Control and Cybernetics

vol. **43** (2014) No. 2

MPI-GPU/CUDA implementation of TVDLF method for the two-dimensional advection equation*

by

Kamil Murawski[1], **Krzysztof Murawski**[2], **and Przemysław Stpiczyński**[3]

[1] Institute of Informatics, UMCS, Pl. M. Curie-Sklodowskiej 1, 20-031 Lublin, Poland

[2] Faculty of Mathematics, Physics and Informatics, UMCS, ul. Radziszewskiego 10, 20-031 Lublin, Poland

[3] Institute of Mathematics, UMCS, Pl. M. Curie-Sklodowskiej 1, 20-031 Lublin, Poland

Abstract: We implement Total Variation Diminishing Lax Friedrichs (TVDLF, or Rusanov) method to obtain numerical solutions of the two-dimensional advection equation. Despite the simplicity of this equation, solving it numerically is a formidable task. Based on the use of the original C++ MPI-GPU/CUDA code we present results of numerical tests we performed. These tests show that our code represents well the square wave profiles, leading to up to 60-times faster calculations with the use of MPI than with its serial counter-part.

Keywords: computer science, modelling, numerical methods, hyperbolic equations

1. Introduction

Despite the significant development of mathematical and numerical tools, solving of some scientific problems still constitutes a formidable task. As a result, in many cases the exact analytical solution is difficult or even impossible to obtain (see, e.g., Toro, 2008, and references therein). In such cases, until a proper analytical method is discovered, finding an approximate solution has to suffice. However, as involving tedious and time-consuming calculations, the approximation can also not be easy to obtain (see Murawski and Lee, 2012, or Murawski et al., 2012, 2013). This is an area in which graphical cards can be adopted for improving the performance of the calculations (see Schieve et al., 2010).

Graphical processing units (GPUs, see Nickolls et al., 2010) have recently been widely used for scientific computing due to their large number of parallel processors, which can be exploited using the Compute Unified Device Architecture (CUDA) programming language (see Nickolls and Dally, 2008) that

*Submitted: August 2013; Accepted: April 2014 facilitates multi-threaded programming by introducing the concept of a kernel. GPUs offer very high performance at low costs for data-parallel computational tasks, when computations are carried out in single precision (see Leist et al., 2009). Thus, it is a good idea to develop algorithms for hybrid (heterogeneous) computer architectures where large parallelizable tasks are scheduled for execution on GPUs, while small non-parallelizable tasks should be run on CPUs.

The application of GPU/CUDA for solving hyperbolic equations is sparse. See, however, Schieve et al. (2010) and Wasiljew and Murawski (2013) who adopted GPU/CUDA for solving multi-dimensional set of hyperbolic (Euler and magnetohydrodynamic) equations. Yet, the numerical methods they implemented are so sophisticated that their codes are difficult to generalize for an arbitrary set of hyperbolic equations. Therefore, our strategy is to devise a MPI-GPU/CUDA numerical code for the simplest conceivable two-dimensional (2D) hyperbolic equation, which is the advection equation. We use Total Variation Diminishing Lax Friedrichs (TVDLF, or Rusanov) method, which can be easily adopted to any set of hyperbolic equations. As a final product of our effort we write the code in C, which can serve as a template for the development of codes for more complex equations.

This paper is organized as follows. Section 2 presents the 2D advection equation. Finite-difference numerical schemes with particular emphasis on TVDLF method are illustrated in Section 3. The original studies on implementation of the TVDLF into the MPI-GPU/CUDA numerical code for the two-dimensional advection equation and the results of numerical experiments are reported in Section 4. This paper closes with discussions and conclusions in Section 5.

2. The advection equation

Among various types of partial differential equations, the family of hyperbolic equations gained much interest, as these equations describe ubiquitous wave phenomena. A one-dimensional (1D) hyperbolic equation can be written in the general form (see LeVeque, 2002)

$$q_{,\mathrm{t}} + f(q)_{,\mathrm{x}} = 0\,, \tag{1}$$

where $q = q(x,t)$ is a real function of coordinate x and time t, $f(q)$ is a flux and the following notation is adopted:

$$q_{,\mathrm{t}} \equiv \frac{\partial q}{\partial t}\,, \quad f(q)_{,\mathrm{x}} \equiv \frac{\partial f}{\partial x}\,. \tag{2}$$

As a particular application of the flux, $f(q)$, in Eq. (1) we can choose

$$f(q) = \lambda_{\mathrm{x}} q\,, \tag{3}$$

where $\lambda_{\mathrm{x}} = const$ is the advection speed along the x-direction. This choice results in the 1D advection equation, which can be written as

$$q_{,\mathrm{t}} + \lambda_{\mathrm{x}} q_{,\mathrm{x}} = 0\,. \tag{4}$$

The 1D advection equation is the simplest conceivable hyperbolic equation (see Toro, 2008).

