Institut für Numerische Simulation
Rheinische Friedrich-Wilhelms-Universität Bonn

Wegelerstraße 6 • 53115 Bonn • Germany
phone +49 228 73-3427 • fax +49 228 73-7527
www.ins.uni-bonn.de

P. Zaspel, M. Griebel

Solving Incompressible Two-Phase Flows on Multi-GPU Clusters

INS Preprint No. 1113

October 2011

Solving Incompressible Two-Phase Flows on Multi-GPU Clusters

P. Zaspel[a,**], M. Griebel[a,*]

[a] *Institute for Numerical Simulation, University of Bonn, Wegelerstraße 6, 53115 Bonn, Germany*

Abstract

We present a fully multi-GPU-based double-precision solver for the three-dimensional two-phase incompressible Navier-Stokes equations. It is able to simulate the interaction of two fluids like air and water based on a level-set approach. High-order finite difference schemes and Chorin's projection approach for space and time discretization are applied. An in-depth performance analysis shows a realistic speed-up of the order of three by comparing equally priced GPUs and CPUs and more than a doubling in energy efficiency for GPUs. We observe profound strong and weak scaling on two different multi-GPU clusters.

Keywords:
Graphics Processing Units; Multi-GPU; Two-Phase Flows; Navier-Stokes Equations; Level-Set Method; Finite Difference.

1. Introduction

Moving forward to Exascale computing, the high performance computing (HPC) community has recognized the application of massively parallel hardware as one of the key ingredients to satisfy future computing requirements. One type of this hardware are graphics processing units (GPUs). They are a prototype for a general class of many-core processors with a high thread-parallelism which is expected to dominate future compute clusters. Consequently, there is now a growing number of multi-GPU-based HPC systems for which scientists need appropriate numerical software.

An important area for multi-GPU applications are computational fluid dynamics (CFD) simulations. Commercial CFD package vendors only slowly start to adapt this technology. But in academia, several groups have already published results on multi-GPU codes in CFD, e.g. for Lattice-Boltzmann applications [1, 2], compressible fluids [3, 4] or meteorology [5]. For grid-based incompressible flow simulations, which are described by the Navier-Stokes equations, Cohen and Molemaker were in [6] among the first to show multi-GPU results. They implemented a finite difference/volume code for single machine multi-GPU parallelism. The first truly distributed-memory MPI-based Navier-Stokes solver with finite differences was done by Jacobsen et al. [7]. Later, this group also showed extensions to this solver including a full geometric multigrid [8]. Other multi-GPU implementations of the Navier-Stokes equations are based on finite elements [9, 3].

In our current research, we are especially focused on *two-phase* flow applications based on the Navier-Stokes equations. Kelly presented in [10] a single-GPU accelerated two-phase solver using the level-set method. Kuo et al. [11] accelerated their shared-memory parallel two-phase flow solver by a single-GPU-based Poisson solver. We are also aware of unpublished work on a multi-GPU two-phase flow solver by the group of Aoki et al. at the Tokyo Institute of Technology. However, to the best of our knowledge, there is no publication on a grid-based double-precision fully GPU-based parallel solver for the two-phase incompressible Navier-Stokes equations, which is able to scale on distributed memory multi-GPU clusters.

In this paper, we now present such a two-phase flow solver. It extends the introducing work by the authors in [12]. Like our original in-house CPU fluid solver NaSt3DGPF [13, 14, 15, 16], the new GPU solver uses a finite difference discretization on a staggered grid in complex geometries and the continuum surface force approach to simulate two fluid phases like air and water. The fluid phases are distinguished by a level-set function. A range of applications e.g. in the domain of droplet / bubble dynamics [16] or water ways simulations [14] is thus able to profit from the performance available on large GPU clusters.

Note here that this publication contains several significant improvements and extensions in contrast to the authors' first publication on this topic [12]. The most important improvement is based on the seamless and full usage of GPUs for all computations. Now, besides just the Poisson solver and the level-set reinitialization, also all other GPU parallelizable parts of the described numerical method are completely implemented on GPUs. This eliminates the large overhead of hundreds of GPU↔CPU data transfers as further explained in Section 3.1. Altogether, this leads to a major performance improvement of more than 30 percent compared to the authors paper [12] which is more precisely described in the dedicated Section 4.5. We additionally integrate a methodology based on performance per Watt and performance per Dollar benchmarks, which allows to have more fair performance comparisons, see Section 4.1, 4.4 and 4.7 also with specific focus on Fermi GPU code optimizations. Finally, we show multi-GPU scaling on a 48 Fermi GPU cluster in Section 4.6.

[*] Principal corresponding author
[**] Corresponding author
Email addresses: zaspel@ins.uni-bonn.de (P. Zaspel), griebel@ins.uni-bonn.de (M. Griebel)

The remainder of this article is organized as follows: In **Section 2**, we introduce the governing equations and the numerical methods applied in our solver. **Section 3** presents details of the GPU implementation. Then, in **Section 4**, we review GPU benchmarking in general and give an in-depth performance analysis of the multi-GPU code. In **Section 5**, we discuss our results and provide final conclusions.

The major contributions of this work are as follows:

- For the first time, results of a fully GPU-based double-precision solver for the two-phase incompressible Navier-Stokes equations discretized on a grid using the level-set method are published.
- The solver is an MPI parallel multi-GPU code which scales on multi-GPU clusters. It thus fulfills the requirements for modern HPC systems.
- We present general guidelines for efficient and future-safe GPU implementations and give best practice rules for multi-GPU benchmarking.

