Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 441–446

Towards an Efficient Multi-Stage Riemann Solver for Nuclear Physics Simulations

Sebastian Cygert,
Joanna Porter-Sobieraj
Warsaw University of Technology
Faculty of Mathematics and Information Science
Koszykowa 75, 00-662 Warsaw,
Poland
Email: j.porter@mini.pw.edu.pl

Daniel Kikoła
Purdue University
Department of Physics
525 Northwestern Ave.,
West Lafayette, IN 47907,
United States
Email: dkikola@purdue.edu

Jan Sikorski,
Marcin Słodkowski
Warsaw University of Technology
Faculty of Physics
Koszykowa 75, 00-662 Warsaw,
Poland
Email: slodkow@if.pw.edu.pl

***Abstract*—Relativistic numerical hydrodynamics is an important tool in high energy nuclear science. However, such simulations are extremely demanding in terms of computing power. This paper focuses on improving the speed of solving the Riemann problem with the MUSTA-FORCE algorithm by employing the CUDA parallel programming model. We also propose a new approach to 3D finite difference algorithms, which employ a GPU that uses surface memory. Numerical experiments show an unprecedented increase in the computing power compared to a CPU.**

I. Motivation

NUMERICAL hydrodynamics has been used in many scientific and engineering applications for decades, mostly due to its relative simplicity. Even a complicated, dynamic system can be described with a small set of relatively simple hyperbolic conservation laws in this framework. All the information about the physical properties of a system is contained in a single equation of state, which is the relationship between the thermodynamic properties of the analyzed system. Knowledge of the details of the interactions that take place on the microscopic level is not required. However, there is one strong assumption underlying the use of hydrodynamics: the system has to be at least in the local thermodynamic equilibrium, which means that the thermodynamic quantities for any point are approximately constant around that point.

Recently relativistic hydrodynamics has been applied in studies of a new field of physics: high energy nuclear science. The goal of high energy nuclear science is to study the interactions between the basic constituents of matter; quarks and gluons. In normal conditions, quarks and gluons are bound together to form nucleons: protons and neutrons. However, the forces binding quarks together can be subjected to a sufficiently high energy density, leading to a transition from ordinary nuclear matter to a new state, where quarks and gluons behave like quasi-free particles. Such a *soup* of quarks and gluons is called a Quark-Gluon Plasma (QGP), and it is hypothesized to have existed in the early universe, a few millionths of a second after the Big Bang. The energy density of nuclear matter created in relativistic heavy ion collisions at the Relativistic Heavy Ion Collider (RHIC) at Brookheaven National Laboratory, or the Large Hadron Collider (LHC) at CERN, is sufficiently high that a phase transition to a QGP is expected in such reactions.

In the collisions of heavy nuclei at RHIC or LHC, a decrease in the amount of nuclear matter occurs with exposure to the extreme energy density. The first phase of collisions consists of interactions with a large momentum transfer. Then the system expands, equilibrates and forms the QGP. Relativistic hydrodynamics can then be employed to extract the properties of the QGP. Relativistic ideal fluid dynamics has indeed been used in the theoretical modeling of the QGP and remarkable agreement between experimental data and simulations has been found, leading to the unexpected conclusion that *hot* nuclear matter behaves like a nearly frictionless liquid (i.e. with extremely low viscosity) [1], [2].

The initial success of relativistic hydrodynamics stimulated further development of the numerical codes needed for a more precise understanding of the fundamental properties of the quark and gluon dynamics in the QGP.

Firstly, fully (3+1)-dimensional simulations (3 spatial dimensions + time) are necessary to describe the system's evolution without any assumptions regarding its symmetries. Such numerical problems are well defined and numerical methods are available; however, (3+1)-dimensional simulations are extremely expensive in terms of computing power. Moreover, the fluctuations in the pre-QGP phase might have an impact on the dynamics of the QGP, therefore *event-by-event* studies (where an *event* is a single collision between heavy ions) with fluctuating initial conditions and with a large amount of statistics are of particular interest and require a fast and efficient computer code. Furthermore, studies of jets (which are narrow beams of particles with a high momentum) and their propagation through the hot nuclear medium can provide information about the fundamental properties of the QGP (transport coefficients, for instance). However, such studies require an accurate representation of relativistic flows and shock waves. This requires a larger numerical grid in the simulations, which in turn increases the computing time

This work was supported in part by Dean's Grant (2012) at the Faculty of Physics Warsaw University of Technology

978-1-4673-4471-5/$25.00

needed significantly.

