

Relativistic Hydrodynamics on Graphic Cards

Jochen Gerhard,^{1,2} Volker Lindenstruth,^{1,2} and Marcus Bleicher^{1,3}

¹*Frankfurt Institute for Advanced Studies, Ruth-Moufang-Straße 1, 60438 Frankfurt am Main*

²*Institut für Informatik, Johann Wolfgang Goethe-Universität,
Robert-Mayer-Straße 11–15, 60054 Frankfurt am Main**

³*Institut für Theoretische Physik, Johann Wolfgang Goethe-Universität,
Max-von-Laue-Straße 1, 60438 Frankfurt am Main*

We show how to accelerate relativistic hydrodynamics simulations using graphic cards (graphic processing units, GPUs). These improvements are of highest relevance e.g. to the field of high-energetic nucleus-nucleus collisions at RHIC and LHC where (ideal and dissipative) relativistic hydrodynamics is used to calculate the evolution of hot and dense QCD matter. The results reported here are based on the Sharp And Smooth Transport Algorithm (SHASTA), which is employed in many hydrodynamical models and hybrid simulation packages, e.g. the Ultrarelativistic Quantum Molecular Dynamics model (UrQMD). We have redesigned the SHASTA using the OpenCL computing framework to work on accelerators like graphic processing units (GPUs) as well as on multi-core processors. With the redesign of the algorithm the hydrodynamic calculations have been accelerated by a factor 160 allowing for event-by-event calculations and better statistics in hybrid calculations.

* jochen.gerhard@compeng.uni-frankfurt.de; Tel.: +4969 798 44112; Fax: +4969 798 44109

I. INTRODUCTION

Relativistic hydrodynamics[1] has a long history since its first application in nuclear collisions[2] and for the evolution of the universe[3]. It is still an excellent tool to describe systems of different scales, ranging from collisions of galaxies down to collisions of heavy ions or even protons at LHC[4]. All these simulations need to respect the relativistic nature of the dynamics, however the focus lies on different aspects. While astronomical calculations have to include viscosity and turbulences, heavy ion simulations can neglect turbulences. Here, algorithms that have the ability to capture shock phenomena or relatively sharp gradients (and potentially very small viscosity) have to be applied. For the present paper we focus on nuclear collisions, however, the methods can also be transferred to cosmological and supernova simulations.

A large body of current numerical tools to explore and interpret the experimental data obtained in high energy experiments employs relativistic hydrodynamics in various facets: On the purely hydrodynamic side there are the models of Heinz and Kolb[5], the model of Romatschke[6], Csernai's[7] hydrodynamic model, and various multi fluid approaches, e.g. by Toneev[8, 9], and Brachmann and Dumitru[10]. In the recent years also hybrid approaches between Monte-Carlo / Boltzmann event simulators and hydrodynamics have gained substantial popularity. Most noteworthy in this respect are the approaches of Hirano and Nara (hydrodynamics+JAM)[11], Bleicher and Petersen (UrQMD 3.3)[12] using the SHASTA, NeXSPheRIO[13] based on smoothed particle hydrodynamics, EPoS[14], Bass and Nonaka[15] realizing a Lagrange approach coupled to UrQMD, the hydro-kinetic model (HKM)[16], and MUSIC[17]. For the numerical solution different methods are applied ranging from smooth particle methods, over the Kurganov-Tadmor[18] algorithm, to the SHASTA[19]. The present examples are based on the SHASTA as it is employed in UrQMD, which offers since version UrQMD 3.3 a hybrid transport model[12], based on the relativistic implementation by Rischke[20]. We shall refer to this implementation by the name of classical FORTRAN implementation, as it is coded completely in FORTRAN 77 compared to our C++ / OpenCL implementation.

Although the reduction of complexity by neglecting turbulences and viscosity reduces the computational demands of the simulation, the running time, typically between 20 minutes and one hour per event for a $(3 + 1)$ dimensional simulation, still prohibits the calculation of high statistics samples. As the SHASTA is well tested and suitable for heavy ion collisions, we have redesigned this algorithm to run on modern accelerator hardware in order to improve the performance. The presented OpenCL implementation allows us to harvest the computational power of accelerators, GPUs, and multi-core processors. After all improvements described in this paper, we gain a speed-up of up to a factor 160 for the full propagation on a GPU, compared to the standard FORTRAN implementation used in UrQMD 3.3 on a CPU. For previous implementations of classical hydrodynamics algorithms on GPUs we refer the reader to [21–24].

II. FORMULATION

The hydro phase of heavy ion collisions is governed in principal by the Euler equations:

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

$$\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\mathbf{v} \otimes (\rho \mathbf{v})) + \nabla p = 0 \quad (2)$$

$$\frac{\partial \varepsilon}{\partial t} + \nabla \cdot (\mathbf{v} (\varepsilon + p)) = 0 \quad (3)$$

With ρ being the density, \mathbf{v} the velocity vector, p the pressure, and ε the energy density. The equations are closed by adding an equation of state. For the heavy ion collision case different equations of state have been proposed and can be used for the simulation. Different models use non analytic equations of state, and provide parametrized equations for the thermodynamical quantities. Various numerical interpolations schemes can be applied here and leave the possibility for additional optimizations on GPUs. For the present numerical study we restrict ourselves to the ideal gas equation of state.

As high energy heavy ion collisions proceed near to the speed of light, relativistic effects have to be taken into account, which reformulates the equations to:

$$\partial_\mu T^{\mu\nu} = 0 \quad (4)$$

$$\partial_\mu J_B^\mu = 0 \quad (5)$$

For the propagation of the energy-momentum tensor $T^{\mu\nu}$ and the baryon current J_B^μ . The energy-momentum-tensor

can be written as:

$$T^{ij} = (\varepsilon + p) \gamma^2 v_i v_j + p \delta_{ij} \quad (6)$$

$$T^{0i} = (\varepsilon + p) \gamma^2 v_i \quad (7)$$

$$T^{00} = (\varepsilon + p v^2) \gamma^2 \quad (8)$$

And taking $J_B^\mu = (\gamma\rho, \gamma\rho\mathbf{v}) = (\mathcal{N}, \mathcal{N}\mathbf{v})$ the energy-momentum-tensor and the Baryon density can be written in terms like mass, momentum and energy:

$$\mathcal{N} = \rho\gamma \quad (9)$$

$$\mathcal{M} = (\varepsilon + p) \gamma^2 \mathbf{v} \quad (10)$$

$$\mathcal{E} = (\varepsilon + p v^2) \gamma^2 \quad (11)$$

Hence, following [25, 26], the relativistic Euler equations take the form similar to the non relativistic differential equations (1), (2), and (3):

$$\frac{\partial}{\partial t} \mathcal{N} + \nabla \cdot (\mathcal{N}\mathbf{v}) = 0 \quad (12)$$

$$\frac{\partial}{\partial t} \mathcal{M} + \nabla \cdot (\mathcal{M}\mathbf{v} + p\bar{\mathbf{I}}) = 0 \quad (13)$$

$$\frac{\partial}{\partial t} \mathcal{E} + \nabla \cdot (\mathbf{v}(\mathcal{E} + p)) = 0 \quad (14)$$

III. NUMERICAL METHOD

The relativistic SHASTA[12, 20] is composed of four phases:

1. Geometric transport.
2. Anti-Diffusion with flux limiter.
3. Relativistic correction.
4. Relativistic calculation of the equation of state.

The geometric transport resembles a finite volume scheme by calculating fluxes between cell boundaries. For an arbitrary quantity U (standing for \mathcal{E} , \mathcal{M} , and \mathcal{N} of the equations (12), (13)), and (14)) the fluxes to neighboring cells are calculated by geometrical factors. These factors are determined by the pressure and velocity of Lagrangian fluid parcells[19]. After the propagation step, the parcells are interpolated back an equidistant lattice with space index j and time index n . The geometrical factors including the interpolation back to the grid are given by[20]:

$$Q_{\pm} = \frac{\frac{1}{2} \mp v_j^{n+\frac{1}{2}} \cdot \lambda}{1 \pm (v_{j\pm 1}^{n+\frac{1}{2}} \cdot \lambda - v_j^{n+\frac{1}{2}} \cdot \lambda)}. \quad (15)$$

With $\lambda = \frac{\Delta t}{\Delta x}$ being the Courant-Friedrichs-Lewy number, and v the propagation velocity, calculated on a staggered grid in time direction.

The propagation of energy and momenta consists not only of the material derivative, but an additional source term f originating from the acceleration of fluid particles by the pressure gradient. Thus, for the propagation of momenta (equation (13)) the additional divergence of the pressure tensor $p\bar{\mathbf{I}}$ and for the propagation of energy (equation (14)) the divergence of velocity and pressure $\mathbf{v}p$ have to be calculated in order to compute a time step. The source term's differential $\Delta(f)$ is also computed on a staggered grid in time direction. We use the central differential $\Delta(f) = -\frac{1}{2}(f_{j+1}^{n+\frac{1}{2}} - f_{j-1}^{n+\frac{1}{2}})$ as proposed in [20]. The purely geometrically propagated quantity \hat{U}_j^n takes the following form:

$$\Delta_j = U_{j+1}^n - U_j^n, \quad (16)$$

$$\hat{U}_j^n = \frac{1}{2}Q_+^2\Delta_j - \frac{1}{2}Q_-^2\Delta_{j-1} + (Q_+ + Q_-)U_j^n + \lambda\Delta(f). \quad (17)$$

As the geometric transport produces inherently a certain amount of numeric diffusion the next step is to correct the results by an anti-diffusion term¹ A^{ph} . The constraint to the anti-diffusion is not to create new maxima or minima on the grid. Thus, the anti-diffusion itself is limited by a flux limiter depending on a neighborhood of points. With the preceding definitions, a time step is completely determined by:

$$\hat{\Delta}_j = \hat{U}_{j+1} - \hat{U}_j \quad (18)$$

$$A_j^{\text{ph}} = \frac{1}{8} \left(\hat{\Delta}_j - \frac{1}{8} (\Delta_{j+1} - 2\Delta_j + \Delta_{j-1}) \right) \quad (19)$$

$$\sigma = \text{sgn } A_j^{\text{ph}} \quad (20)$$

$$A_j = \sigma \cdot \max \left\{ 0, \min \left\{ \sigma \hat{\Delta}_{j+1}, |A_j^{\text{ph}}|, \sigma \hat{\Delta}_{j-1} \right\} \right\} \quad (21)$$

$$U_j^{n+1} = \hat{U}_j^n - A_j + A_{j-1} \quad (22)$$

In the relativistic case, all quantities are boosted from computational frame to their eigenframe in order to calculate the thermodynamic pressure, the baryo-chemical potential, and the propagation velocity. To obtain these quantities from the energy density ε and the baryon density ρ we employ the ideal gas equation of state. Although the ideal gas equation enables an analytical solution to the calculation of the propagation velocity, we have used a fixed point root finding algorithm to allow for an easy implementation of a tabled equation of state without major changes in the codebase.

A. Algorithm Design

The usage of GPUs had been of great interest to the field of high energy physics even before state-of-the-art programming frameworks, like CUDA or OpenCL, have been developed[27]. In order to harvest best performance of GPUs one has to bear in mind certain architectural constraints of these devices. This implies often a very different approach than classical CPU programming. The implementation in this paper is designed in OpenCL and fitted to an AMD 5870 GPU. Therefore certain optimizations are also different [28] to typical optimizations carried out on NVIDIA GPUs. Although designed for this special kind of GPU the implementation still shows significant speed-up on CPU-only systems. Here additional optimizations, respecting caching and memory layout, e.g. interleaving the 3 dimensional grid variables, would mitigate the losses due cache misses when propagating in y or z direction, allow for further significant accelerations.

The OpenCL-SHASTA consists of a C++-part, managing the memory allocation and enqueueing of the *kernels*. The kernels are routines written in OpenCL and completely run on the GPU or multi-core CPU. Kernels are executed in a parallel manner, and each singular instance of a kernel is called a *work-item*. These work-items are mapped, in hardware, to the stream cores of GPUs and CPUs. The mapping occurs in small groups, whose size depends on the hardware used. The smallest possible group is called *wavefront*. Within each wavefront the execution flow must be uniform, i.e. when branching occurs within a wavefront, all branches are computed serially.