2. Governing equations and numerical solution

We describe three-dimensional incompressible two-phase flow problems using the Navier-Stokes equations which are extended by a level-set formulation [17] to cope with phase-dependent densities and viscosities [18]. Surface tension effects at the free surface between the fluid phases are modeled by the continuum surface force method [19]. This approach has been previously described in detail in [16] for our CPU-based fluid solver NaSt3DGPF. We here just give a short sketch of the idea.

The model for two-phase incompressible fluids can be described in a set of equations with

$$\rho(\phi)\frac{D\vec{u}}{Dt} + \nabla p = \nabla\cdot(\mu(\phi)\mathbf{S}) - \sigma\kappa(\phi)\delta(\phi)\nabla\phi + \rho(\phi)\vec{g}, \quad (1)$$

$$\nabla\cdot\vec{u} = 0, \quad (2)$$

$$\partial_t\phi + \vec{u}\cdot\nabla\phi = 0, \quad (3)$$

where equation (1) is the momentum equation with time t, the fluid velocity $\vec{u}$, pressure p and the level-set function ϕ. The level-set function, a signed distance function with $|\nabla\phi| = 1$, implicitly describes the free surface $\Gamma_f = \{\vec{x}\in\Omega \mid \phi(\vec{x}) = 0\}$ and allows to define phase dependent densities $\rho(\phi)$ and viscosities $\mu(\phi)$ by

$$\rho(\phi) := \rho_2 + (\rho_1 - \rho_2)H(\phi),$$
$$\mu(\phi) := \mu_2 + (\mu_1 - \mu_2)H(\phi),$$

$$\text{with}\quad H(\phi) := \begin{cases} 0 & \text{if } \phi < 0, \\ \frac{1}{2} & \text{if } \phi = 0, \\ 1 & \text{if } \phi > 0, \end{cases}$$

$$\text{and}\quad \phi(\vec{x},t) := \begin{cases} <0 & \text{if } \vec{x}\in\Omega_1, \\ =0 & \text{if } \vec{x}\in\Gamma_f, \\ >0 & \text{if } \vec{x}\in\Omega_2. \end{cases}$$

Here, Ω_1 and Ω_2 are the domains of the two fluid phases and ρ_1, ρ_2, μ_1 and μ_2 denote the respective material parameters. The material derivative $\frac{D\vec{u}}{Dt}$ is given by $\frac{D\vec{u}}{Dt} := \partial_t\vec{u} + (\vec{u}\cdot\nabla)\vec{u}$. In the surface tension force term $\sigma\kappa(\phi)\delta(\phi)\nabla\phi$ of equation (1), κ identifies the curvature of the free surface and σ denotes the surface tension coefficient which is a material constant. Furthermore, δ is the Dirac-delta functional. Finally, $\vec{g}$ stands for volume forces e.g. gravity and $\mathbf{S}$ is the stress tensor $\mathbf{S} := \nabla\vec{u} + \{\nabla\vec{u}\}^T$. Equation (2) is the continuity equation and describes the incompressibility constraint. The last equation (3) models the transport of the level-set function i.e. the dynamics of the free surface.

We discretize the above equations with the finite difference method on a staggered uniform grid. For numerical reasons, we apply a smoothing [18] to the Heaviside functional $H(\phi)$ and the Dirac-delta functional in an ε-environment of the free surface. Chorin's projection approach [20] then leads to a solution method which is described in detail in Algorithm 1. Here, we employed a first-order Euler time integration to get a simple formulation. In practical application problems, we use a second-order Adams-Bashforth time integration to compute the intermediate velocity field and to transport the level-set function. The time derivative for the artificial time τ in the level-set reinitialization process is discretized by a third-order Runge-Kutta method. Note that we have to reinitialize the level-set function after each transport step to recover the distance property $|\nabla\phi| = 1$ which is necessary for a correct evaluation of the free surface's normal and curvature κ. All transport terms and the level-set gradient in the reinitialization step are discretized using a fifth-order weighted essentially non-oscillatory (WENO) scheme, while the diffusion term in the second step is computed using second-order central differences. The Poisson equation is discretized by a second order method and solved with a Jacobi-preconditioned conjugate gradient (CG) method for sparse linear systems. Here, the Poisson equation's non-constant coefficients, namely the density jump for two-phase flows, lead to a high condition number for the linear system and thus to a slow convergence of the iterative solver. It is therefore a well-known fact that the Poisson solver dominates the overall run-time for such a solution method.

The full GPU solver supports different kinds of boundary conditions. These include slip and no-slip solid boundaries. Inflows and outflows can be controlled by Dirichlet and homogeneous Neumann boundary conditions for the velocity field. We introduce complex geometries by flagging out some cells of the simulation domain and apply appropriate boundary conditions at the fluid-solid interface cells of the obstacles.

3. Implementation

The GPU implementation is based on the well-known parallel programming extension CUDA from Nvidia. We basically map one GPU thread to one grid point and linearize the three-dimensional grid. All operations are done per-cell with a SIMD (single instruction multiple data) approach.

3.1. Porting process

We started our GPU implementation based on the existing MPI-parallel CPU code NaSt3DGPF. Since our approach in

Algorithm 1 (Chorin's projection approach)

for $n = 1, 2, \ldots$ do:

1. set boundary conditions for $\vec{u}^n$
2. compute intermediate velocity field $\vec{u}^*$:
$$\frac{\vec{u}^* - \vec{u}^n}{\delta t} = -(\vec{u}^n \cdot \nabla)\vec{u}^n + \frac{1}{\rho(\phi^n)} \nabla \cdot (\mu(\phi^n)\mathbf{S}^n) - \frac{1}{\rho(\phi^n)} \sigma\kappa(\phi^n)\delta(\phi^n)\nabla\phi^n + \vec{g}$$
3. apply boundary conditions and transport level-set function:
$$\phi^* = \phi^n + \delta t\,(\vec{u}^n \cdot \nabla\phi^n)$$
4. reinitialize level-set function by solving
$$\partial_\tau d + sign(\phi^*)(|\nabla d| - 1) = 0, \quad d^0 = \phi^*$$
5. solve the pressure Poisson equation with $\phi^{n+1} = d$:
$$\nabla \cdot \left(\frac{\delta t}{\rho(\phi^{n+1})} \nabla p^{n+1} \right) = \nabla \cdot \vec{u}^*$$
6. apply velocity correction:
$$\vec{u}^{n+1} = \vec{u}^* - \frac{\delta t}{\rho(\phi^{n+1})} \nabla p^{n+1}$$