The 2D generalization of the 1D advection equation (1) is (see LeVeque, 2002)

$$q_{,\mathrm{t}} + \lambda_{\mathrm{x}} q_{,\mathrm{x}} + \lambda_{\mathrm{y}} q_{,\mathrm{y}} = 0\,, \tag{5}$$

where $\lambda_{\mathrm{y}} = const$ is the advection speed along the y-direction. The initial state for this equation is specified as

$$q(x, y, t = 0) = q^0(x, y)\,, \tag{6}$$

where $q^0(x, y)$ denotes the initial condition. We can verify by inspection that the initial-value (Cauchy's) problem is solved as

$$q(x, y, t) = q^0(x - \lambda_{\mathrm{x}} t, y - \lambda_{\mathrm{y}} t)\,. \tag{7}$$

As a result, we infer that the initial condition $q^0(x, y)$ is translated along the x-axis (y-axis) by $\lambda_{\mathrm{x}} t$ ($\lambda_{\mathrm{y}} t$). As the initial-value problem is trivially solved, this equation is ideal for testing numerical methods, this being exactly done in the resent paper.

3. TVDLF method for the advection equation

The goal in this part of the paper is to present the TVDLF method of solving the 1D advection equation (4). Extension of this method to higher dimensions is straightforward. We introduce the numerical grid of the size Δx and the time-step Δt as

$$x_i = i\Delta x\,, \qquad i = 0, 1, \ldots, i_{\max}\,, \qquad \Delta x = x_{i+1} - x_i\,, \tag{8}$$

$$t^n = n\Delta t\,, \qquad n = 0, 1, \ldots, n_{\max}\,, \qquad \Delta t = t^{n+1} - t^n\,. \tag{9}$$

Here, x_i denotes the spatial coordinate that is evaluated at the i-th cell center and t^n is a moment of time, specified by integer n. The symbol Δx denotes the size of a numerical cell. For simplicity reasons we assume that the numerical grid is uniform, which corresponds to a constant grid size, $\Delta x = const$. Having defined x_i and t^n we introduce the following notation:

$$q_i^n \equiv q(x_i, t^n)\,. \tag{10}$$

We approximate now the partial derivatives by finite differences; for the temporal (spatial) derivative we adopt the *forward (centered) Euler scheme* (see Morton, 2005, or Murawski, 2002)

$$q_{,t} \simeq \frac{q_i^{n+1} - q_i^n}{\Delta t}\,, \quad q_{,x} \simeq \frac{q_{i+1}^n - q_{i-1}^n}{2\Delta x}\,. \tag{11}$$

From the 1D advection equation (4) we get the Forward in Time and Centered in Space (FTCS) numerical scheme (see Murawski, 2002)

$$q_i^{n+1} = q_i^n + \frac{c}{2}(q_{i-1}^n - q_{i+1}^n)\,, \tag{12}$$

where we implemented the Courant-Friedrichs-Lewy (CFL or Courant) number c as

$$c = \frac{\lambda_x \Delta t}{\Delta x} = \frac{\lambda_x}{\Delta x / \Delta t} = \frac{\text{advection speed}}{\text{grid speed}} \,. \tag{13}$$

From Eq. (12) we compute explicitly the evolution of q in time at every i point, except for $i = 0$ and $i = i_{\max}$. For these points we need to specify the boundary conditions. As an example we set open (or transmissive) boundary conditions

$$q_0^n = q_1^n \,, \quad q_{i_{max}+1}^n = q_{i_{\max}}^n \,. \tag{14}$$

The main drawback of the FTCS scheme is that it is unconditionally unstable and therefore it cannot be used in practice (see Murawski, 2002).