The approach to parallelize on GPUs must bear in mind, that the running program at the end consists not of a few parallel threads, but merely thousands of concurrent execution streams. Yet most of this streams are computing *almost* the same, which ranks this approach as a *Single Instruction Multiple Data* (SIMD) approach. In order to organize these thousands of execution streams the first, and arguably most important steps, are the *problem decomposition* and *domain decomposition*. Both points are somewhat entangled, still first order they can be handled separately.

1. Problem Decomposition

Firstly the dataflow of the algorithm has to be analyzed. Not only the physical quantities, that are finally stored, but also the steps in between and intermediate results of the computation are important. In figure 1 the dependencies within each timestep are illustrated. Before each quantity can be calculated, all the quantities pointed to, have to be computed. The dataflow and the spacial and causal dependencies of variables within build the constraints to any possible parallel computation. The SHASTA performs the propagation of the energy-momentum tensor and the net baryon current. Therefore five physical quantities (\mathcal{E} , \mathcal{M} , and \mathcal{N}) are propagated (equations (9), (10), and

¹ Different anti-diffusion terms are possible. As stated in [20] the most suitable for this case is the so called *phenical* anti-diffusion.

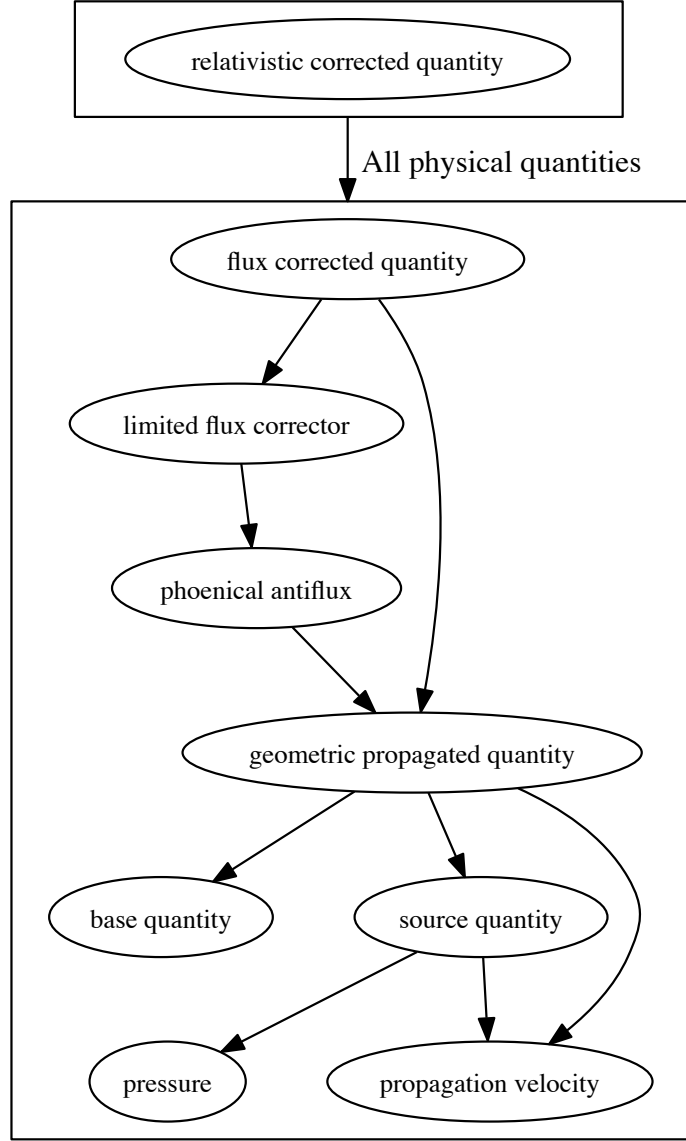


FIG. 1. : Variable dependencies in data flow, time and space indices are omitted. (Each arrow reads as “depends on”)

(11)). The full propagation of each quantity is done by the sequent steps from equation (15) to equation (22). Subsequently relativistic corrections have to be applied, as well as the calculation of the thermodynamic quantities in their eigenframe.

Most of the propagated quantities can be calculated, completely in parallel. As illustrated in figure 2, the propagation of each quantity, including the anti-diffusion and flux corrector, is calculated independently. Though, as the relativistic corrector needs the propagated state of all quantities (see also figure 1), the execution of the relativistic corrector and the calculation of the equation of state is scheduled after all five kernels, propagating the quantities independently, have finished computation. This is realized by the orchestrating method of the GPU-class (listing 2).

Another possible decomposition we have investigated on, is using the kernels firstly to calculate all the geometric propagated quantities (equation (16)) in parallel and subsequently applying new kernels for the anti-diffusion step.

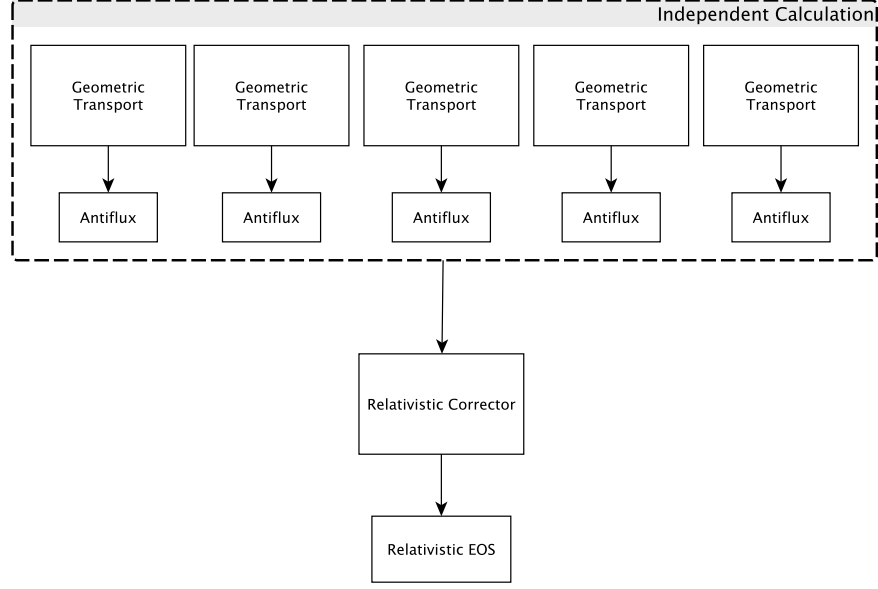


FIG. 2 : Problem decomposition and data flow in OpenCL-SHASTA.