Due to all these factors, there is a constantly increasing demand for computing resources in relativistic hydrodynamics simulations. Graphics Processing Unit (GPU) computing is a promising solution for this problem, and offers an unprecedented increase in computing power compared to standard CPU simulations. In this paper, we present the concept of an implementation of 3+1 ideal hydrodynamics simulations carried out on a GPU using an NVIDIA CUDA framework and test results for selected physics problems.

II. Equation System for the Riemann Problem

A. Hydrodynamics Equations

Equations of relativistic hydrodynamics can be written in a conservative form:

$$\frac{\partial U}{\partial t}+\frac{\partial F(U)}{\partial x}+\frac{\partial G(U)}{\partial y}+\frac{\partial H(U)}{\partial z}=0 \qquad (1)$$

where $U=(E,M_x,M_y,M_z,R)$ is a vector of conserved quantities in the *laboratory rest frame*, E is the energy density, M_i is the momentum density in the i-th Cartesian coordinate and R is a conserved charge density (e.g. a baryon number). F,G,H are vectors of fluxes of those quantities in the x,y,z directions, defined as:

$$F(U)=\begin{pmatrix}(E+p)v_x\\ M_xv_x+p\\ M_yv_x\\ M_zv_x\\ Rv_x\end{pmatrix}$$

$$G(U)=\begin{pmatrix}(E+p)v_y\\ M_xv_y\\ M_yv_y+p\\ M_zv_y\\ Rv_y\end{pmatrix} \qquad (2)$$

$$H(U)=\begin{pmatrix}(E+p)v_z\\ M_xv_z\\ M_yv_z\\ M_zv_z+p\\ Rv_z\end{pmatrix}$$

where v is the velocity and p is pressure, defined by an equation of state: $p=p(e,n)$; e and n are the energy and charge density in the *fluid rest frame* (i.e. in a frame where velocity vanishes, $v=(0,0,0)$).

B. Numerical Scheme

In numerical applications, all continuous fields have to be represented in a finite number of degrees of freedom, e.g. on a fixed numerical grid. In our program we use a finite-difference scheme on a Cartesian grid. Since non-conservative methods (i.e. methods based on non-conservative variables) have been shown to fail (do not converge to a correct solution) if a shock wave is present in the solution [3], a conservative method is used.

There are two standard approaches to solving the problem (1): *dimensional splitting* and a *finite volume method*. Derivation of dimensional splitting methods is based on Taylor series expansions and may give incorrect results for discontinuous solutions [4], thus a finite volume method was chosen. For a three-dimensional problem, such a scheme reads:

$$\begin{aligned}U^{n+1}_{i,j,k} &= U^n_{i,j,k}+\frac{\Delta t}{\Delta x}\left(F_{i-\frac12,j,k}-F_{i+\frac12,j,k}\right)\\ &+\frac{\Delta t}{\Delta y}\left(G_{i,j-\frac12,k}-G_{i,j+\frac12,k}\right)\\ &+\frac{\Delta t}{\Delta z}\left(H_{i,j,k-\frac12}-H_{i,j,k+\frac12}\right)\end{aligned} \qquad (3)$$

where $U^n_{i,j,k}$ represents a conserved quantity at the discrete time t_n; Δt and Δx, Δy, Δz are time and space steps, respectively, and $F_{i-\frac12,j,k}$, ... , $H_{i,j,k+\frac12}$ are numerical fluxes through cell boundaries.

The central point of a particular scheme is the construction of intercell fluxes $F_{i-\frac12,j,k}$, ... , $H_{i,j,k+\frac12}$. There are two distinct approaches to this problem: the *upwind* and *centered* schemes.

The main feature of upwind schemes is that they explicitly exploit information about wave propagation contained in the equations, usually by solving a one-dimensional Riemann problem locally. The accuracy of such schemes is highly dependent on the choice of a particular Riemann solver, which should ideally be *complete* (i.e. take into account all characteristic fields present in the exact solution).

On the other hand, centered methods do not solve the Riemann problem directly, and therefore are usually simpler and more general, at the cost of accuracy (given that there is a complete Riemann solver available).

In order to obtain a general and accurate algorithm, we use a hybrid MUSTA (MUlti-STAge) approach [5], [6]. This utilizes a centered flux in a predictor-corrector loop, solving the Riemann problem numerically, i.e. without using a priori information about waves.

