Using GPUs for Parallel Stencil Computations in Relativistic Hydrodynamic Simulation

Sebastian Cygert[1], Daniel Kikoła[3], Joanna Porter-Sobieraj[1], Jan Sikorski[2], and Marcin Słodkowski[2]

[1] Faculty of Mathematics and Information Science, Warsaw University of Technology
[2] Faculty of Physics, Warsaw University of Technology
Koszykowa 75, 00-662 Warsaw, Poland
[3] Department of Physics, Purdue University
525 Northwestern Ave., West Lafayette, IN 47907, United States
cygerts@student.mini.pw.edu.pl, j.porter@mini.pw.edu.pl,
{kikola,slodkow}@if.pw.edu.pl

Abstract. This paper explores the possibilities of using a GPGPU for complex 3D finite difference computation. We propose a new approach to this topic using surface memory and compare it with 3D stencil computations carried out via shared memory, which is currently considered to be the best approach. The case study was performed for the extensive computation of collisions between heavy nuclei in terms of relativistic hydrodynamics. To provide a more detailed comparison between our approach and the one using shared memory we present tests for complex and simplified versions of the algorithm.

1 Introduction

Relativistic hydrodynamics is a theory which provides a simple and straightforward solution to many complicated physical problems, for instance in high energy physics (HEP), high energy nuclear science and astrophysics [1–4]. Even a complicated, dynamic system can be described with a limited set of relatively simple hyperbolic conservation laws in this framework. All the information regarding the physical process is contained in a single equation of state, which describes the relationship between the thermodynamic properties of a studied system. Assuming the collective fluid system, the knowledge of the details of interactions on the microscopic level is not required. In the case of relativistic hydrodynamics, an accurate representation of relativistic flows and shock waves is crucial for a precise description of many important phenomena, for example jet propagation in nuclear hot matter during heavy nuclei collisions (a jet is a narrow beam of particles with high momenta). This requires full 3+1 (the three spatial dimensions + time) dimensional simulations on a larger numerical grid, which are extremely demanding in terms of computing resources.

The hydrodynamic simulation is equivalent to solving a set hyperbolic conservation laws with a given boundary and initial conditions, with additional constraints provided by the equation of state [4, 5]. The equations, which describe relativistic hydrodynamic evolution, have a general form of:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial F_x(\mathbf{U})}{\partial x} + \frac{\partial F_y(\mathbf{U})}{\partial y} + \frac{\partial F_z(\mathbf{U})}{\partial z} = 0 \ , \tag{1}$$

where $\mathbf{U} = [E, M_x, M_y, M_z, R]$ is a vector of conserved quantities in a *laboratory rest frame*, E – energy density, M_x, M_y, M_z – momentum density and R is a charge density (for instance mass or the byron number); $F_x(\mathbf{U})$, $F_y(\mathbf{U})$ and $F_z(\mathbf{U})$ are fluxes defined as:

$$F_x(\mathbf{U}) = \begin{bmatrix} (E+p)v_x \\ M_x v_x + p \\ M_y v_x \\ M_z v_x \\ R v_x \end{bmatrix} \quad F_y(\mathbf{U}) = \begin{bmatrix} (E+p)v_y \\ M_x v_y \\ M_y v_y + p \\ M_z v_y \\ R v_y \end{bmatrix} \quad F_z(\mathbf{U}) = \begin{bmatrix} (E+p)v_z \\ M_x v_z \\ M_y v_z \\ M_z v_z + p \\ R v_z \end{bmatrix} , \tag{2}$$

and p is a pressure. The equation of state has a general form of:

$$p = f(e, n) \ , \tag{3}$$

where e and n are energy density and charge density in a *rest frame of fluid* (i.e. in a frame where velocity $\mathbf{v}$ is vanishing: $\mathbf{v} = [0, 0, 0]$).

