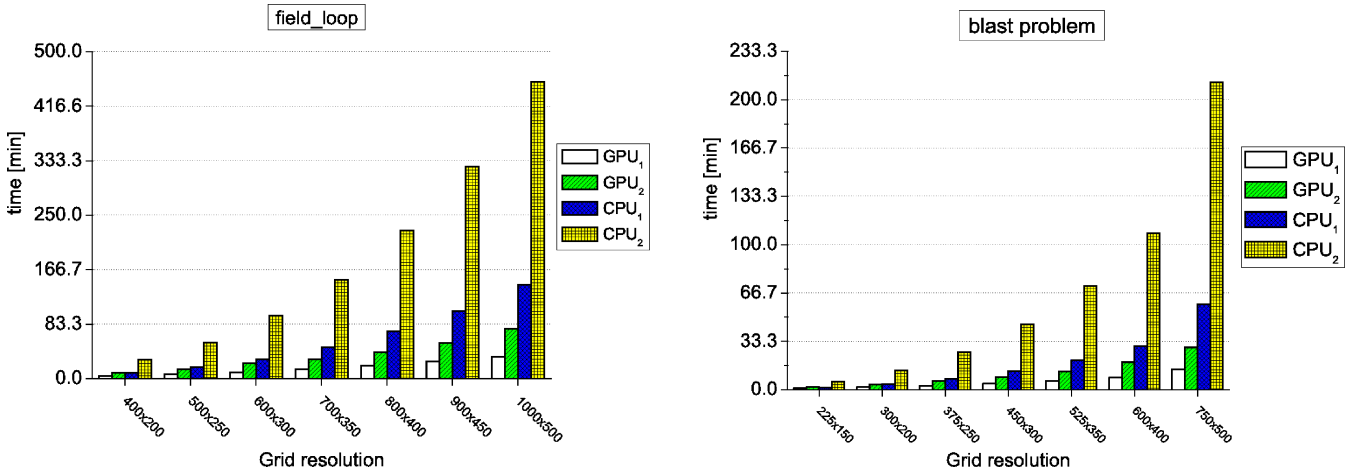

Fig. 7. Computer running time for the field loop (left panel) and blast wave (right panel) problems vs. grid resolution

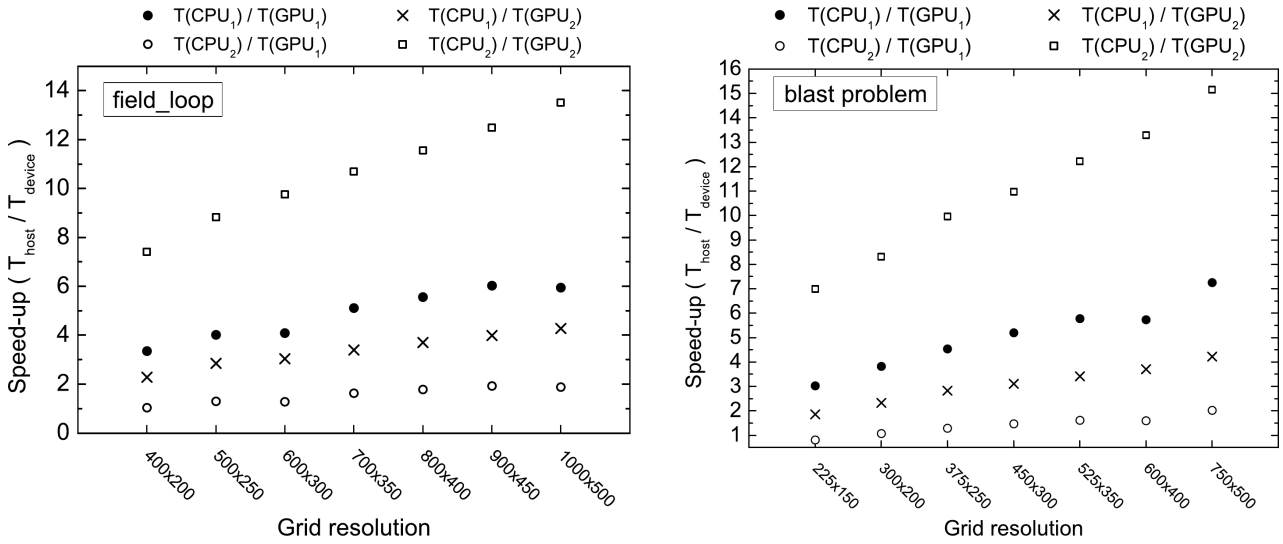

Fig. 8. Relative speed-up of the Athena-GPU code for the field loop (left panel) and blast (right panel) problems vs. grid resolution