3.1. Lax-Friedrichs numerical scheme

The FTCS scheme of Eq. (12) can be stabilized by replacing q_i^n, standing on the right-hand side, by the arithmetical average of solutions at the neighboring points. As a result, we arrive at the *Lax-Friedrichs (LF) scheme*,

$$q_i^{n+1} = \frac{q_{i+1}^n + q_{i-1}^n}{2} + \frac{c}{2}(q_{i-1}^n - q_{i+1}^n) \,. \tag{15}$$

We can rewrite this scheme as

$$q_i^{n+1} = \frac{1+c}{2} q_{i-1}^n + \frac{1-c}{2} q_{i+1}^n \,. \tag{16}$$

The LF scheme is conditionally stable for $|c| \leq 1$ (see Toro, 2008), and it is first-order accurate in space and time, which means that numerical errors are proportional to Δx and Δt (see Toro, 2008). In order to obtain a spatially high-order differencing, one can implement a piece-wise linear representation of a function $q(x)$ and add slope limiters (see LeVeque, 2002) to filter out the numerically induced oscillations. The LF method of Eq. (15) is known for its simplicity as no Riemann problem (see Toro, 2008) needs to be solved. We replace q_{i+1} and q_i with the second-order accurate boundary values $q^R_{i+\frac{1}{2}}$ and $q^L_{i+\frac{1}{2}}$ (Fig. 1), respectively, as

$$\begin{aligned} q^R_{i+\frac{1}{2}} &= q_{i+1} - \frac{1}{2}\Delta\bar{q}_{i+1} \,, \\ q^L_{i+\frac{1}{2}} &= q_i + \frac{1}{2}\Delta\bar{q}_i \,, \end{aligned} \tag{17}$$

where $\Delta\bar{q}_i$ is a limited version of Δq_i and the index $i + \frac{1}{2}$ corresponds to the right interface. A good choice is the minmod slope limiter which is given by (see LeVeque, 2002)

$$\begin{gathered} \Delta\bar{q}_i^n = minmod(\Delta q_i^n) = sgn(q_i^n - q_{i-1}^n) \\ \max[0, \min\{|q_i^n - q_{i-1}^n|, (q_{i+1}^n - q_i^n) sgn(q_i^n - q_{i-1}^n)\}]. \end{gathered} \tag{18}$$

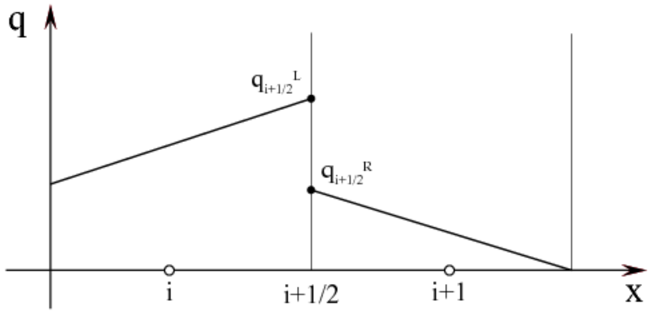


Figure 1. Piece-wise linear representation of $q(x)$ and boundary extrapolated values $q^R_{i+\frac{1}{2}}$ and $q^L_{i+\frac{1}{2}}$

As a result, the first-order accurate LF scheme becomes TVDLF (or Rusanov), which is second-order accurate in space (see Tóth and Odstrcil, 1996),

$$q_i^{n+1} = q_i^n - \frac{\Delta t}{\Delta x}(F^n_{i+\frac{1}{2}} - F^n_{i-\frac{1}{2}}) , \tag{19}$$

where the numerical flux is specified as

$$F^n_{i+\frac{1}{2}}(q^n_{i+\frac{1}{2}}) = \frac{1}{2}\left[f(q^{\rm R}_{i+\frac{1}{2}}) + f(q^L_{i+\frac{1}{2}}) - \left| c^{\max}\left(\frac{q^{\rm R}_{i+\frac{1}{2}} + q^{\rm L}_{i+\frac{1}{2}}}{2} \right) \right| \Delta q^{\rm RL}_{i+\frac{1}{2}} \right] . \tag{20}$$

Here

$$\Delta q^{\rm RL}_{i+\frac{1}{2}} = q^{\rm R}_{i+\frac{1}{2}} - q^{\rm L}_{i+\frac{1}{2}} \tag{21}$$

and the symbol $c^{\max}(q)$ denotes the maximum value of the characteristic speed in the system, i.e.

$$c^{\max}(q) = \max\left(\frac{\partial f}{\partial q}\right) . \tag{22}$$

Note that $f(q)$ is a physical flux, which for the 1D advection equation is $f(q) = \lambda_x q$. Hence, $\partial f/\partial q = \lambda_x$ and $c^{\max}(q) = \lambda_x$.