[12] to use the GPU as accelerator for the Poisson solver and the level-set reinitialization showed motivating results, we moved forward and ported now the remaining parts of the fluid solver's main loop step by step (including appropriate GPU $\leftrightarrow$ CPU data transfers) to the GPU. The advantage of this approach is evident. We always had a fully working and previously validated (cf. e.g. [14]) CPU code to compare with, such that the correctness of our GPU implementation could be checked during the porting process by direct comparison of the CPU and the GPU data fields. In fact, we almost always saw equal results up to machine accuracy. The only exception to this were global reduction operations which obviously have slightly different results based on the different summation orders. Nevertheless, we still observe equal results up to discretization error and convergence order.

After we finished the stepwise porting of the solver components we centralized all data fields and removed unnecessary data exchanges. Consequently, we now copy all simulation data to the GPU once at the beginning of the application. Thereafter, almost no GPU $\leftrightarrow$ CPU data transfers are necessary during the simulation. This results in an implementation which largely profits from the existing GPU performance and allows us to work around the typical PCI Express bus data exchange issues as much as possible. Exactly this is the major advantage of a *full* GPU port over just a GPU-accelerated code. In the final implementation, the only remaining CPU-based parts are now the configuration file parser, the data field setup, binary/visualization file input/output and parallel communication.

3.2. Code design

Our GPU implementation follows on some basic principles: First, we always use double precision calculations. This is necessary since we often simulate problems at small scales like millimeters or less which requires high precision. Even though mixed-precision approaches might give improved performance for some parts of the code, we decide to stick to double-precision. We believe that GPU hardware will improve when it comes to double-precision performance in the future anyway.

Furthermore, we are highly interested in keeping the GPU implementation as simple as possible. Thus, we try to minimize memory-hierarchy specific optimizations. We assume that future GPU hardware will be more and more automatically cache-oriented. This view is supported by the latest developments in the market. As a consequence, the only highly memory optimized (thus shared-memory using) parts of the code are global reduction operations. For these, we use modified versions of the reduction kernels from the CUDA software development kit.

3.3. GPU specific optimizations

Even though we stay with this simplified implementation approach which largely avoids shared memory, there are still some optimizations that should be done for high performance on GPUs. For example, we see the behavior of older GPU hardware (i.e. pre-*Fermi* GPUs) to perform less efficient when using GPU functions i.e. *kernels* with a higher amount of calling parameters and a high number of operations: Older GPUs store all kernel parameters in the local *registers* of each GPU thread. We even nowadays see a largely growing register use with growing per-kernel instruction count. This might be caused due to a non-optimal register use by the compiler. Anyway, since the amount of registers per thread is somehow limited, we often have to force the compiler to move data from the very fast registers to *local memory* i.e. the slow general GPU memory, which is often called *register spilling*. This can cause a huge performance loss on older GPUs and might still harm the performance on the newer *Fermi*-type GPUs. To overcome this specific drawback, we split up big computation kernels to smaller ones as already discussed in [12] for the example of the WENO computation kernel.

The *compute configuration* for kernel execution is the mapping of parallel threads to the so-called *symmetric multiprocessors (SM)* of the GPU. Choosing an appropriate mapping for our kind of per-cell parallelism is quite easy. We can often maximize the performance by setting the number of threads per SM (i.e. the *block size*) to 256 or 512.

As already mentioned, we support several types of boundary conditions (slip, no-slip, Dirichlet, Neumann, periodic), a variety of finite difference stencils, complex geometries and we employ boundary cells (i.e. halo/ghost cells) for the parallelization. Therefore, the original CPU code loops over the three-dimensional data fields with a lot of different data access patterns (see Figure 1). However, mapping these patterns to GPU-parallelism is not obvious, since the heavy use of conditionals, which is typically accepted for CPUs, might cause a dramatic

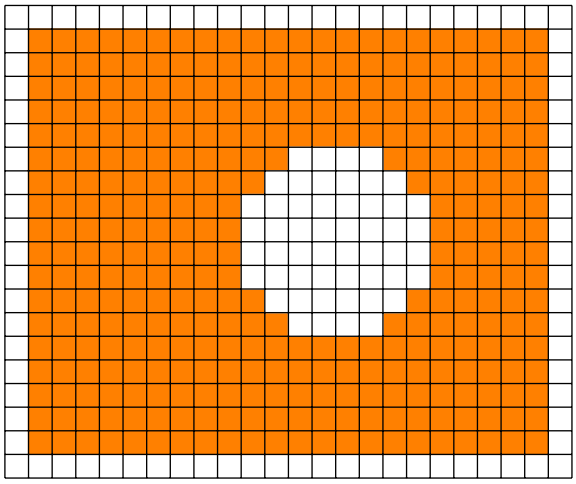

Figure 1: Complex geometries and boundary cells in the finite difference scheme lead to irregular data access patterns for data fields. In this 2D example which includes a discretized circular solid geometry, computations have only to be done for the colored cells.

loss in performance on the GPU. To work around this, we precompute appropriate Boolean data field access patterns once on the CPU before the main compute loop and move them to the GPU. Consequently we just have to include one conditional and one additional global read per thread to handle customized data accesses. Our measurements show a performance gain of about 25 percent for this approach in the relevant GPU kernels which results in an improved overall code run time. Obviously, this leads to a slightly higher memory use on the GPU.