In the numerical applications, all continuous hydrodynamic fields have to be represented as discrete quantities on a numerical grid. In our program, we use a finite-difference scheme on a Cartesian grid for the hydrodynamic simulations. In this approach, time evolution in a one-dimensional case for a particular cell i is given by:

$$U_i^{n+1} = U_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+1/2} - F_{i-1/2}\right) , \tag{4}$$

where U_i^n represents a conserved quantity at the discrete time t_n; Δt and Δx are time and space steps, respectively, and $F_{i-1/2}$, $F_{i+1/2}$, are numerical fluxes through the cell boundaries.

The numerical fluxes are obtained by solving a Riemann problem at each cell boundary [4, 5]. We use a second order (in space and time) MUSTA-FORCE approach [6] to obtain $F_{i-1/2}$ and $F_{i+1/2}$. MUSTA-FORCE is a multi-stage (MUlti-STAge) predictor-corrector algorithm which uses a relatively simple central scheme called FORCE for solving a Riemann problem in the intermediate steps. This approach provides excellent accuracy while at the same time being simple and general. The central scheme does not require any assumption to be made about a simulated physical process; therefore it can be used for solving any system of hyperbolic conservation laws. Moreover, the number of MUSTA steps can be adjusted to obtain a shock wave resolution required for a particular study. However, this approach is more expensive in terms of computing power compared to traditional algorithms. A typical test would use 200–300 cells in each direction (overall up to 27 million cells). Therefore, it is necessary to use parallel processing in order to achieve a reasonable simulation time in the case of (3+1)-dimensional simulations. A promising route towards higher efficiency is computing using graphics processing

units, which offer an unprecedented increase in computing power compared to standard CPU simulations.

2 3D Finite Difference Computation on a GPU

2.1 GPU Background

The MUSTA-FORCE algorithm presented in the previous section has been implemented on a GPU with the use of the CUDA parallel programming model. The main concepts and an extensive description of this technology are given in NVIDIA guidebooks [7–9].

The functions executed on a GPU are called *kernels*. Each kernel is executed by launching blocks of threads. Threads from one block run together on the same streaming multiprocessor, each containing streaming processors with on-chip shared memory. All threads have access to common global memory with high access latency. Multiprocessors also contain register memory that guarantees low-latency access that can be completely hidden by the thread scheduler. These registers are partitioned among concurrent threads; when threads perform computations, all the variables are placed by default in registers for as long as they are available. If there is lack of registers, variables awaiting computation are held in local (private for a thread) memory, which is part of the global device memory and which gives the same time penalty for using it. This is called *register spilling* or *register pressure* [8] and slows down the computations. Besides this, a limited amount of shared memory is available for all threads within one block – thus, they can communicate through it. The advantage of using shared memory is that it can be significantly faster than global memory.

Furthermore, CUDA threads are partitioned into *warps*. The threads that comprise a warp are designed to process one common instruction at a time. As a result, when there are conditional sentences in the code and when threads within one warp follow different paths, a warp scheduler needs to issue instructions for all the paths. This unwanted behavior is called branching and therefore, in general, conditional sentences should be used carefully.

The warp scheduler selects a warp that is ready to execute its next instruction. This implies that the CUDA program executes efficiently when there are thousands of threads executed in parallel, so the scheduler can hide the memory latencies.

2.2 Related Works

Using graphical processors to speed up computations of 3D finite difference algorithms is a problem that has recently been well addressed in many articles [10-12]. The general approach, called *sliding window*, is to slice the 3D grid into 2D slices and keep the data shared between many threads in the shared memory. In Fig. 1 we present a 16x16 data grid and halos that are kept in shared memory. Halos are elements that are required by the border cells to compute their value in the next

iteration. Since we use a second order algorithm in space, the width of the area of halos is 2. The next important point is to also keep the 2 cells in front and the 2 behind in registers. When the threads are iterating through the z-axis, they shift their registers and load only the next cell.