For practical reasons this numerical method should be enhanced to second order in time by implementing the Predictor-Corrector method (see LeVeque, 2002),

$$q_i^{n+\frac{1}{2}} = q_i^n - \frac{\Delta t}{2\Delta x}[f(q^n_{i+\frac{1}{2}}) - f(q^n_{i-\frac{1}{2}})] , \tag{23}$$

$$q_i^{n+1} = q_i^n - \frac{\Delta t}{\Delta x}[F(q^{n+\frac{1}{2}}_{i+\frac{1}{2}}) - F(q^{n+\frac{1}{2}}_{i-\frac{1}{2}})] , \tag{24}$$

where $F(q)$ is specified by Eq. (3.1).

4. MPI-GPU/CUDA code for the 2D advection equation

The code *scalar_eq_2d_cuda_mpi.cu* that we developed comprises a collection of routines, which run on the host or on the GPU. The following CUDA routines are executed on the device:

- *flux_x/flux_y* - returns the flux value along x-/y-direction for the given velocity and $q(x, y, t)$ value,
- *fluxTVDLF_x/fluxTVDLF_y* - returns the TVDLF flux value along x-/y-direction,
- *setBoundaryConditionsOpen* - sets open boundary conditions,
- *setBoundaryConditionsPeriodic* - sets periodic boundary conditions,
- *evolveQ_predictionStep* - first step (prediction) of evolving the solution,
- *evolveQ_correctionStep* - final step (correction) of evolving the solution,
- *cudaSafe, cudaCheckError* - error debugging functions.

The following functions are executed on the host:

- *readSettings* - reads the parameters of the problem from the file *settings.ini*,
- *outputData* - saves results to file *n_data.xxx*, where n is the rank id and *xxx* is the consecutive data number,
- *outputSettingsForGDL* - saves to file *config.ini* settings used in GDL for visualizing the data,
- *setVariables* - declares and sets variables and arrays,
- *setGridXY* - sets the 2D grid,
- *setInitialConditions* - specifies the initial conditions,
- *loopIteration* - a single iteration in time of the main loop of the program,
- *main* - calls the required routines and manages the communication between MPI ranks.

The flow chart of the code is illustrated in Fig. 2. After the program has been called, the initial settings are read from a file. These include the CFL value, advection speeds λ_x and λ_y, limits of a numerical box, initial signal size and position, the number of grid cells along x- and y-directions, maximum time of the simulation, type of boundary conditions, number of data dumps, and CUDA block size (number of threads in a block). In the following step, the variables (i.e. Δx, Δy, Δt, etc.), arrays (2D q, 1D x and y) and initial conditions are set. The following part of the code is the main loop, which is run until current time ($tCurr$) is smaller or equal than maximum simulation time ($tMax$). In the loop, the main simulation array q is sent to the device, and then a single kernel iteration is executed. Next, the array is copied back to the host and common parts are being exchanged between MPI nodes, transfer of which is described below. Then, current simulation time is being updated and in the case it is larger than the data output time ($tOut$), data is being dumped into a file, which completes the main simulation loop.

MPI is used as follows. Each node keeps a part of the array q. For instance, in case of four available nodes, each one holds a quarter of the array. As for the calculation of q_{ij}^{n+1} in a single cell, eight neighboring values are required