The algorithm in a one-dimensional case is as follows:

1) In order to calculate flux $F_{i+\frac12}$ we introduce auxiliary variables $U^{(l)}_L$ and $U^{(l)}_R$ and their fluxes $F^{(l)}_L$ and $F^{(l)}_R$.
2) Set $U^0_L=U_i$, $U^0_R=U_{i+1}$.
3) Calculate $F^{(l)}_L=F(U^{(l)}_L)$ and $F^{(l)}_R=F(U^{(l)}_R)$ using (1).
4) Calculate $F^{(l)}_{i+\frac12}$ using a centered flux, $U^{(l)}_L$, $U^{(l)}_R$, $F^{(l)}_L$ and $F^{(l)}_R$. If l reached a predefined value, stop.
5) Solve Riemann problem locally:

$$\begin{aligned}U^{(l+1)}_L &= U^{(l)}_L-\tfrac{\Delta t}{\Delta x}\left(F^{(l)}_{i+\frac12}-F^{(l)}_L\right)\\ U^{(l+1)}_R &= U^{(l)}_R-\tfrac{\Delta t}{\Delta x}\left(F^{(l)}_R-F^{(l)}_{i+\frac12}\right)\end{aligned} \qquad (4)$$

6) Go back to step 3.

One drawback to using such an algorithm is that it is numerically more expensive than other, more conventional, algorithms, for instance the SHASTA (SHarp And Smooth Transport Algorithm) [7], [8].

As a centered flux, we used the FORCE (First ORder CEntered) scheme:

$$F^{\text{force}}_{i+\frac{1}{2}} = \frac{1}{2}\left(F^{\text{lw}}_{i+\frac{1}{2}} + F^{\text{lf}}_{i+\frac{1}{2}}\right) \tag{5}$$

where $F^{\text{lw}}_{i+\frac{1}{2}}$ is the Lax-Wendroff type flux (in terms of MUSTA auxilliary variables):

$$F^{\text{lw}}_{i+\frac{1}{2}} = F\left(\frac{1}{2}(U_L + U_R) - \frac{1}{2}\frac{\alpha\Delta t}{\Delta x}(U_R - U_L)\right) \tag{6}$$

and $F^{\text{lf}}_{i+\frac{1}{2}}$ is the Lax-Friedrichs type flux:

$$F^{\text{lf}}_{i+\frac{1}{2}} = \frac{1}{2}(F_L + F_R) - \frac{1}{2}\frac{\Delta x}{\alpha\Delta t}(U_R - U_L) \tag{7}$$

In a three-dimensional case $\alpha = 3$, but other values may also be considered.

To achieve second order accuracy in space and time, we extend our algorithm with MUSCL-Hancock scheme. The basic idea of this scheme is to use more cells to interpolate inter-cell values and evolve them half a time step. The algorithm is:

1) Replace cell average values U^n_i by a piece-wise linear function inside i-th cell:

$$U_i(x) = U^n_i + \frac{(x - x_i)}{\Delta x}\Delta_i \tag{8}$$

where Δ_i is a slope vector and will be defined later. In the local coordinates the points $x = 0$ and $x = \Delta x$ correspond to boundaries of the cell $x_{i-\frac{1}{2}}$ and $x_{i+\frac{1}{2}}$. The values at these points are $U^L_i = U^n_i - \Delta_i/2$ and $U^R_i = U^n_i + \Delta_i/2$.

2) Propagate U^L_i and U^R_i by a time $\frac{1}{2}\Delta t$:

$$\begin{aligned}
\tilde{U}^L_i &= U^L_i + \frac{1}{2}\frac{\Delta t}{\Delta x}(F(U^L_i) - F(U^R_i)) \\
&\quad + \frac{1}{2}\frac{\Delta t}{\Delta y}(G(U^L_i) - G(U^R_i)) \\
&\quad + \frac{1}{2}\frac{\Delta t}{\Delta z}(H(U^L_i) - H(U^R_i)) \\
\tilde{U}^R_i &= U^R_i + \frac{1}{2}\frac{\Delta t}{\Delta x}(F(U^L_i) - F(U^R_i)) \\
&\quad + \frac{1}{2}\frac{\Delta t}{\Delta y}(G(U^L_i) - G(U^R_i)) \\
&\quad + \frac{1}{2}\frac{\Delta t}{\Delta z}(H(U^L_i) - H(U^R_i))
\end{aligned} \tag{9}$$

3) Use $\tilde{U}^L_i$ and $\tilde{U}^R_i$ as U^0_L and U^0_R in MUSTA.