Thereby one kernel can calculate more than one geometric propagated quantity (e.g. all the momenta together) which favors the usage of the vector units of GPUs and modern CPUs². However this approach makes extensive usage of the GPU memory, by storing intermediate results to the global memory and reloading it in the anti-diffusion kernels later. In the execution model of OpenCL a global synchronization between work-units is not possible. Thus, different kernels have to be enqueued to serialize the task.

2. Domain Decomposition

For every grid based algorithm the domain decomposition is often suggested by the grid structure. Though the exact mapping of kernel number to grid size depends on hardware and algorithm type. The current implementation of OpenCL-SHASTA uses a kernel per physical quantity for each of the eight million grid cells. This choice works very well on GPUs. Optimizations aimed mainly on multi-core CPUs would do more efficiently using kernels responsible for more than one cell or more than one physical quantity per cell. Depending on the algorithm type, see section III A 1, the dependencies within each grid cell might lead to even more kernels per grid cell on GPUs. The $(3+1)$ -dimensional problem of relativistic hydrodynamics is perfectly suitable to the three dimensional *NDRange* arguments in OpenCL. In order to implement the $(3+1)$ -dimensional propagation of the energy-momentum-tensor, we use a dimension split approach. Therefore each work-item needs for its calculation only a one dimensional neighborhood. The size of the 1-D differential stencil still depends on the applied scheme, nevertheless this is a significant reduction of the buffer size³ needed. In this implementation each work-item computes independently the geometric flux of all neighboring



FIG. 3. The neighborhood used to compute the propagated quantity in cell j (black) spans seven different cells. To compute the flux corrected value for cell j the geometric transport for cells $(j-2) \dots (j+2)$ (grey) is needed.

² Almost every modern processor offers a right set of vector units (SSE).

³ number of registers

cells in order to calculate its flux corrector and save the flux corrected value. In figure 3 the needed neighborhood of each cell is illustrated: to compute the flux corrected value for cell j (black) each work-item has to calculate the geometric propagated quantity in cells $(j - 2) \dots (j + 2)$ (grey). Therefore additionally the cells $j - 3$ and $j + 3$ have to be used. This is done in the `for`-loop serially and buffered in the `flux[]`-variable. However, the loop increases the computational workload of each kernel significantly, as each geometric flux has to be computed five times more often than in the serial implementation. An additional overhead is caused by the calculation of both limited anti-fluxes, i.e. A_j and A_{j-1} in equation (22) in the variables `ea` and `eb` in listing 1, hence this step doubles the workload compared to the serial implementation. Although the alternative problem decomposition in section III A 1 avoids the multiple calculation of geometric fluxes and anti-fluxes, its overall performance proved to be less due to the additional memory usage. This is because the additional computations allows each work-item to compute the propagated quantity independently from all other work-items, and thus no serialization e.g. between the geometric propagation and the anti-diffusion step is needed.

For each quantity a specialized kernel (similar to listing 1) is enqueued. To use the full capacity of a GPU, the kernel size must be chosen carefully. If the kernel's active working set is too big, only few work-items can run contemporary, as they must share the available buffer memory. (Not only the neighborhood (figure 3) of the propagated quantity is necessary, but additionally each work-item needs to access a similar neighborhood of velocity, pressure, and different control variables. On the other hand, if too many kernels are needed to compute the propagation, the increased number of enqueued kernels propose an additional orchestrating overhead. The use of mid weight kernels, computing all needed fluxes of a neighborhood but propagating only one quantity per kernel, allows for an efficient usage of the underlying hardware (all stream cores in parallel) without overly increasing the orchestration overhead.

3. Branching Free Execution Flow

We have designed the components of OpenCL-SHASTA in a modular way and implemented different kernels and auxiliary functions. This allows shorter development cycles and ensures good validation of the algorithm. Additionally the kernels and auxiliary functions can be substituted even at run time leaving the choice of different anti-flux functions, different source terms, and even different equations of state. In (listing 1) the constant `diff` allows a fine controlled application of numerical viscosity. The value of the constant as well as the selection of more sophisticated or faster anti-diffusion routines are controlled by meta-programming[29]. As the kernels are compiled and loaded onto the GPU at run-time this choices do not infer additional branchings, which would slow the execution down. Nevertheless a free selection of the desired numerical hydrodynamics realization is possible. Therefore a wide variety of calculations can be carried out by the suggested implementation without the need of complicated branching patterns within the computational relevant parts of the program.

Let us stress the importance of avoiding (unnecessary) branching in GPU programs: current GPUs do not offer a *branch predictor*, therefore branching comes generally with a significant penalty on GPUs. (As stated in [28] up to a loss of 30% of the execution speed.) To avoid branchings we have designed specialized kernels for all quantities in OpenCL-SHASTA. The kernels vary according to their propagated quantity, source terms and propagation direction. For example the kernels responsible for the momenta parallel to the propagation direction have an additional source term, whereas momenta perpendicular to the propagation direction are transported source free. We use a number of halo cells to avoid complicated indexing⁴, for the finite difference schemes at the border of the grid. The kernel operates only within the boundaries of the inner grid, limited by grid size (GS) and halo size (HS). For stability reasons the code has been implemented based on a half-step method. The different half steps are implemented without a branching control structure, instead additional kernels change underlying control variables like λ . We use a special double buffering scheme to spare expensive memory copies. Instead of copying onto different grids, we have designed to each kernel call an adjunct kernel call, which reverses the used buffers. The adjunct kernel calls are orchestrated in the launcher method (listing 2) of the GPU-class.

All different governing equations have been implemented in different kernels in order to stay clear of any branches within the execution flow. The orchestration of this set of 30 different kernel calls, is done by the execution method on the host (listing 2). Accordingly, the GPU can start the computation, while the correct kernels are still enqueued into the command queue. No branching within the GPU is necessary.

⁴ Complicated indexing methods not only imply branchings but often a very inefficient memory access.