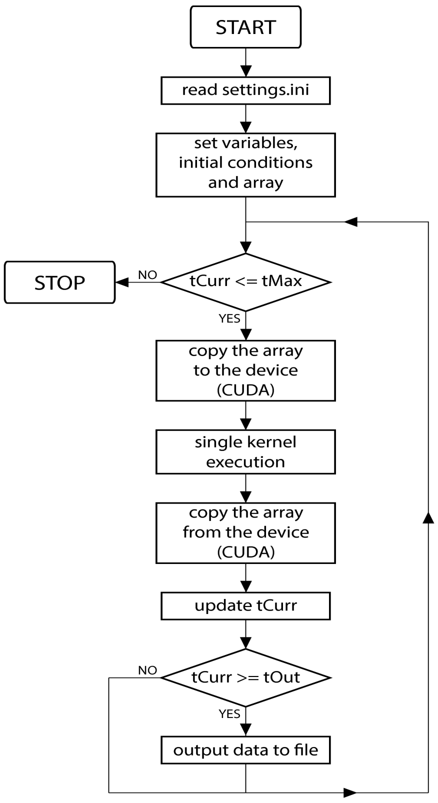


Figure 2. Flow chart of the scalar_eq_2d_cuda_mpi.cu code

(Fig. 3), there are common parts of the array which have to be exchanged between particular nodes after each iteration. As shown in Fig. 4, node (0, 0) sends its lowest common part to node (0, 1), while (0, 1) sends its uppermost common part to node (0, 0). The same happens to the most right common part of node (0, 0), which is sent to node (1, 0). Node (1, 0) also sends its most left common part to node (0, 0). In a case of more available nodes, the number of send/receive operations can grow up to four for each node. The number of these operations for each possible node configuration is presented in Fig. 5.

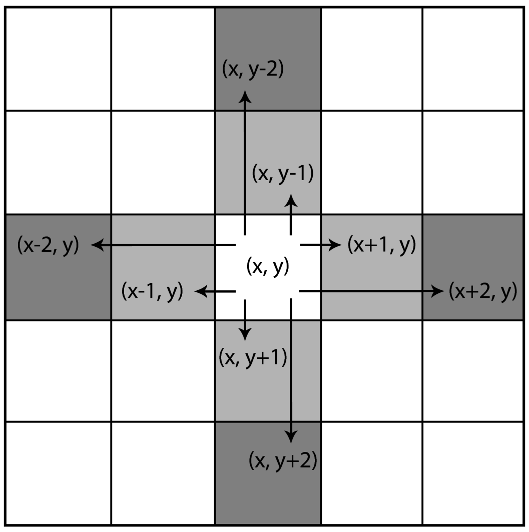


Figure 3. Cells used to evaluate a single cell value $q_{\mathrm{ij}}^{\mathrm{n}+1}$

4.1. Numerical results

The advection equation (5) is solved numerically using the original code. As the initial condition we set at $t = 0$ seconds the square wave profile (Fig. 6, top-left panel). Sample results with the use of TVDLF scheme and 1024x1024 grid points are displayed in Fig. 6. It follows from Eq. (7) that the initial condition is supposed to move towards the upper-right corner, without changing its form, with advection velocities λ_x and λ_y, which in the current numerical experiment are both set to 1. However, as a result of numerical diffusion, which is an inherent feature built in every numerical code, the initial profile spreads in space, and this spreading grows in time. Similarly, the dispersive errors, which reveal themselves by numerically induced unphysical oscillations, are present too. As the presented spatial profiles of $q(x,t)$ do not show much of these inherent features we infer from the numerical data that the TVDLF method represents well the square wave profile.

4.2. Performance tests

The code has been tested on the SOLARIS cluster, which is located in the building of the Faculty of Mathematics, Physics and Computer Sciences of the Maria Curie Skłodowska University in Lublin. This cluster contains 40 nodes,

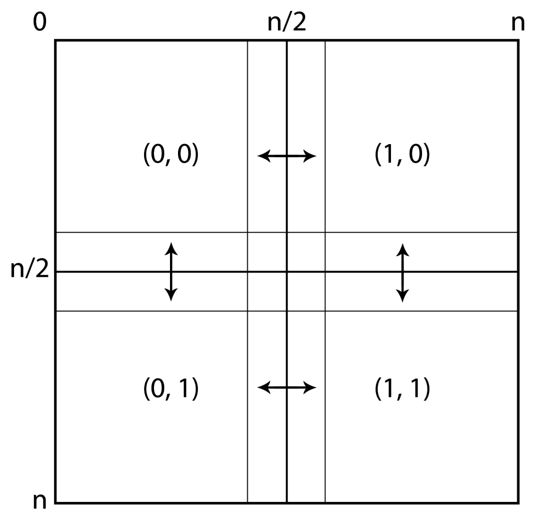


Figure 4. Parts of arrays being exchanged between four nodes, using MPI

each with two Intel Xeon X5650 (6 cores each, 2.67 GHz, 48GB RAM) and two NVIDIA Tesla M2050 (448 cores, 3GB GDDR5 RAM with ECC off), connected using 40 Gbit/s Infiniband, running under Linux with NVIDIA CUDA Toolkit ver. 5.0 and Intel Cluster Studio ver. 2012. The MPI program has been tested for 4, 16, and 64 MPI processes with two processes per node, thus each process has been responsible for managing one GPU card. The tests have been run for the following grid sizes: 512×512, 1024×1024, 2048×2048, 4096×4096, 8192×8192, and 16384×16384.