A simple choice for the *slope* Δ_i in (8) is:

$$\Delta_i = \frac{1}{2}(U^n_{i+1} - U^n_{i-1}) \tag{10}$$

which indeed results in a second-order accurate algorithm. However, as predicted by Godunov's theorem, it has the unpleasant effect of producing spurious oscillations in the vicinity of strong gradients.

To solve this issue, a number of flux limiting and slope limiting methods have been proposed. We employed a slope limiting method; instead of Δ_i as in (10) we use:

$$\tilde{\Delta}_i = \xi(r_i)\Delta_i \tag{11}$$

in (8), where ξ is called the *slope limiter*, and r_i is defined as:

$$r_i = \frac{U_i - U_{i-1}}{U_{i+1} - U_i} \tag{12}$$

There are a number of possible choices for ξ, each with its own characteristics and features. One possibility is the MINBEE limiter:

$$\xi_{\text{mb}}(r) = \max(0, \min(1, r)) \tag{13}$$

and there is another, called SUPERBEE:

$$\xi_{\text{sb}}(r) = \max(0, \min(2r, 1), \min(r, 2)) \tag{14}$$

Introducing non-linearity in the scheme provides less oscillations near high gradients and retains good accuracy in smooth areas of the solution.

III. Implementation Notes

A. GPUs – an Overview

Recent developments in GPU technology have transformed them into very powerful devices offering a notable speed increase compared to traditional CPUs in high performance computing. The reason behind this lies in the difference in structure between these two processors. GPUs use many more resources for arithmetic operations at the expense of cache and flow control.

The basic idea of a GPU is that it is built around an array of Streaming Multiprocessors (SMs). Compute Unified Device Architecture (CUDA) allows a programmer to define a special function *kernel* which is executed on a GPU device by a number of threads which are organized into blocks. Each thread block executes in parallel on a single SM independently and in undefined order.

CUDA threads can access several memory spaces. *Global memory*, which has a very big latency, can be accessed by all the threads. Threads within one block can cooperate through *shared memory*. Shared memory is expected to be much faster than global memory and many applications benefit from using it. Registers, whose limited number is distributed to threads by a streaming multiprocessor, offer the lowest latency. By default, all the variables are placed in registers for as long as the latter are available. When there is a lack of registers, other variables go in the local memory which resides in device memory and provides the same high memory latency. This is called *register spilling* and causes a notable slow down for applications.

The main drawback of shared memory is its limited size (48 KB in contemporary GPUs). This is sufficient for many applications, but for others - such as image processing where millions of pixels are transformed in parallel - it is not. For such cases data may be put into texture memory which resides in device memory and is cached in the texture cache. As a

result, data read from texture memory cost one read from global memory on cache miss, and give almost immediate access otherwise. The texture cache is optimized for 2D spatial locality. Texture memory offers only data reading while surface memory offers both read and write operations.

Threads are executed in groups of 32 parallel threads called warps, of which all execute the same instruction scheduled by the warp scheduler. If, due to conditional sentences, different threads within a warp follow different paths, then the scheduler visits each path taken sequentially, disabling threads that are not active in the current path. This is called branching and effects slow down in execution.

A common bottleneck for many GPU applications is memory access latency due to the limited size of cache, especially when using global device memory. However, those latencies can be hidden by a warp scheduler. At every instruction issue time, a warp scheduler selects a warp that is ready to execute its next instruction, if any, and issues the instruction to the active threads of the warp. It can easily be seen that CUDA performs this most effectively when there are plenty of threads to be executed.

B. Data Organization

GPUs have recently been widely used for many advanced computations. Typical examples have involved 3D finite difference computations using the most popular sliding-window approach, which operates using shared memory [9] - [12]. Some papers have also studied using texture memory to optimize the solution in fluid dynamics [13], [14].

In our approach surface memory is applied to hold the simulation data. Surface memory retains all the benefits of texture memory, but it also works in write mode. It is available for NVIDIA GPUs with a compute capability of 2.x or higher. Currently, surface memory does not support double precision floating-point arithmetic and all the calculations are done solely in single precision.