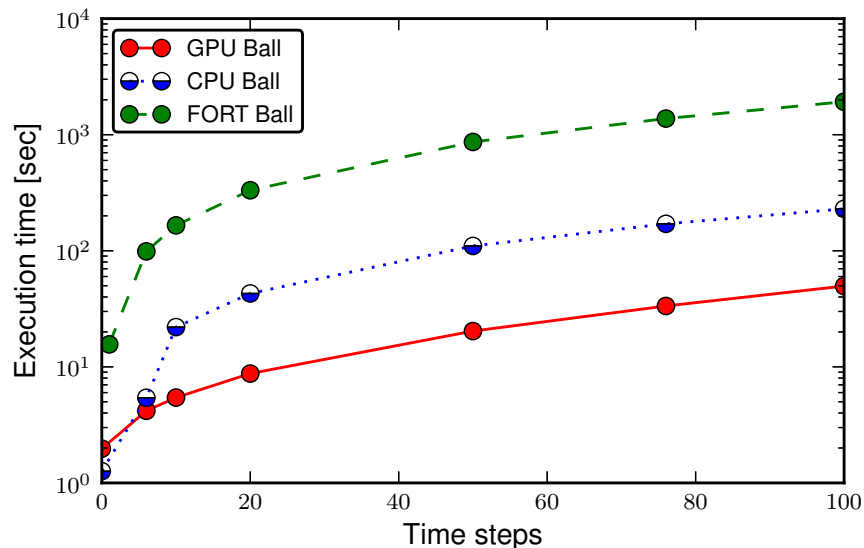


FIG. 4. Total execution time for the expanding ball of $r = 2$ fm with constant energy density. The CPU and GPU are measured with the exact same OpenCL code and compared to the Fortran (FORT) implementation[12, 20].

4. Memory Aware 3-D Calculation

Due to the dimension split scheme each operation is executed only in one dimension at a time. This reduces the amount of registers needed for each kernel drastically, as only 1-D differential stencils (figure 3) have to be calculated. This reduces the needed memory size to a third compared to a full 3-D stencil implementation. The propagation direction follows a fixed permutation of the three axes.⁵ After the initial copy of all needed quantities to its private memory, the kernel (listing 1) does not need any access to global memory for its calculations. On the contrary to the classical FORTRAN implementation no differentials have to be stored. Intermediate results, like the geometric flux, anti-diffusion, and flux-limiter can be stored in registers. The functions yielding the necessary geometric factors (`qpt()` and `qmt()`) and the antiflux (`antiflux()`) are inlined functions. They are calculated exactly when needed (figure 1) and need not to be calculated in advance. (The former mentioned alternative problem decomposition (section III A 1) needs additional global memory for the geometric fluxes, anti-fluxes, and flux limiters.)

According to the permutation scheme only the propagation speed $v_{||}$ parallel to the actual propagation direction is calculated beforehand, which limits the global memory usage to only one field for the propagation speed. This approach underlines again the *recompute instead of memory lookup* paradigm which holds for various applications on many-core architectures[30]. The global memory footprint is determined by the seven quantities residing on a 200^3 cell grid. As only single precision is needed to represent these quantities, the total memory consumption has been reduced: including the double buffering memory scheme, the total consumption is less than 500 MB. This allows to run the code even on commodity GPUs. By limiting the working set to a minimum and concentrating on recompute instead of expensive memory lookups, the remaining memory can be used to increase the grid size, enhance precision or to hold more complicated (tabled) equations of state, or even to enhance SHASTA with the necessary tables to calculate viscous hydro dynamics[31, 32].

IV. RESULTS

We have tested different examples on the LOEWE-CSC cluster using AMD 5870 GPUs and AMD OpteronTM 6172 processors (24 core). Test cases included classic test problems, spheres of different metrics, and realistic initial conditions generated by UrQMD. For realistic cases the physical simulation time is on the order of 10 – 20 fm/c which transforms to 200 time steps (equalling to 16 fm/c in the current setup).

The measured accelerations depend slightly on the geometry of the input. The best acceleration is achieved for the realistic UrQMD input files. Here the overall computing time for a physical running time of $t = 8$ fm/c for an Au+Au

⁵ We found this approach more stable than a Monte Carlo approach.

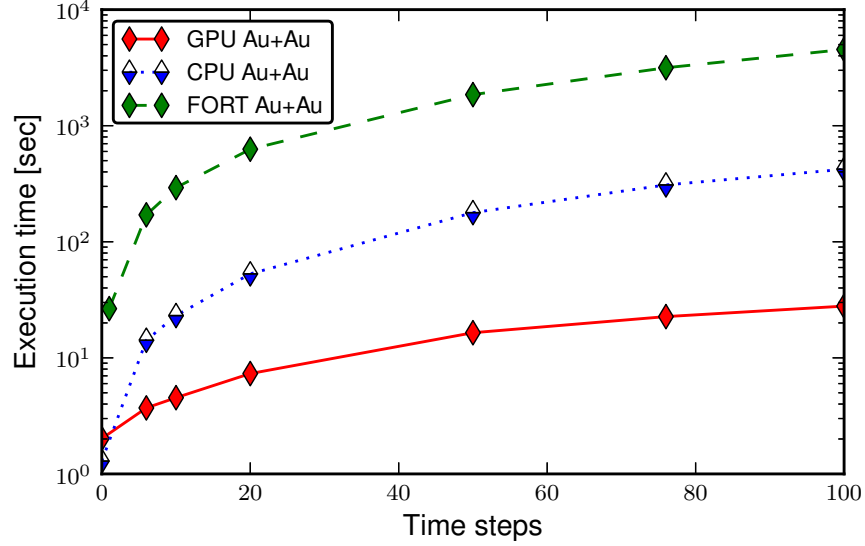


FIG. 5. Total execution time for a Au+Au collision with $\sqrt{s_{NN}} = 200$ GeV. The CPU and GPU are measured with the exact same OpenCL code and compared to the FORTRAN (FORT) implementation[12, 20].