3.4. Moving forward to multi-GPU

In standard MPI-based CPU parallelization implementations we always have to copy data from the system memory to the network adapter which then performs the actual data transfers. For parallel multi-GPU implementations, we additionally have to transfer the data between the GPUs and the CPUs. Furthermore, there is an uneven balance between the actual floating point performance of GPUs and the GPU↔CPU data transfer rates. That is, GPUs can compute much faster than we can transfer the data to the CPU. This performance bottleneck heavily hampers all multi-GPU parallelizations if there is no special optimization for a good scaling taken into account.

The authors' introducing work [12] on accelerating the discussed fluid solver with multi-GPU support already outlined the important steps of such an optimization. We here shortly recall these. One key step is to overlap computation and communication as e.g. proposed in [21]. Modern GPUs are able to exchange data with CPU memory while performing computations. They can also compute in a non-blocking way, i.e. CPUs and GPUs can compute independently. The idea is to perform a time-intensive GPU computation while the full data exchange between GPU, CPU and the network interconnect is done. Thus, one is able to hide most of the time needed for the data exchange *behind* the GPU computation. We can apply this strategy in our iterative solver for the Poisson equation, where we overlap the large sparse matrix-vector product with the data field's boundary data exchange. The only non-overlapped part here is the final matrix-vector product application on the boundary cells which is done after the data transfers. Overall, this method leads to a very nice scaling for the Poisson solver and the full multi-GPU code, cf. Section 4.6. On pre-Fermi GPU hardware, we are not able to apply this approach to the remaining parts of our code, since here, the boundary-cell-only application for other compute kernels is too costly. However, an appropriate code optimization for a Fermi-based GPU cluster is future work.

Another important multi-GPU optimization step is an efficient reordering of scattered boundary data into one continuous data block in the fast GPU memory before data is copied to the CPU memory. Of course, we use so-called *pinned memory* for all our data exchanges to optimize the bandwidth.

The overall *multi*-GPU implementation, as already described in [12], is realized by mapping the domains of the original domain decomposition parallelization of the CPU MPI approach directly to the GPU. Thus, one CPU core/process handles one GPU.

4. Performance results

4.1. Best practice in GPU benchmarking

First papers on GPU computing discussed performance by comparing the run-times on one GPU and one CPU core. This comparison of course was not fair. Instead, it is more reasonable to compare one CPU *socket* with one GPU. This means that one compares a multi-core implementation on e.g. four CPU cores with the performance of one GPU. But, even this type of comparison may be questionable if it is not clearly stated which CPUs and GPUs are compared. From the CPU perspective, one might benchmark dual-/quad-/hexa- or even octo-core CPUs which of course are all different in their performance characteristics, but are installed in one socket. Moreover, there exist several generations and classes of GPUs, on the market.

We therefore want to advocate the concepts of *performance per dollar* and *performance per Watt*. In the first approach, we compare *similar priced* hardware. To this end, it is still unclear if the price of two full systems or just the hardware price of GPUs and CPUs should be opposed. On one hand, one always has to buy a CPU if one wants to use a GPU. On the other hand, one can buy a quite cheap CPU (up to an order of magnitude less expensive) and apply it just as a controller for the GPU. A perfectly fair system-to-system comparison could only be done using a mixed GPU and multi-core CPU code. To shorten the discussion, we focus here only on the direct hardware price comparison of one GPU and appropriate CPUs, since for highly specialized GPU systems, the controlling CPUs in general cost less than ten percent of the GPU price. We believe that this performance per dollar analysis allows us to have a quit realistic look at the real commercial advantage of GPU computations.

The other metric for performance should be *performance per Watt*. That is, we measure and compare the required power consumptions for a given simulation task. We expect this approach

to become more and more important especially in connection with the *Green IT* discussion. Beside of the obvious environmental advantages of smaller carbon footprints for large data centers, power consumption is clearly related to energy costs. This is why large companies will be more and more interested to acquire power efficient hardware. Furthermore, ongoing Exascale projects address power consumption as a key issue for the next generation of compute clusters since the current processor technology will not scale due to power limitations. We therefore also perform a power consumption analysis for our benchmarking hardware and the developed multi-GPU code.

4.2. Benchmark setup

For a performance analysis, we use the following GPU and CPU setup: Our first in-house GPU system is an 8-GPU-cluster built by two workstations with an Intel Core i7-920 CPU at 2.66 GHz which are each attached to a Nvidia Tesla S1070 GPU system containing four GT200 Tesla GPUs (which are effectively four M1060 GPUs). The compute nodes are connected by a Mellanox ConnectX QDR 40G InfiniBand interconnect. Since the described GT200 GPUs have (at the time of writing this paper) each a price which is comparable to an Intel Xeon X5650 six-core CPU with 2.66 GHz, we take a standard 1U compute node with this CPU for comparisons in our performance per Dollar analysis. The second in-house GPU system contains a single Nvidia Tesla C2050 GPU with the newer *Fermi* generation GF100 GPU core which costs approximately twice the price of the M1060 GPUs. For the perf/Dollar benchmark, we therefore compare it with a dual-socket system that has *two* Intel Xeon X5650 CPUs with in total 12 CPU cores. All in-house benchmarking systems use Ubuntu Linux 10.04, OpenMPI, the GCC 4.4 compiler and the Nvidia CUDA Toolkit 3.2. We compile the GPU code with the optimization flag `-O3` and the CPU code with `-O3 -march=native`. To perform speed-up and scale-up tests for a larger GPU count, we also got access to the GPU cluster of the Center for Computing and Communication at the RWTH Aachen University, Germany. We use up to 48 Fermi-type GPUs, Nvidia Quadro 6000. They are installed in double-GPU systems with two Intel Xeon X5650 CPUs which are connected by QDR InfiniBand. The software stack is Scientific Linux 6.1, GCC 4.4.5, OpenMPI 1.5.3 and CUDA 4.0.17.