Table 1 shows the execution times of the MPI program, for $\#GPUs = 4$, 16, 64, and for the single-GPU version for particular grid sizes and various numbers of MPI processes, where all results are averaged values of ten separate runs. Figure 7 presents the speedup of the MPI program over its single-GPU version for particular from two points of view. The first one (left) shows how the speedup scales when the number of cells grows. The next one (right) shows how the speedup changes when the number of processes (and GPU cards) grows.

Note that for smaller grid sizes, namely 512×512 and 1024×1024, the MPI version of the code is slower than the non-MPI version. The reason for this lack of performance boost is that the execution of MPI transfer functions takes time, which in case of smaller arrays has a big impact on the overall computing time. However, in the case of the grid 2048×2048, the MPI version already

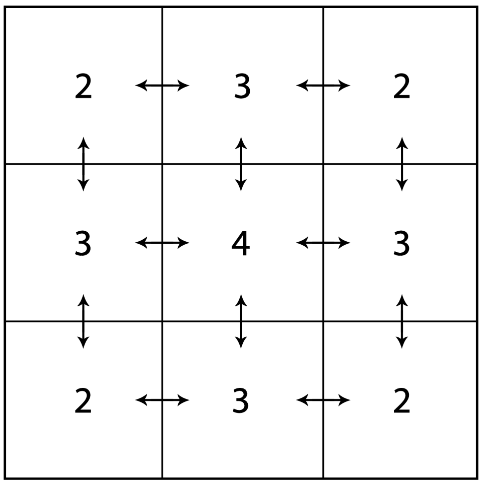


Figure 5. The numbers of MPI send/receive operations for each possible node configuration

becomes faster than the non-MPI version. The bigger the grid size, the higher the performance gain measured by the speedup factor. However, the use of 64 processes is reasonable only for grid sizes bigger than 4096×4096 and then the performance of the MPI version of the code grows significantly. For the grid size of 8192×8192 the speedup is about 12, whereas for 16384×16384 the performance gain grows up to 60 when 64 processes are used.

5. Summary and conclusions

In this paper we presented the TVDLF numerical scheme for the 2D advection equation, its implementation into the original MPI-GPU/CUDA code, and the results of numerical tests that we performed. These tests reveal that the code represents well the steep spatial profiles and it performs up to 60-times faster for large numerical grids than its non-MPI counterpart.

The original MPI-GPU/CUDA code that was introduced in Section 4 can be generalized along various ways. A first potential generalization of this code would be to adapt it for a nonlinear hyperbolic equation such as inviscid Burgers equation (see Toro, 2008),

$$q_{,t} + qq_{,x} + qq_{,y} = 0\,. \tag{25}$$

The code is ready for this adaptation, which would require minor changes of the routines *evolveQ_predictionStep* and *evolveQ _correctionStep*. These routines