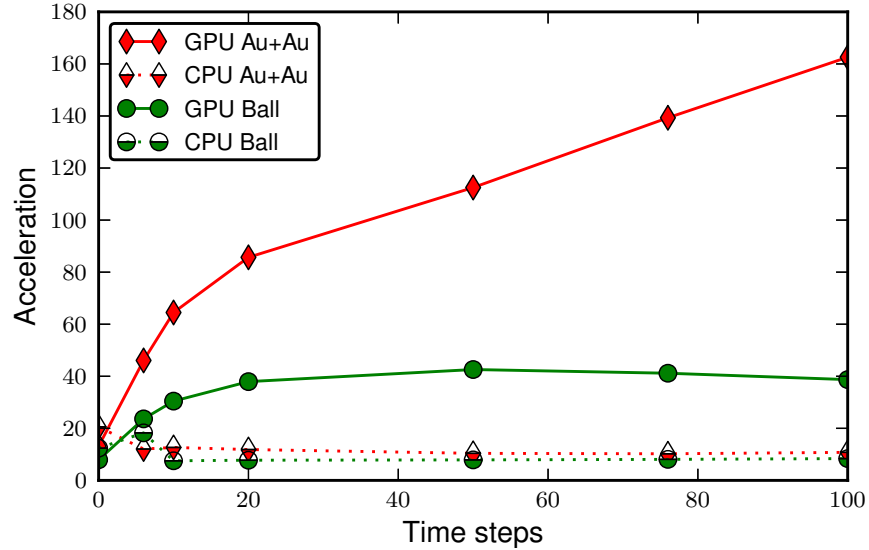


FIG. 6. The acceleration of the execution speed for different initial geometries. Here again the exact same code is executed on CPU and GPU.

collision is reduced to less than 30 seconds. Compared to the standard FORTRAN implementation[12, 20] which needs 1 hour and 15 minutes, we find an acceleration of more than a factor of 160.

In figure 4 and figure 5 the total execution time for two different initial geometries is depicted. Figure 4 shows the comparison for a spherical setup (*ball*, $\|\cdot\|_2$) of radius 2 fm. Figure 5 shows the comparison between the FORTRAN and OpenCL implementation on CPU/GPU for the UrQMD initial configuration for a Au+Au collision at $\sqrt{s_{NN}} = 200$ GeV with impact parameter $b = 7$ fm. Even without a GPU at hand the OpenCL implementation provides a significant speed-up on CPUs. On the AMD OpteronTM 6172 processor (24 core), we measured an acceleration by a factor of ten. Although this is not very efficient, let us stress that a simple parallel execution of the standard FORTRAN implementation is not possible, as the memory consumption of the standard implementation is more than five times higher than the OpenCL implementation. Over the above the exact same implementation on GPU and CPU are compared here. Though the program design allows to choose appropriate kernels and environment variables for their execution, depending on the provided architecture (see section III A 1 and III A 2), enabling further speed-ups on CPUs.

Figure 6 summarizes the increasing speed-up of the execution of the OpenCL implementation on the GPU. The

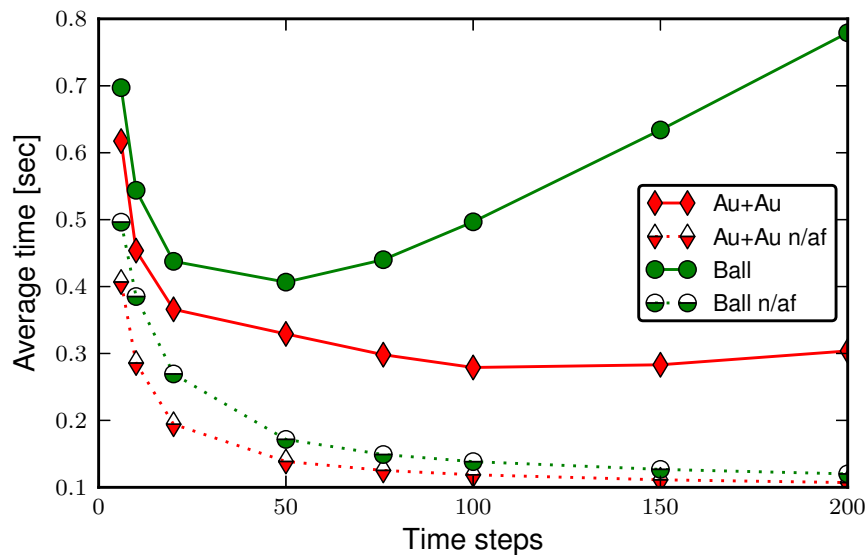


FIG. 7. Average time consumption for one time step in the expansion of a spherical symmetric system (*ball*) and for a Au+Au collision on the GPU. The *n/af* routines show calculations without anti-flux.

difference between the geometries is reflected in the better distribution of computations to work-items. In the spherical symmetric case (denoted as *ball*) the initial geometry is concentrated in the center of the grid, therefore at the beginning all the workload is concentrated on few work-items. During the execution more cells are filled with non-zero values. As each quantity in each cell is computed separately in a work-item, a sparse grid is not very efficient, while a full grid makes best use of all the stream cores of a GPU.

In figure 7 we observe the initial cost of memory transfer to the GPU, which becomes insignificant after a small number of time steps. Nevertheless time consumption increases again later, depending on the problem's geometry. One observes the impact of the complex anti-flux function on the average execution time. Without the complex anti-flux no increase can be observed and the acceleration due filling the grid takes fully place. In SHASTA the anti-flux is corrected by a flux limiter. This flux limiter is calculated by a search of maxima and minima of the surrounding cells and fluxes towards this cells. In this calculation branching is inherent and takes also place within different wavefronts. Therefore all branches within these wavefronts are calculated by the device and the correct results are gained by masking the wrong branches out. Hence the execution time is increased significantly, when the flux limiter is not uniform.

Finally figure 8 shows the direct comparison between the present *single precision* implementation (full line) and the standard FORTRAN implementation (dotted line). For the realistic initial setup of a $\sqrt{s_{NN}} = 200$ GeV Au+Au collision provided by UrQMD we find only minor differences between both implementations. However, we work on a mixed precision implementation of OpenCL-SHASTA. In this implementation we use *double precision* for the less stable parts of the numerics, like the calculation of the boost. This becomes relevant at higher energies.