Our benchmarking problem is (similar to [12]) a rising bubble of air inside a tank of water with surface tension effects (see Figure 2). The bubble has a diameter of 6 cm and the water tank has a side length of 20 cm. In our simulations, the bubble becomes torus-shaped over time, which complies with experimental results in [23]. This kind of example is both representative for some of our applications and computationally challenging. We always measure wall-clock times including all required timings for GPU↔CPU data transfers. Computations on the CPU and on the GPU are performed in double precision. Since our fluid solver uses adaptive time step sizes, we measure the runtime required in the main loop to compute a fixed number of 20 time steps instead of a fixed simulated physical time to get comparable results. A calculation of 20 time steps of our model problem at a grid resolution of $256 \times 256 \times 128$ takes about 51 minutes on the full six-core Xeon processor.

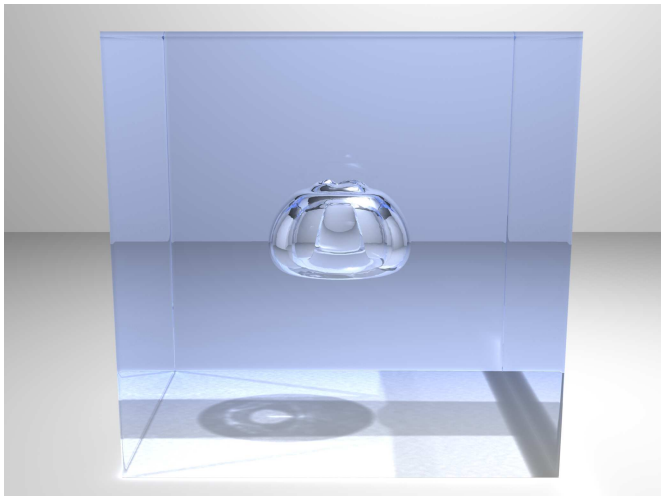

Figure 2: Our performance benchmark example is an air bubble rising in water. The visualization was done following [22].

4.3. Performance expectations

The original CPU-based fluid solver NaSt3DGPF is a research code which has been developed over more than a decade. It is presently applied in various cooperations and projects. Consequently, this code is optimized for performance only to a point such that it is still readable and expandable. On the other hand, the new GPU code has been developed over a few months in net effective development time and is also not over-optimized in the sense of heavy shared-memory use or micro-benchmarking. We therefore think that this is a sufficiently fair base for a performance comparison.

We now look at the different hardware platforms to judge the results presented in the next sections. Table 1 gives the theoretical peak performance values in double-precision floating-point operations and memory bandwidth. Note that we used non-ECC results for the GPUs. ECC-protected GPU calculations have a reduced memory bandwidth by at least 8 percent. The table also specifies the speed-up ratios between the hardware that we will compare in our test. As a matter of fact, the direct comparison of raw theoretical peak performances is often misleading, since these numbers do not reflect real application performance. However, it allows us to get a first feeling for the achievable speed-ups. One interesting result is the limited raw double-precision floating-point performance for the GT200 GPUs. However, since the most time consuming part of our code, i.e. the sparse iterative solver, is typically memory-bound, the corresponding memory bandwidth differences are expected to be the dominant performance limiters. Therefore, we should be satisfied if we see speed-ups in the range of two to three in our performance per dollar analysis.

4.4. Single-GPU speed-up analysis

Figure 3 shows the single-GPU speed-ups for the older GT200-based GPU and the new Fermi generation GPU. We get

CPU/GPU type	double-precision floating-point peak performance	ratio	peak memory bandwidth	ratio
Intel Xeon X5650	∼64 Gflops		32 GB/s	
Nvidia M1060 (no ECC)	78 Gflops	1.2x	102 GB/s	3.2x
two Intel Xeon X5650s	∼128 Gflops		64 GB/s	
Nvidia C2050 (no ECC)	515 Gflops	4x	148 GB/s	2.3x

Table 1: CPUs and GPUs have different performance characteristics. The ratios reflect the speed-ups between the grouped CPUs and GPUs with respect to double-precision floating-point performance and memory bandwidth. Gflops numbers for the CPUs are accumulated values for the 6 / 12 physical cores and were extracted from the Top500 list [24].

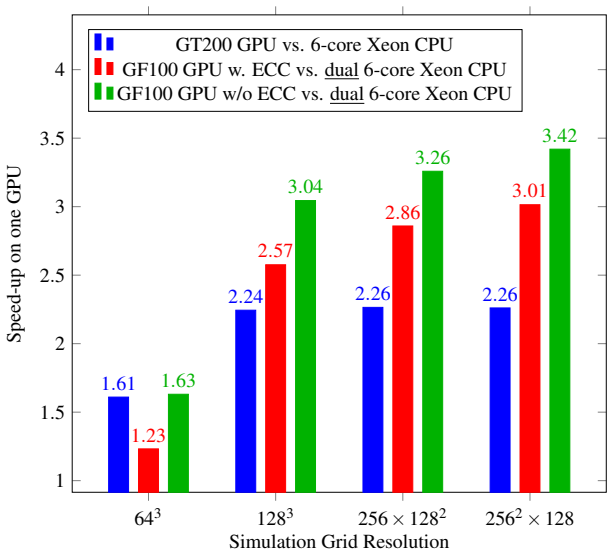

Figure 3: Speed-ups of single GPUs relative to *similar-priced* multi-core CPUs for different resolutions and different GPU generations.