5. Summary

In this paper we have presented a new (albeit unofficial) implementation of the Athena-GPU code which can be run on NVIDIA CUDA GPUs. A performance analysis of this code shows that GPUs are capable to run complex physical problems such as magnetohydrodynamics [15]. We also made results correctness comparison between the Athena-CPU and Athena-GPU codes and showed that it is acceptable for the double precision. In order to show that performance could be much better on a GPU than on its CPU counterpart we measured computation running time. For a double precision calculations made by a GPU, running times become shortest for Fermi architecture. By taking into account a scalable architecture of CUDA GPUs the Athena-GPU could perform even better on the newest versions of NIVIDA's graphic chips. Performing physical simulations with the use of a GPU allows us to run more complex problems in short time, which is cheaper than using expensive CPU platforms.

The Athena-GPU code may be extended to three-dimensional problems and optimized for the Fermi architecture. This is a formidable task as a significant amount of memory would be required for running simulations. There is also possibility to extend the Athena-GPU code to use multiple graphic cards simultaneously similarly to MPI multiple processes, increasing by this way performance of the code. See, e.g. [20].

Acknowledgements. The authors express their cordial thanks to Prof. James Stone for his comment on the earlier version of this draft.

REFERENCES

[1] K. Murawski and T. Tanaka, "Godunov-type methods for two-component magnetohydrodynamic equations", *Bull. Pol. Ac.: Tech.* 60 (2), 343–348 (2012).

[2] `http://www.astro.princeton.edu/~jstone/zeus.html` (2011).

[3] `http://flash.uchicago.edu/website/home` (2011).

[4] http://plutocode.ph.unito.it/ (2011).
[5] http://nirvana-code.aip.de/ (2011).
[6] http://folk.uio.no/mcmurry/amhd/ (2011).
[7] https://trac.princeton.edu/Athena (2011).
[8] http://www.mcs.anl.gov/research/projects/mpi (2011).
[9] NVIDIA, *NVIDIA CUDA Programming Guide 3.1*, NVIDIA (2010).
[10] H. Schive, Y. Tsai, and T. Chiueh, "GAMER: a graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics", *Astrophys. J. Suppl.* 186, 457–484 (2010).
[11] B. Pang, U. Pen, and M. Perrone, "Magnetohydrodynamics on Heterogeneous architectures: a performance comparison", CoRR, abs/1004.1680 (2010).
[12] H.-C. Wong, U.-H. Wong, X. Feng, and Z. Tang, "Efficient magnetohydrodynamic simulations on graphics processing units with CUDA", *eprint arXiv:0908.4362* (2009).
[13] J.M. Stone, T.A. Gardiner, P. Teuben, J.F. Hawley, and J.B. Simon, "Athena: a new code for astrophysical MHD", *Astrophys. J. Suppl.* 178 (1), 137–177 (2008).
[14] T.A. Gardiner and J.M. Stone, "An unsplit Godunov method for ideal MHD via constrained transport", *J. Comp. Phys.* 205 (2), 509–539 (2005).
[15] K. Murawski, "Numerical solutions of magnetohydrodynamic equations", *Bull. Pol. Ac.: Tech.* 59 (1), 1–8 (2011).
[16] P. Colella and P.R. Woodward, "The Piecewise Parabolic Method (PPM) for gas-dynamical simulations", *J. Comp. Phys.* 54 (1), 174–201 (1984).
[17] P.L. Roe, "Approximate Riemann solvers, parameter vectors, and difference schemes", *J. Comp. Phys.* 43 (2), 357–372 (1981).
[18] B. Einfeldt, C.D. Munz, P.L. Roe, and B. Sjogreen, "On Godunov-type methods near low densities", *J. Comput. Phys.* 92 (2), 273–295 (1991).
[19] NVIDIA, *Whitepaper. NVIDIA Next Generation CUDA Compute Architecture: Fermi*, NVIDIA (2009).
[20] K. Murawski, K. Murawski Jr., and H.-Y. Schive, "Numerical simulations of acoustic waves with the graphic acceleration GAMER code", *Bull. Pol. Ac.: Tech.* 60 (4), 787–792 (2012).