V. SUMMARY AND OUTLOOK

On current hardware, like the AMD 5870 GPU, double precision calculations come with a slowdown of a factor of five⁶, additionally a full double precision approach doubles the memory consumption. We conclude: a single precision implementation allows for fast calculations, as well as the enhancements of the underlying physical model, e.g. by adding calculations of viscosity [31, 32]. If numerical instabilities occur, e.g. in the calculation of the Lorentz-boost γ we suggest the usage of mixed precision implementations.

We have designed the OpenCL-SHASTA to work on commodity GPUs, nowadays present in almost every computer⁷. As the OpenCL implementation allows for usage on GPUs, accelerators, as well as on classical multi-core processors, the OpenCL-SHASTA can be included in bigger frameworks, that need to be executed on a variety of different architectures.

⁶ A special setup is needed to use double precision as well as the increased memory consumption. These setups are highly hardware dependent and future hardware may be more suited to double precision calculations.

⁷ Even, where no such device is present, the implementation benefits of all present cores and vector units.

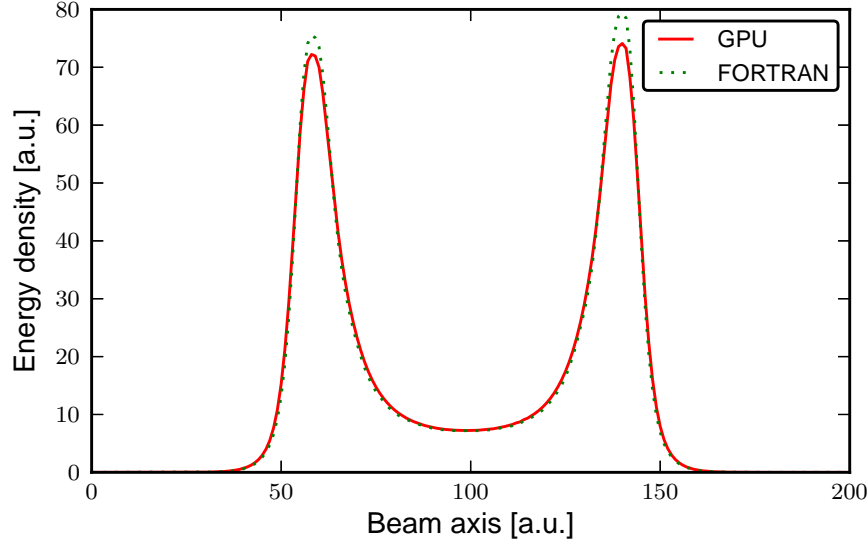


FIG. 8. Comparison between FORTRAN and OpenCL propagation for realistic initial conditions of a Au+Au collision with $\sqrt{s_{NN}} = 200$ GeV and impact parameter $b = 7$ fm at $t = 8$ fm/c provided by UrQMD. (The asymmetry present in both implementations is due to fluctuations in the initial state.)

Due to the tremendous speed-up it resolves the problem of a computationally demanding hydrodynamic phase in hybrid models, like UrQMD, and allows for better statistics, stability analyses[33], and unprecedented event-by-event simulations.

ACKNOWLEDGMENTS

The computational resources were provided by the LOEWE-CSC[34]. This work was supported by HIC for FAIR within the Hessian LOEWE initiative.

-
- [1] J. L. Synge, *General Relativity and Gravitation* **34**, 2177 (2002), 10.1023/A:1021151905577.
 - [2] S. Belen’kii and L. Landau, *Il Nuovo Cimento* (1955-1965) **3**, 15 (1956), 10.1007/BF02745507.
 - [3] J. R. Wilson, in *Sources of Gravitational Radiation*, edited by L. L. Smarr (1979) pp. 423–445.
 - [4] K. Werner, I. Karpenko, T. Pierog, M. Bleicher, and K. Mikhailov, *Phys. Rev. C* **83**, 044915 (2011).
 - [5] P. F. Kolb and U. W. Heinz, (2003), arXiv:nucl-th/0305084 [nucl-th].
 - [6] R. Baier, P. Romatschke, and U. A. Wiedemann, *Phys.Rev.* **C73**, 064903 (2006), arXiv:hep-ph/0602249 [hep-ph].
 - [7] L. Csernai, V. Magas, H. Stöcker, and D. Strottman, *Phys.Rev.* **C84**, 024914 (2011), arXiv:1101.3451 [nucl-th].
 - [8] Y. Ivanov, V. Russkikh, and V. Toneev, *Phys.Rev.* **C73**, 044904 (2006), arXiv:nucl-th/0503088 [nucl-th].
 - [9] V. Toneev and V. Skokov, (2008), arXiv:0809.2413 [nucl-th].
 - [10] A. Dumitru, J. Brachmann, E. Fraga, W. Greiner, A. Jackson, *et al.*, *Heavy Ion Phys.* **14**, 121 (2001), arXiv:nucl-th/0010107 [nucl-th].
 - [11] T. Hirano, P. Huovinen, K. Murase, and Y. Nara, *Prog.Part.Nucl.Phys.* (2012), arXiv:1204.5814 [nucl-th].
 - [12] H. Petersen, J. Steinheimer, G. Baur, M. Bleicher, and H. Stöcker, *Phys. Rev. C* **78**, 044901 (2008).
 - [13] R. Andrade, F. Grassi, Y. Hama, T. Kodama, and O. Socolowski Jr., *Brazilian Journal of Physics* **37**, 717 (2007).
 - [14] K. Werner, I. Karpenko, T. Pierog, M. Bleicher, and K. Mikhailov, *Phys.Rev.* **C82**, 044904 (2010), arXiv:1004.0805 [nucl-th].
 - [15] C. Nonaka and S. A. Bass, *Nucl.Phys.* **A774**, 873 (2006), arXiv:nucl-th/0510038 [nucl-th].
 - [16] I. Karpenko and Y. Sinyukov, *Phys.Lett. B* **688**, 50 (2010).
 - [17] B. Schenke, S. Jeon, and C. Gale, *Phys.Rev.* **C82**, 014903 (2010), arXiv:1004.1408 [hep-ph].
 - [18] A. Kurganov and E. Tadmor, *Journal of Computational Physics* **160**, 241 (2000).
 - [19] J. P. Boris and D. L. Book, *Journal of Computational Physics* **11**, 38 (1973).
 - [20] D. H. Rischke, S. Bernard, and J. A. Maruhn, *Nucl.Phys.* **A595**, 346 (1995), arXiv:nucl-th/9504018 [nucl-th].
 - [21] A. Kolb and N. Cuntz, in *In Proc. of the 18th Symposium on Simulation Technique* (2005) pp. 722–727.