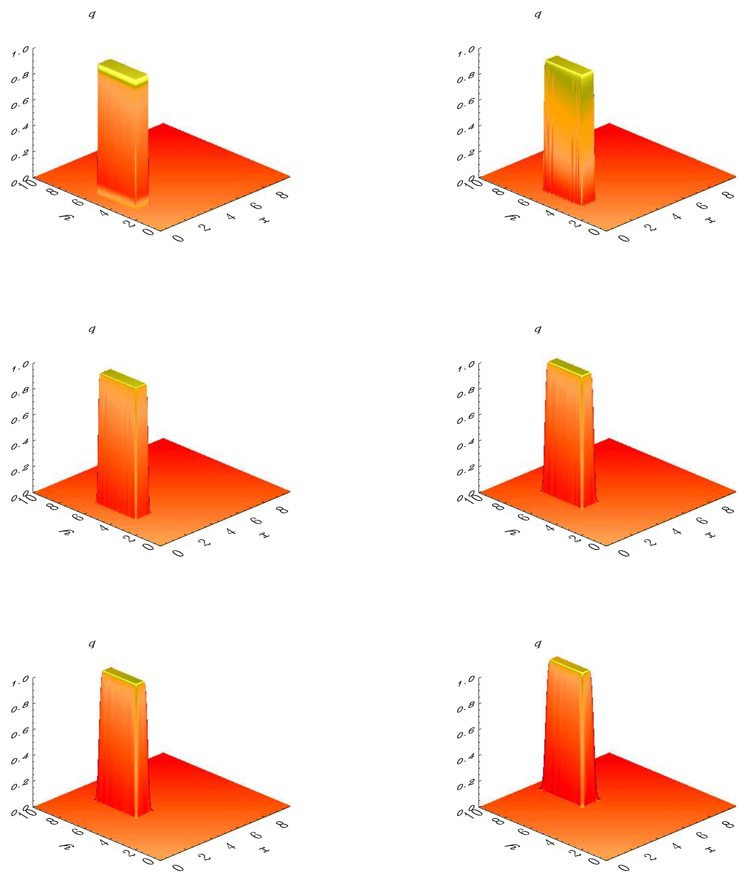
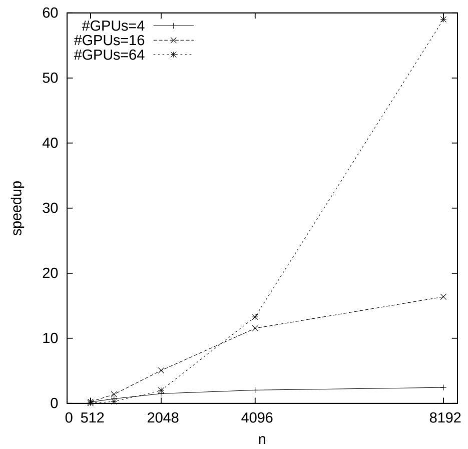


Figure 6. Numerical solutions of the 2D advection equation for TVDLF method and 1024×1024 grid cells at times $t = 0$ s (top-left panel), $t = 0.8$ s, $t = 1.6$ s, $t = 2.4$ s, $t = 3.2$ s and $t = 4$ s (bottom-right panel)

Table 1. Execution time of the MPI program and its single-GPU version for particular grid sizes and various numbers of processes

	#GPUs=			
n	1	4	16	64
512	0.30	1.49	1.30	8.13
1024	2.17	3.06	1.56	8.12
2048	16.43	11.01	3.26	8.26
4096	133.64	65.77	11.59	10.06
8192	1145.91	469.33	70.02	19.42
16384	–	3727.84	500.57	78.73



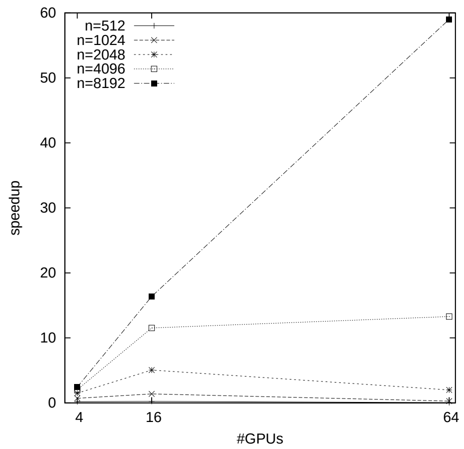


Figure 7. Speedup of the MPI program over its single-GPU version for particular grid sizes (left) and various numbers of processes and GPUs (right)

evaluate fluxes along the x- and y-directions in the prediction and correction steps, respectively.

The MPI-GPU/CUDA code can be also extended for a set of hyperbolic equations (see LeVeque, 2002)

$$\mathbf{q}_{,t} + \mathbf{f}(\mathbf{q})_{,\mathbf{x}} + \mathbf{g}(\mathbf{q})_{,\mathbf{y}} = \mathbf{0}\,, \tag{26}$$

where $\mathbf{q} = (q_1, q_2, ..., q_m)^{\mathrm{T}}$ is a state vector and $\mathbf{f}$ and $\mathbf{q}$ are vector fluxes along x- and y-direction, respectively. Here $()^{\mathrm{T}}$ stands for transposition. This task does not seem to be very difficult, as it requires recoding for the vector $\mathbf{q}(x, y, t)$ instead for the scalar $q(x, y, t)$.