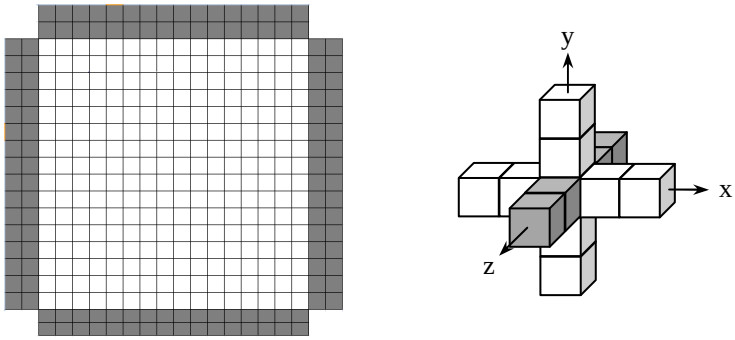

Fig. 1. 16x16 data grid (white) and halos (grey) for the 4th order stencil in shared memory (*left*) and the pattern of data distribution between shared memory (white cells on the xy plane) and registers (grey cells along the z-axis) for a thread for 3D stencil computation (*right*)

This approach - although very good - has several drawbacks.

1. *Problems with halos.* When the grid is divided into the blocks there are always some halo elements needed (as seen on the left of Fig. 1) to proceed with the computations. The border threads of the block are used to load these elements. When the size of the grid does not fit perfectly into threadblocks, the last block, of a smaller size, needs to be taken into account during implementation. There is also another case when the size of the last block is smaller than the order of the algorithm – in this case the corresponding threads need to load more cells than normally. This all complicates the code and increases branching, which should be avoided as denoted in Section 2.1.
   Note that the algorithm presented in the appendix of [11] works properly only when all block are of the same size.
2. *Data redundancy.* With the presented approach the halo elements will sometimes be read more than once. In such a model, the number of reads per cell is $(n*m + k(n + m) / (n*m)$ [11], where n and m stand for a block's size and k is the order of stencil. For the presented example, with 16x16 data tiles and a 4th order stencil, the read redundancy is equal to 1.5. Furthermore, due to the limited maximum size of shared memory per block (48 KB in contemporary GPUs) in each iteration, the data is read from global to shared memory, read from shared memory and then written back to the global memory, which increases the data redundancy factor by 1. This all means that the overall data redundancy in this example is 2.5.
3. *Relatively heavy usage of registers.* In the finite element method each cell contains a number of variables. When the cell size is multiplied by the number of elements needed to be kept in registers, it may turn out that there are no registers left to proceed with other computations. As denoted in Section 2.1,

variables not stored in registers are placed in local memory, which is significantly slower.

Despite these drawbacks, the sliding window algorithm is regarded as the best way to approach 3D finite difference algorithms. However, recent advances in CUDA hardware have also opened up the possibility of using another competing method, based on surface memory.

Texture memory was initially designed for graphics usage where thousands of pixels are processed in parallel. Texture memory resides in device memory and is cached in a special cache which is optimized for 2D spatial locality. This means that a read from this memory costs one memory read from global memory on a cache miss, otherwise access to the data is almost instant. However, texture memory allows for only reading operations, and – in devices of compute capability 2.x and higher – surface memory was introduced to handle read/write operations.

All this means that texture/surface memory performs well in tasks where there is 2D locality in accessing memory. In particular, solving hydrodynamics equations on a grid suits it perfectly, as well as most 3D finite difference algorithms. Texture memory has already been used for solving fluid dynamics [13–15].

2.3 Implementation Notes

The algorithm presented in Section 1 has been implemented on a GPU in two versions – one using shared memory and another using surface memory.

The shared memory algorithm is described below.

Algorithm 1. An order-4 stencil computation using shared memory

```
Load data from host to device global memory.
For t_n in 1..N do
   Load two front, a current and one behind cell into
      registers.
   For z in 3..Z_Dimension–2 do
      Load the second behind cell into register.
      Save a current cell into shared memory.
      For border threads in block do
         Load halos into shared memory.
      End for
      Synchronize threads.
      Load neighboring cells from shared memory.
      Synchronize threads.
      Compute cell.
      Write result to global memory.
      Shift registers.
      Synchronize threads.
   End for
   Send data back to host.
End for
```

It needs to be stressed that the *Compute cell* function is quite complicated. The MUSTA-FORCE algorithm uses many temporary data from interpolated cells. As a result, most variables are sent to local memory because of a lack of registers. Now, due to the operator splitting method we used in integration, and the fact that for most of the interpolated cells the velocity also needs to be computed, this function is called 90 times for a single cell.