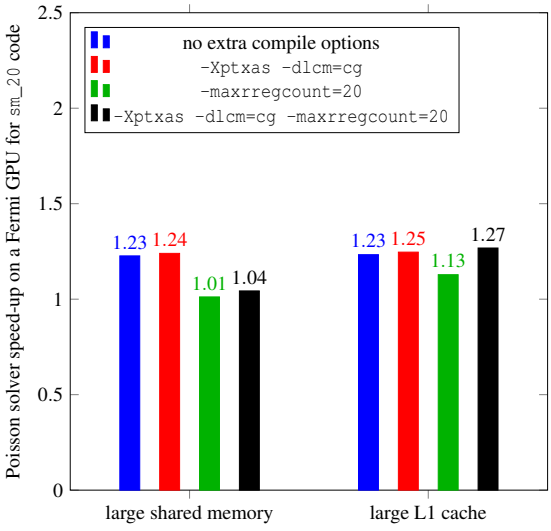

Figure 4: The pressure Poisson solver can be accelerated by switching from code compilation for compute architecture 1.3 to the feature-enriched 2.0 architecture on a Fermi GPU. Some compiler flags allow further optimizations.

better performance for higher grid resolutions. This is a typical behavior for GPUs since the cost-intensive global memory data requests can be better overlapped for computations with higher work loads per GPU. However, the performance increase has to stagnate for some problem size, if the maximum floating point performance or the maximum memory bandwidth limit is hit and interleaving of memory fetches is no longer possible. This stagnation can be clearly seen for the slower GT200 GPU but is less pronounced for the Fermi GPU.

We compile the code for the M1060 / GT200 GPUs with the additional compiler flag `-maxrregcount=20` which limits the used number of registers per GPU kernel to 20. Further experiments showed that this leads to the best performance for this architecture for our specific code. We obtain the following result (cf. Figure 3): By investing the same amount of money, we get more than twice the performance on the older GPU in contrast to one CPU. These results completely meet our expectations. Note again that we compare here one GT200 GPU with *six* CPU cores.

Now, we focus on the new Fermi-type of GPU. It comes with configurable L1 and L2 caches and shared memory. Thus, there are several parameters which influence the performance behavior of the resulting code. We undertake an in-depth performance optimization study for the Poisson solver to find the best parameter configuration for the full code. This is reasonable because this part of the code requires 80 percent of the overall computation costs and also has global reduction operations which are optimized for shared memory. The comparison base for the Fermi GPU optimization is a binary, which is compiled for *compute architecture* 1.3 (i.e. for GT200 GPUs) but without limits to the register count. We then compile the code including Fermi-specific features (i.e. for compute architecture 2.0, `sm_20`). There, we can choose between a large shared memory of 48 KB and 16 KB of L1 cache or a large L1 cache of 48 KB and a small shared memory of 16 KB per symmetric multiprocessor. Furthermore, we try to improve the performance by disabling L1 caches for global memory accesses by the compiler flag `-Xptxas -dlcm=cg`. Finally, we also experiment with a limit of 20 registers per kernel just for the Poisson solver.

The overall results of our study are given in Figure 4. The presented speed-ups are relative to the start configuration with compute architecture 1.3 and are based on accumulated run times for the Poisson solver during the first 20 time steps. The performance of the solver is almost equal to the start configuration if we maximize the available shared memory and limit the register count. However, without the limitation on the reg-

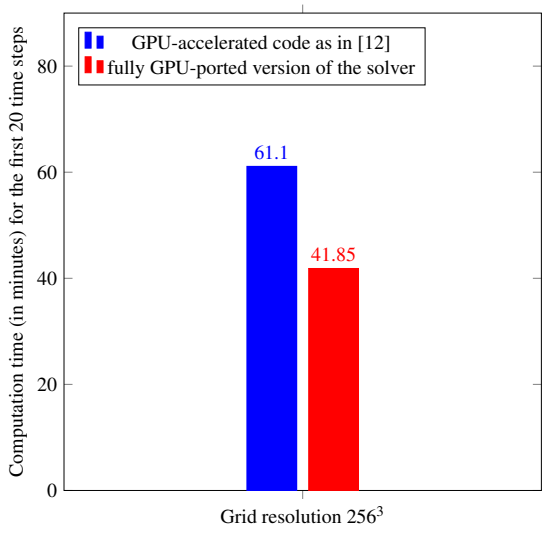

Figure 5: By fully porting the fluid solver to the GPU we get more than a 30 percent improvement in runtime.

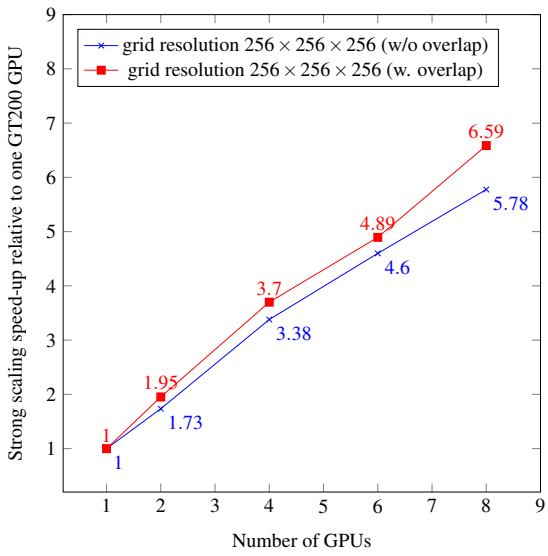

Figure 6: Strong scaling (i.e. speed-up) results on the GT200-based GPU cluster with and without overlapping of computation and communication.