Surface memory offers several benefits over the popular sliding-window approach. Due to the limited size of shared memory, only a limited amount of the data can be held in it and hence loop tiling has to be performed. An input grid is divided into smaller blocks (*tiles*) that fit into shared memory, the threads copy these parts of the data from global to shared memory and perform computations in the latter. Thus, the shared memory can be seen as a manually managed cache. It can easily be noticed that this results in data access redundancy which can be computed using the formula $(n * m + k(n + m))/(n*m)$, where n and m stand for a block's size and k is the order of stencil [10]. In contrast to shared memory, which is highly limited in size, the maximum number of elements we can bound to surface memory is 65536x32768x2048 [15]. Therefore, in practice surface memory is limited only by its size and in most cases all the data can be kept in surface memory.

Surface memory allows the algorithm to decrease register usage. In our approach, the numerical scheme requires a single cell to contain more than one simple variable. When we multiply the size of a cell by the number of cells needed in stencil computation it is easy to notice that the registers are quite heavily used in this algorithm.

Using surface memory it is also easier to modify and test different numerical schemes. Code which uses shared memory is usually dedicated to just one kind of stencil. Changing the order or *direction* of the stencil results in changing many lines of code. In the case of surface memory, since all data are held in it and global indexing is used, this change may be done at once and handling special cases is kept to a minimum. Hence, it is a more general approach.

C. GPU Algorithm

In this section we present the idea behind the algorithm which uses surface memory.

Algorithm 1 Data processing schema for a thread (i, j) and an order-4 stencil computation

```
for n in 1..N do
  for k in 3..Z-Dimension −2 do
    Load neighbor cells from surface memory.
    Compute cell U(i, j, k).
    Write result to surface memory.
    Synchronize threads.
  end for
end for
```

The presented algorithm was built based on the idea of the sliding-window algorithm. All threads work on a 2D *xy*-slice of the grid, benefiting from optimization for the 2D spatial locality of surface memory, and iterate through the *Z*-axis. The *Compute cell* method refers to the MUSTA-FORCE algorithm. In this algorithm we use two surfaces which are employed alternately for read/write operations. It is possible to use just one surface but this would put more pressure on registers and require additional synchronizations. Since memory usage is not a problem in our simulation we decided to stay with two surfaces being used.

IV. Experimental Results

We examined the performance of the parallel GPU code and compared it to the performance of the sequential CPU code. The goal of our research was just to verify the usefulness of GPUs for solving equations of relativistic hydrodynamics and to approximate the order of magnitude of the possible speed-up. Therefore, we measured the time needed to perform the simulations on a single GPU and a single CPU core. This allowed us to estimate the time required to perform massive physics simulations.

The numerical experiments were executed on an Intel Pentium B960, 2.2 GHz processor with an NVIDIA GeForce 610 1 GB graphics card with Compute Capability 2.1. The figures show the time taken for a hydrodynamic simulation, using the MUSTA-FORCE algorithm approach for various configurations of input data.

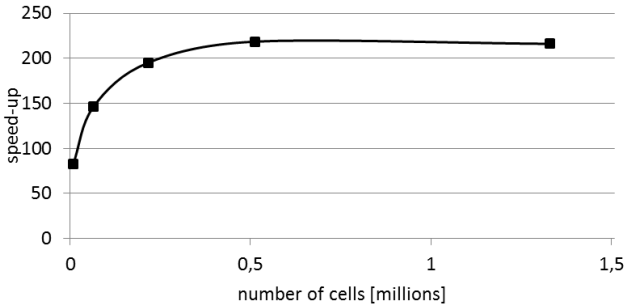

Fig. 1. The speed-up gained by using surface memory compared to CPU implementation for the MUSTA-FORCE algorithm as a function of the total number of cells

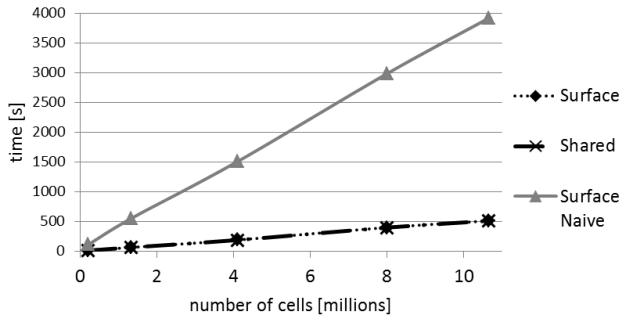

Fig. 2. Execution time for shared memory and surface memory approaches for 100 steps of the MUSTA-FORCE algorithm as a function of the total number of cells

With the GPU, the tests were carried out for 100 time steps of the MUSTA-FORCE algorithm, using grids of dimensions 60^3, 110^3, 160^3, 200^3, and 220^3. A single cell, containing a vector U, occupies 20 bytes of memory. The maximum grid that fitted within the memory limitations was 240^3.