Furthermore, in the presented algorithm a cell is a vector of five variables: E, M_x, M_y, M_z and R as mentioned in Section 1. Each of these variables is computed separately. When we multiply the size of the cell structure by the number of overall cells – which in our computations reaches millions – it can very clearly be seen that this algorithm is very expensive in terms of both computations and memory usage.

The algorithm using surface memory was built based on the idea of the previous one. One great thing about using surface memory is that implementation becomes extremely simple compared to that with shared memory. Only the way of accessing specific cells is a little more difficult but it can easily be hidden using some functions. The pseudo-code is presented in Algorithm 2 below.

Algorithm 2. An order-4 stencil computation using surface memory

```
Load data from host to device global memory.
Load data from global memory into surface memory.
For t_n in 1..N do
   For z in 3..Z_Dimension–2 do
      Load neighboring cells from surface memory.
      Synchronize threads.
      Compute Cell.
      Write result to surface memory.
      Synchronize threads.
   End for
End for
Save data from surface memory to global memory.
Send data back to host.
```

Two surfaces are used in this algorithm; both are used for writing and reading alternately. Because surface memory is part of global memory, in most cases a data grid of almost the same size as in the algorithm using shared memory will fit in the memory. It is very clear that using surface memory simplifies the algorithm. In the main loop the algorithm works using just surface memory, so data redundancy is almost nonexistent and fewer synchronizations are needed.

One other important aspect is that surface memory is available for NVIDIA GPUs with compute capability 2.x or higher. This makes implementation with surface memory hardware dependent (in contrast to implementation with shared memory), although we would not consider it to be a problem because these cards have been on the market for over 3 years. It must also be noted that all the calculations are done using single precision floating-point arithmetic. This is important due to the fact that currently surface memory does not support double precision arithmetic. Obviously, if

it were necessary, it could be simulated but it would probably negate most of the advantages of using surface memory.

2.4 Results and Analysis

The numerical experiments were performed on a PC with an NVIDIA GeForce 610 1 GB graphics card. The goal of our research was to compare the effectiveness of two implementations: surface and shared memory for solving 3D finite difference algorithms.

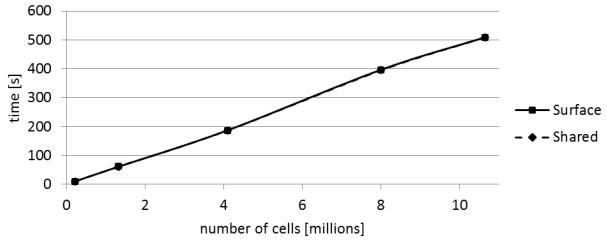

Fig. 2. Execution time for 100 steps of a hydrodynamic simulation using the MUSTA-FORCE algorithm as a function of the total number of cells

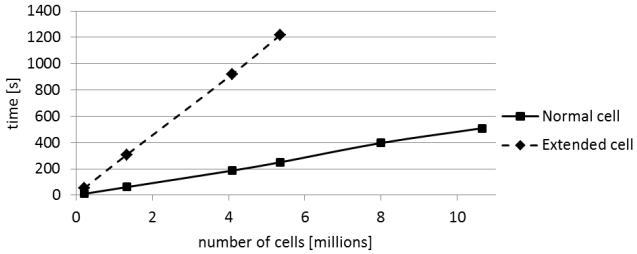

Fig. 3. Execution time for 100 steps of a hydrodynamic simulation using the MUSTA-FORCE algorithm for different sizes of a single cell

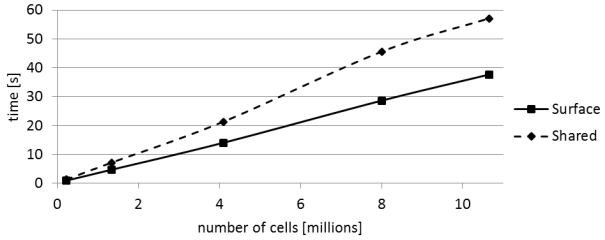

Fig. 4. Execution time for 100 steps of a generalized 3D finite difference algorithm