- [22] T. Brandvik and G. Pullan, in *46th AIAA Aerospace Sciences Meeting* (American Institute of Aeronautics and Astronautics, 2008).
- [23] T. Hagen, K.-A. Lie, and J. Natvig, in *Computational Science - ICCS 2006*, Lecture Notes in Computer Science (2006).
- [24] A. Frezzotti, G. Ghiroldi, and L. Gibelli, *Computer Physics Communications* **182**, 2445 (2011).
- [25] L. Csernai, *Introduction to relativistic heavy ion collisions* (Wiley, Chichester New York, 1994).
- [26] J.-Y. Ollitrault, *Eur.J.Phys.* **29**, 275 (2008), arXiv:0708.2433 [nucl-th].
- [27] G. I. Egri, Z. Fodor, C. Hoelbling, S. D. Katz, D. Nogradi, *et al.*, *Comput.Phys.Commun.* **177**, 631 (2007), arXiv:hep-lat/0611022 [hep-lat].
- [28] M. Daga, T. Scogland, and W.-c. Feng, in *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11 (IEEE Computer Society, Washington, DC, USA, 2011) pp. 316–323.
- [29] K. Aehlig, H. Dietert, T. Fischbacher, and J. Gerhard, (2011), arXiv:1110.5936 [hep-th].
- [30] S. Kalcher and V. Lindenstruth, *Cluster Computing*, IEEE International Conference on , 290 (2011).
- [31] E. Molnar, H. Niemi, and D. Rischke, *Eur.Phys.J.* **C65**, 615 (2010), arXiv:0907.2583 [nucl-th].
- [32] H. Niemi, G. S. Denicol, P. Huovinen, E. Molnar, and D. H. Rischke, *Phys.Rev.Lett.* **106**, 212302 (2011), arXiv:1101.2442 [nucl-th].
- [33] J. Gerhard, B. Bauchle, V. Lindenstruth, and M. Bleicher, *Phys.Rev.* **C85**, 044912 (2012), arXiv:1202.5768 [nucl-th].
- [34] M. Bach, M. Kretz, V. Lindenstruth, and D. Rohr, *Computer Science - Research and Development* **26**, 153 (2011), 10.1007/s00450-011-0161-5.

Appendix B: Parallelization Guidelines

Of course no simple and general mechanism of parallelizing algorithms can be stated. Nevertheless, most of the steps we investigated on for the *OpenCL-SHASTA* can be applied to similar algorithms in this field. Therefore we state here a *best practice* approach, we believe can be followed by other research groups, who wish to port their code on GPUs.

1. Problem Decomposition

- Physical quantities, that by superposition are not dependent on each other, are the first choice to be computed in parallel with different kernels. (These are e.g. \mathcal{E} , \mathcal{M} , and \mathcal{N} .)
- If the independence between variables can be gained by a finer time step resolution, the additional execution time of more time steps is easily compensated by the parallelization.
- Not only physical quantities can be parallelized. Also intermediate results within a numerical scheme can often be calculated in parallel.
- A data dependence graph, like figure 1, helps to group calculations to kernels. (Often the connectivity between cliques is a first hint to beneficial cuts)

2. Domain Decomposition

- In grid based algorithms a decomposition to work-items for each grid point is a natural choice. (We group $206 \times 206 \times 206$ work-units per kernel to the queue, for the 200^3 -cell grid, adding halo cells.)
- The aim of the decomposition is to minimize the communication between work-items: numerical schemes, that need a high communication between grid cells might be more efficient with work-items handling small neighborhoods of the grid. (In OpenCL-SHASTA's kernels the initial step is to store **all** necessary quantities in local memory, thus no further communication to other cells is necessary.)
- In OpenCL exists the possibility to group kernels to workgroups, simplifying and accelerating communication within work-items of workgroup.
- Here also a data dependence graph for the spacial dependencies can show how a useful grouping can be done.
- In algorithms not based on a grid, but e.g. test-particles a possible choice is to map the number of particles to work-items.
- The ordering of the particles should mimic the spacial ordering (in phase-space) of the particles, to benefit from potential cache and coalescent memory lookup effects.

3. Kernel Design Criteria

- Depending on the number of work-items that shall be executed in parallel, the active working set of a kernel may not be too big, as the ressources are divided by all contemporary executed kernels. (We chose to compute only **one** physical quantity per kernel, as many fluxes have to be computed. Algorithms with a smaller update step maybe more efficient e.g. by computing the momenta together.)
- Kernels should be as independent as possible, synchronization between virtually thousands of work-items is cumbersome and error prone. (We have chosen to calculate all fluxes needed within each kernel, thus no communication was necessary.)
- A favorable approach is to start with smallest possible kernels and merging kernels whilst measuring the execution time. (For example in the OpenCL-SHASTA we started with extra kernels calculating the geometric flux and kernels calculating the anti-flux. The merge of both kernels proofed to be more efficient.)

4. Hardware Specific Optimizations

- Branching within kernels is most detrimental on GPUs, codes designed for CPUs are less affected by branchings.
- Often the needed branchings in physical models follow a fixed scheme completely determined before execution, e.g. the model's choice of the EoS. Therefore it should be encoded with different kernels and orchestrated in advance. (The kernels for velocity and pressure can easily be substituted before they are loaded to the GPU.)
- Memory access is always a critical point: it should be avoided wherever possible. (Many simulation codes save differentials in arrays, they can be calculated directly)
- Floating point calculations are highly efficient on GPUs. Even transcendental calculations should not be stored in arrays with intermediate results.
- Instead the use of inlined functions (like e.g. `qp` and `qm` in OpenCL-SHASTA) allows for a clean code and an efficient execution.
- Caching might be an issue on special hardware, spacial locality ensures often an efficient usage of present cache structures.
- Coalesced memory access is also often gained by a correctly ordered enqueueing. (Spacial neighbors should have neighboring work-item-ids.)