The simulations on the CPU were conducted on smaller grids. We interrupted the calculations for grids when the time exceeded a few hours, because it would have taken days to perform massive tests on the CPU.

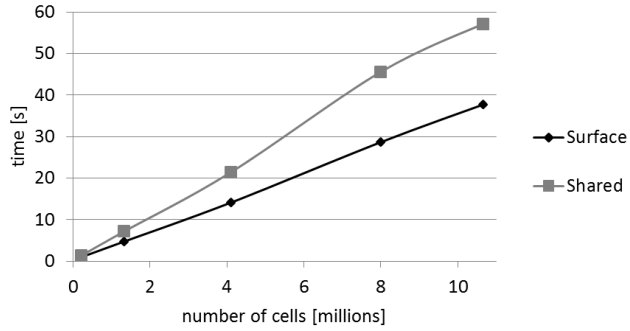

Fig. 3. Execution time for 100 steps of a generalized 3D finite difference algorithm

Fig. 1 shows the acceleration factor gained by using a GPU instead of a CPU. The GPU implementation speed-up is over 200 for bigger numbers of cells. For a grid of size 110^3, the GPU simulation took 1 minute while the CPU required over 3 hours. This proves that finite difference computations are a perfect example of an algorithm that fits the parallel computation concept. The whole algorithm can easily be divided into small parts that single threads can perform in parallel.

Note that the simulation on the CPU is very slow. The estimated effort for 100 time steps and a grid with 220 cells in each dimension is almost 30 hours. Such large grids are necessary in the case of studies of ultra-relativistic flows and strong shocks. Moreover, event-by-event simulations require samples of thousands of such simulations (events). Because of this, the total computing time needed for such analysis (assuming 1,000 events) amounts to as much as 3 years, which makes such a study extremely hard, if not impossible, without parallel computing. This example illustrates how expensive, in terms of computation and memory usage, relativistic hydrodynamic simulations are.

The next thing we examined was various implementations on a GPU. Fig. 2 presents a comparison between the execution time for the sliding-window algorithm using shared memory, and surface memory implementations. For shared memory we used a 16x16 data tile, which we found to be the most effective size. The first, naive surface memory implementation used exactly the same approach as with the shared memory algorithm. Thus it can be concluded that surface memory is about 8 times slower. The revised version of the algorithm with surface memory, presented in section 3c, used all its benefits - keeping all data in surface memory, lower usage of registers, and smaller branching. It decreased the simulation time significantly. As a result the timing of the surface memory and sliding-window approach turned out to be almost identical.

Profiling of the application showed that here we are faced with register spilling which causes a serious slow down. This is due to the fact that the MUSTA-FORCE and MUSCL algorithms we have used, both use many temporary cells interpolated during computations. Now, when there only a maximum of 63 registers per thread, and each cell takes 5 of them, a lot of data need to be kept in local memory. Usage of local memory instead of registers is one of the biggest limitations in current GPUs. Our tests showed that just increasing the size of a single cell kept in memory without any other extra computation cost, made the time of simulation increase 5-fold.

Since one of our goals was to evaluate the effectiveness of the surface memory approach for 3D finite difference computations we prepared another version of the application, which performs an interpolation between the cells instead of the whole MUSTA-FORCE algorithm. Fig. 3 shows that the sliding-window approach is more than 50% slower than the surface algorithm. This proves our thesis that the results in Fig. 2 are affected by register pressure. Profiling of this

application shows that achieved occupancy is higher for the surface algorithm (0.45 instead of 0.32), there is lower usage of registers (in the surface algorithm all data is kept in registers while in the second approach 8 variables are also kept in local memory), and that there is a slightly smaller number of branches (12.5% instead of 14.5% on average). These numbers and the test results prove that despite the fact that surface memory is slower than shared memory, the benefits it offers allow the application to achieve better performance.