The figures present the times for a hydrodynamic simulation for various configurations of input data. The tests were performed for grids of dimensions 60, 110, 160, 200, and 220 in each of the three axes and for 100 steps of the algorithm. On the horizontal axis of the graphs the total number of cells in the corresponding grids is presented. The tests were performed using a 16x16 data grid which we found to be the most effective. For comparison, a 32x32 threadblock - which is quite effective in many applications - turned out in our simulation to be two times slower than a 16x16 one. This is caused by the fact that in the MUSTA-FORCE algorithm there is already a lack of registers for threads, so increasing the block size causes a decrease in available registers per thread and as a result it slows down the computations. The maximum grid that fits within the memory limitations for a single cell, including vector **U,** and thus occupying 20 bytes of memory, was 240x240x240.

Fig. 2 shows that the execution times for both implementations are practically the same. However, we must be aware that some properties of this algorithm affect the results. As was described in Section 1, it can be seen that the algorithm is quite difficult, especially because it uses a lot of memory. Due to this fact, we are faced here with quite a big register spill which causes a latency in computations. As a result, more time is spent just loading and saving the variables in local memory instead of doing computations.

To give an overview of how increasing the single cell's size (and thus keeping data in local memory) impacts the time required for simulation, another test was performed. As mentioned in Section 1 the algorithm operates on variables in both – the laboratory frame and the fluid rest frame. Initially, it was convenient to keep all this data within a single cell, however while optimizing the code only laboratory frame variables were preserved. Fig. 3 presents the timing for both cell sizes using surface memory. The *extended cell* implementation used 46 B instead of 20 B for a single cell. It has to be stressed that in both tests we only keep the laboratory frame variables in surface memory, which means that the *extended cell* does not provide any extra fetching from surface memory. Fig. 3 shows that just increasing the size of the cell, without any extra computation cost, causes the algorithm to take about 5 times longer. This confirms the great impact of register pressure on the time needed for computations. Note that the maximum grid size that fit in the memory was just 180x180x180 and thus the tests could only be performed on a smaller number of cells.

To investigate the impact of register spilling on the timing in our application, we prepared a simplified version of an algorithm for a generalized 3D finite difference method. In comparison to the pseudo-code presented in Section 2.3, only the *Compute cell* method has changed. Instead of using the MUSTA-FORCE algorithm we just compute a simple interpolation between the neighboring cells. Fig. 4 shows the time taken for the simplified algorithm. The shared memory implementation is about 1.5 times slower than the one with surface memory for a huge number of cells.

The presented results show that the register spilling we are dealing with in our algorithm influences the timing results. In an ideal situation where there is no register spilling, surface memory is clearly better. In the case of an algorithm which is expensive in terms of used memory, both implementations turned out to be very comparable.

To investigate both algorithms in more detail we used Compute Visual Profiler [8] for the simplified algorithm. The most interesting statistic was the number of registers

used. The maximum number of registers per thread on the tested graphics card was 63. The surface memory algorithm used 43 of them. The shared memory algorithm used all of them and another 32 bytes were transferred to the local memory. This number fits perfectly our thesis in Section 2.2 that the shared memory algorithm is heavily dependent on registers. Another interesting statistic is the control flow divergence [9]. This gives the percentage of thread instructions that were not executed by all threads in the warp, hence causing divergence, which obviously should be as low as possible. In the case of surface memory it is 12.5% while in the case of shared memory it ranges from 13.5% to 15.5%. This is a smaller difference than we expected but surface memory still gives a 15% relative gain. Another quite important statistic is the occupancy achieved. This ratio provides the actual occupancy of the kernel based on the number of warps executed per cycle. This is the ratio of active warps and active cycles divided by the maximum number of warps that can be executed on a multiprocessor. In the case of surface memory this ratio is equal to 0.45 while in shared memory – 0.32. The rest of the statistics were more or less the same for both implementations.