ister count, we get an increase of more than 20 percent in performance for exactly the same code on the same GPU. This is due to the fact that Fermi GPUs now automatically spill registers to the fast L1 cache, while GT200 GPUs use the slow DRAM, cf. [25]. On the other hand, applying bigger L1 caches to our solver kernels gives better results for all configurations. The best results can be seen, if we disable the L1 caches for global memory accesses, limit the register count and apply bigger L1 caches. This effectively means that (instead of spilling the registers automatically to the L1 cache) we manually move a lot of registers to the *local memory* and maximize the L1 cache size for local memory accesses since we do not use it for the global memory. In fact, most of the registers are then handled by the cache. Exactly this parameter configuration is taken for the Poisson solver in all the remaining Fermi speed-up comparisons. To be concise, we do not describe further performed benchmarks for other fluid solver parts which show that e.g. avoiding the register count limitation leads to the best performance for these other kernels. In general, we use different configurations for different portions of the code to get the best results.

We now have a look at the performance per Dollar analysis for the Fermi GPU. The results are given in Figure 3. Here, we should note that we compare this GPU with *12* CPU cores. We see an impressive speed-up factor of three for the GPU computations with ECC protection and of about 3.4 for non-ECC calculations. These results even exceed our expectations if we compare the pure memory bandwidth of the GPU and the CPUs. Obviously, these good results are due to the caches of the Fermi chip which lead to a performance plus of more than 20 percent over the standard memory bandwidth. Remember here that a speed-up of 3.4 in this comparison corresponds to more than a 40 fold speed-up if we compare the GF100 GPU to just one CPU core.

Another observation is that the new Fermi architecture almost gives us a threefold speed-up in contrast to the older GT200 GPU for our double-precision code. We can see this in Figure 3 if we extrapolate the GF100 GPU speed-up over 12 CPU cores to a speed-up over 6 cores and compare these numbers with the GT200 GPU performance results. The better Fermi performance is in our case the combined result of the improved double-precision performance, a higher memory bandwidth and caching effects.

4.5. Performance improvements over the accelerated code

In addition to the previous single-GPU benchmarks, we are also interested in the actual performance gain we achieve by performing a full GPU port of our fluid solver. The first publication by the authors to integrate GPUs in NaSt3DGPF [12] showed a pure acceleration approach. In that case, only the pressure Poisson solver and the reinitialization of the level set function were ported to the GPU. All other computations were done on the CPU which required a high data exchange load between CPU and GPU memory. Now, as stated in Section 3.1, we finally avoid a large number of data exchanges since all computations are ported to the GPU. It is however not a priori clear that this full GPU port really gives us a high performance improvement, since the Poisson solver already takes about 80 percent of the overall CPU computation time. We therefore perform a benchmark which compares the run times of the GPU code used in [12] with our current implementation.

Figure 5 displays the results of this comparison. We can see a clear improvement in run time for the full GPU port. In fact, we get more than a 30 percent performance gain. In our opinion, such an impact in run time clearly shows the necessity of porting full scientific compute codes to GPUs instead of relying on just the pure acceleration approach presented in the earlier days of GPU computing.

4.6. Multi-GPU performance

The following multi-GPU performance study shows results of our local eight-GPU cluster with GT200 GPUs and results

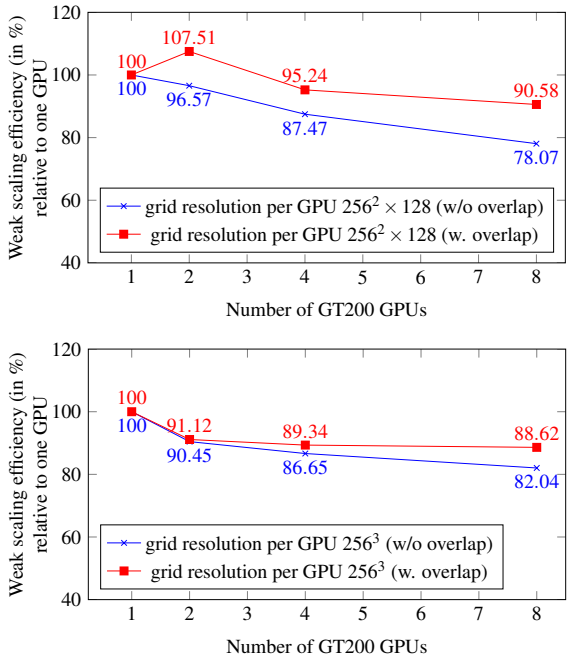

Figure 7: Weak scaling (i.e. scale-up) efficiency results on the GT200-based GPU cluster with and without overlapping strategy for two different per-GPU problem sizes.

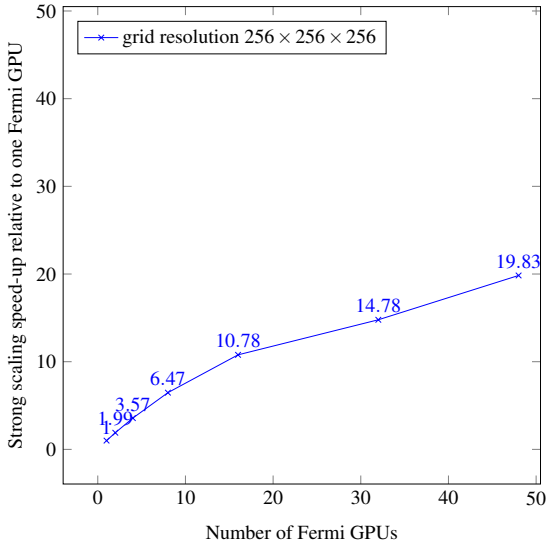

Figure 8: Strong scaling (i.e. speed-up) results on the Fermi-based GPU cluster with overlapping of computation and communication.

of the larger GPU-cluster with Fermi GPUs. We start with the smaller system. There, we always keep the number of GPUs used per workstation equal. Thus, when computing on e.g. 4 GPUs we use two GPUs of the first and two of the second workstation. Note that the PCI Express bus bandwidth of the workstations limits our overall results. Here, we use mainboards with one chipset and up to 48 PCI Express 2.0 lanes.