Other important generalization of the code would be the development of its version for the 3D hyperbolic equation,

$$q_{,t} + f(q)_{,x} + g(q)_{,y} + h(q)_z = 0\,, \tag{27}$$

where $f(q)$, $g(q)$ and $h(q)$ denote scalar fluxes along x-, y- and z-directions, respectively. This is a formidable task as this generalization would require quite a large amount of effort. In particular, CUDA grid and block configuration would have to be reconstructed in order for the code to perform efficiently.

Although the TVDLF method can correctly describe discontinuous spatial profiles, the advantages offered by its employment are evident in presence of smooth flow. As a result, more advanced and accurate, numerical methods for solving nonlinear hyperbolic equations, which naturally possess shock solutions, can be implemented. Among these methods are robust higher-order Godunov-type numerical schemes (see Toro, 2008; LeVeque, 2002; or Murawski, 2012). Finally, the code can be optimized in order to gain higher computing performance. One way of optimizing it would be to change the array transfer between the host and the device. In the current version during one simulation iteration the whole simulation array is copied to the device. Next, the kernel is executed and the array is copied back to the host. The major part of time is consumed on CUDA transfers, which are almost 200 times slower than MPI transfers of the common parts of the array. This can be fixed by transferring only the common parts, instead of the whole array. Then, these parts would be exchanged between nodes using MPI and sent back to the device. Although the non-optimized version of the code worked well for the 2D advection equation, its optimization seems to be vital for efficient calculations in the case of more complex systems.

The authors express their thanks to all referees for their effort and time spent on reading the draft and providing the stimulating comments, which resulted in a significant improvement of the draft.

References

LEIST, A., PLAYNE, D.P., HAWICK, K.A. (2009) Exploiting graphical processing units for data-parallel scientific applications. *Concurrency and Computation: Practice and Experience* **21**, 2400-2437.

LEVEQUE, R.J. (2002) *Finite-volume Methods for Hyperbolic Problems.* Cambridge University Press, Cambridge.

MORTON, K.W., Mayers, D.F. (2005) *Numerical Solution of Partial Differential Equations, An Introduction.* Cambridge University Press, Cambridge.

MURAWSKI, K. (2002) *Analytical and Numerical Methods for Wave Propagation in Fluids.* World Scientific, Singapore.

MURAWSKI, K., LEE, D. (2012) Godunov-type algorithms for numerical modeling of solar plasma. *Control and Cybernetics* **41** (1), 35-56.

MURAWSKI, K., Jr., MURAWSKI, K., STPICZYŃSKI, P. (2012) Implementation of MUSCL-Hancock method into the C++ code for the Euler equations. *Bull. Pol. Ac.: Tech.* **60** (1), 45-53.

MURAWSKI, K., MURAWSKI, K., Jr., SCHIEVE, H.-Y. (2013) Numerical simulations of acoustic waves with the graphic acceleration GAMER code. *Bull. Pol. Ac.: Tech.* **60** (4), 787-792.

NICKOLLS, J., DALLY, W.J. (2010) The GPU Computing Era. *IEEE Micro* **30**, 56-69.

NICKOLLS, J., BUCK, I., GARLAND, M., SKADRON, K. (2008) Scalable Parallel Programming with CUDA. *ACM Queue* **6**, 40-53.

SCHIEVE, H.-Y., TSAI, Y., CHIUEH, T. (2010) GAMER: a graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. *Astrophys. J. Suppl.* **186** (2), 457-484.

TORO, E. (2009) *Riemann Solvers and Numerical Methods for Fluid Dynamics.* Springer, Berlin.

TÓTH, G., ODSTRCIL, D. (1996) Comparison of Some Flux Corrected Transport and Total Variation Diminishing Numerical Schemes for Hydrodynamic and Magnetohydrodynamic Problems. *J. Comput. Physics* **128** (1), 82-100.

WASILJEW, A., MURAWSKI, K. (2013) A new CUDA-based GPU implementation of the two-dimensional Athena code. *Bull. Pol. Ac.: Tech.* **61** (1), 239-250.