3 Conclusions

In the paper the possibilities of using surface memory for 3D finite difference algorithms have been examined. To study the performance of this novel idea compared to the shared memory method, which is currently considered to be the best approach for this type of problem, two algorithms using surface memory and one using shared memory were implemented.

The main objective was to investigate the usefulness of surface memory, which seems to be a promising approach for complex 3D finite difference methods. The presented algorithm based on using surface memory provides a number of benefits:

1. There is almost no data redundancy. Only the first and the last iteration use the cells twice, while during the whole computation all the data is kept only in surface memory.
2. It ensures lower usage of memory per thread which can be considered a great advantage because registers are significantly faster than local memory.
3. There is a slightly smaller number of branches.

Despite all of these advantages, surface memory turned out to be only slightly faster than the algorithm using shared memory. This is due to the fact that in current graphic cards shared memory is generally the fastest type of memory after the registers.

The speed increase gained by using surface memory depends on the characteristics of the algorithm used. For expensive algorithms like MUSTA-FORCE which we used, the register pressure and computation costs cause the time of computations for both algorithms to be comparable. But, as was proved, for algorithms that are not as expensive the usage of surface memory can result in a speed-up of up to 1.5 times.

Besides these performance related advantages of surface memory, there are also some other benefits. Above all, the implemented code is more general. The code can

be easily changed to use any other kind and order of isotropic or anisotropic stencil in any direction.

The studies performed show that surface memory is a promising alternative for shared memory in 3D finite difference algorithms. CUDA technology is relatively new and advances in graphics cards are proceeding very fast. A possible gain in texture cache would definitely increase the benefits of using surface memory. However, an increase in the amount of shared memory would cause the opposite effect. Either way, surface memory proved to be a very competitive alternative for using shared memory in 3D finite difference algorithms and therefore should also be considered in complex stencil computation.

References

1. Adams, J., et al. (STAR collaboration): 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. A 757, pp. 102–183 (2005)
2. Marti, J.M., Muller, E.: Numerical hydrodynamics in special relativity. Living Reviews in Relativity (2003)
3. Duncan, G.C., Hughes, P.A.: Simulations of relativistic extragalactic jets. Astrophys. J., 436: L119–L122 (1994)
4. Rischke, D.H., Bernhard, S., Maruhn, J.A.: Relativistic hydrodynamics for heavy-ion collisions: general aspects and expansion into vacuum. Nucl. Phys. A, 595: 346–382 (1995)
5. Toro, E.F.: Riemann solvers and numerical methods for fluid dynamics, Springer, Berlin, Germany (1997)
6. 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)
7. CUDA C Best Practices Guide, NVIDIA Corporation (2012)
8. NVIDIA Corporation: Compute Visual Profiler User Guide.
9. NVIDIA Corporation: NVIDIA CUDA Programming Guide Version 5.0 (2012)
10. Zumbusch G., Vectorized Higher Order Finite Difference Kernels, In:·Proc. of the 11th international conference on Applied Parallel and Scientific Computing, pp. 343–357 (2012)
11. Micikevicius, P.: 3D finite difference computation on GPUs using Cuda. In: Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units (2009)
12. Michéa D.,Komatitsch D., Accelerating a 3D finite-difference wave propagation code using GPU graphics cards, Geophys. J. Int. 182 (1) pp. 389–402 (2010)
13. Brandvik, T., Pullan, G.: Acceleration of a two-dimensional Euler flow solver using commodity graphics hardware. In: Proc. of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science, 221(12), 1745–1748 (2007)
14. Elsen, E., LeGresley, P., Darve, E.: Large calculation of the flow over a hypersonic vehicle using a GPU, J. Comput. Phys., 227(24), pp. 10148–10161 (2008)
15. Phillips, E., Fatica, M.: Implementing the Himeno benchmark with CUDA on GPU clusters. In IEEE International Parallel & Distributed Processing Symposium, pp. 1–10 (2010)