V. Conclusion

In this paper the possibilities of using GPUs for developing a solver for the Riemann problem have been examined. We studied two methods of 3D finite difference computation – a sliding-window algorithm using shared memory and our new approach based on surface memory.

First of all, the GPU proved to be a good choice for 3D finite difference computations. Such a problem scales perfectly with parallel computations and thus is very effective. Our implementation is over 200 times faster than a sequential implementation on a CPU. This number shows that graphics cards offer greater computational power for problems that can be divided into independent subproblems.

We have investigated the usefulness of our novel approach using surface memory. Because the amount of memory available is very big all the data can be kept in surface, which thus decreases data redundancy and pressure on registers. On the other hand, shared memory is in general faster than surface memory. As a result, in our application using the MUSTA-FORCE algorithm, both implementations have very comparable speeds. However, as we showed, the results were affected by register spilling caused by the high memory cost of the algorithm used. To investigate the effectiveness of surface memory in 3D finite difference methods we prepared a simplified version of an algorithm that minimized register usage. As a result, the surface memory approach turned out to be faster than the one with the sliding-window approach. It should also be stressed that surface memory implementation is more general and, in contrast to the shared memory approach, can easily be changed to use any other kind and order of isotropic, or anisotropic, stencil in any direction.

In this paper we showed that GPUs are very effective for hydrodynamics simulations in comparison to CPUs. The current GPU implementation allows a device to perform a massive number of simulations in a reasonable time. This was previously impossible, even in our preliminary parallel CPU implementation with the use of a cluster computer and MPI. The designed GPU algorithm, based on surface memory, is easy to modify and proved to be a valuable new tool for high energy nuclear science.

References

[1] J. Adams *et al.*, “Experimental and theoretical challenges in the search for the quark gluon plasma: The STAR Collaboration’s critical assessment of the evidence from RHIC collisions,” *Nucl.Phys.*, vol. A757, pp. 102–183, 2005.

[2] “RHIC Scientists Serve Up "Perfect" Liquid,” Brookhaven National Laboratory Press Release. [Online]. Available: http://www.bnl.gov/newsroom/news.php?a=1303

[3] T.Y. Hou, P.G. LeFloch, Why non-conservative schemes converge to the wrong solutions: error analysis, *Math. of Comput.*, 62: 497-530, 1994.

[4] E. F. Toro, Multi-stage predictor-corrector fluxes for hyperbolic equations. Isaac Newton Institute for Mathematical Sciences Preprint Series NI03037-NPA , University of Cambridge, UK, 2003.

[5] E.F. Toro, MUSTA: A multi-stage numerical flux, *Applied Numerical Mathematics*, 56(10-11), pp.1464-1479, 2006.

[6] E.F. Toro, V.A. Titarev, MUSTA fluxes for systems of conservation laws, *Journal of Computational Physics*, 216(2), pp.403-429, 2006.

[7] J.P. Boris, D.L. Book, Flux-corrected transport. I. SHASTA, a fluid transport algorithm that works, *J. Comput. Phys.*, 11(1), pp.38-69, 1973.

[8] J. Gerhard, V. Lindenstruth, M. Bleicher, Relativistic hydrodynamics on graphic cards, *Computer Physics Communications*, 184(2), pp. 311-319, 2013.

[9] G. Zumbusch: Vectorized Higher Order Finite Difference Kernels, In: Proc. of the 11th international conference on Applied Parallel and Scientific Computing, pp. 343-357, 2012.

[10] P. Micikevicius: 3D fnite difference computation on GPUs using Cuda. In: Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009.

[11] T. Nagaoka, S.Watamabe, A GPU-Based Calculation Using the Three-Dimensional FDTD Method for Electromagnetic Field Analysis in 32nd Annual Internation Conference of the IEEE EMBS Buenos Aires, 2010.

[12] V. Demir, A. Z. Elsherbeni, Compute Unified Architecture (CUDA) Based Finite-Difference Time-Domain (FDTD) Implementation in Aces Journal vol. 25, 2010.

[13] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, *J. Comput. Phys.*, 227(24), pp. 10148-10161, 2008.

[14] E. Phillips, M. Fatica, Implementing the Himeno benchmark with CUDA on GPU clusters. In IEEE International Parallel & Distributed Processing Symposium, pp. 1-10, 2010.

[15] NVIDIA Corporation: NVIDIA CUDA Programming Guide Version 5.0, 2012.