Figure 6 shows the strong scaling / speed-up results for our multi-GPU tests with and without the technique of overlapping computation and communication, cf. Section 3.4. It gets clear that our good scaling is supported by this overlapping strategy, which we currently use in the Poisson solver. By applying it, we see an almost linear scaling with a profound 6.6 fold speed-up for a simulation with a 256^3 grid size on eight GPUs. We loose only about 18 percent of performance on the maximum number of GPUs by overlapping computation and communication, which is a quite good result.

In Figure 7, we present weak scaling efficiency results for our multi-GPU solver again with and without the overlapping strategy. We use a problem size which is fixed per GPU and increase the number of GPUs. This is done for a problem size of $256 \times 256 \times 128$ and 256^3 per GPU. Thus, our largest test case is a very huge 512^3 grid on just eight GPUs. The weak scaling efficiency values are the absolute run time on one GPU divided by the absolute run times on several GPUs multiplied by 100 percent. For both problem sizes, we see an almost excellent weak scaling behavior which is also based on the described overlapping strategy. The smaller GPU work load for the per-GPU problem size of $256 \times 256 \times 128$ leads to a larger performance difference between non-overlapping and overlapping computations. In the overlapping case, the bigger problem shows only a small increase in execution time when data is transferred over the network for the first time (i.e. for the two-GPU case) while the treatment of the smaller problem even gets faster in this case due to a favorable array alignment on two GPUs. Noting that the larger problem size even hits the available memory limits of the host systems in the 8 GPU case, we only see a very low performance loss on the full eight GPU system for both problem sizes. Overall we loose less then 12 percent from the perfect weak scaling of 100 percent if we overlap computation and communication. This is a nearly optimal result for a multi-GPU scale-up.

Next, we focus on the results measured on the larger Fermi-based 48-GPU cluster of the RWTH Aachen University. Figure 8 shows strong scaling results for a problem size of $256 \times 256 \times 256$ grid points. The scaling is similar to the GT200 cluster for up to eight GPUs. For the largest GPU count of 48 GPUs, we still observe a further scaling with more than 40 percent parallel speed-up efficiency. This is a quite good result, noting that GPUs are less efficient for smaller problem sizes (cf. Figure 3).

Weak scaling results are presented in Figure 9. We see an almost perfect weak scaling going up to 48 GPUs. There is less than a 9 percent loss in parallel performance for all GPU counts. Furthermore, these numbers are better than the results on our small GT200 cluster, since the Fermi-based cluster has a favorable PCI Express and InfiniBand bandwidth per GPU. The slightly fluctuating results can be explained by a varying cluster work load and different array alignments. Again, this is an almost optimal result for a multi-GPU scale-up on a larger and faster GPU system.

4.7. Energy consumption

We finally discuss the performance per Watt results for our code on the different in-house hardware platforms. To do that, we run the bubble simulation for a grid resolution of $256 \times 256 \times 128$ on the full twelve-core CPU system, the full

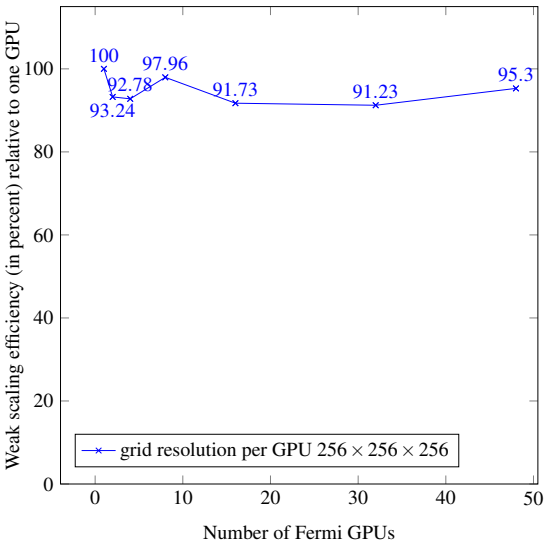

Figure 9: Weak scaling (i.e. scale-up) efficiency results on the Fermi-based GPU cluster using the overlapping strategy.

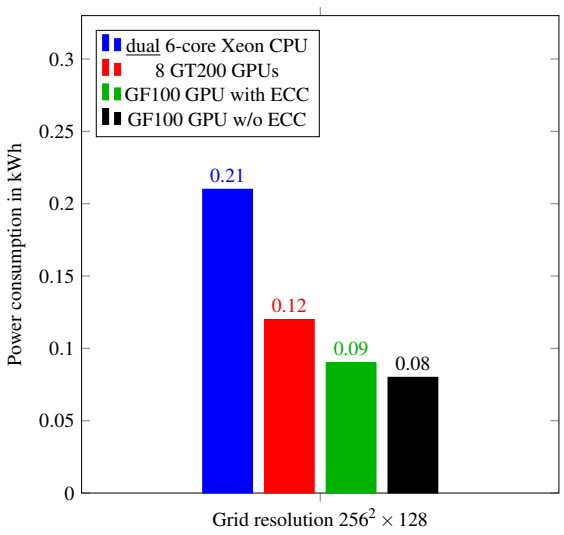

Figure 10: The measured power consumption figures show that the Fermi GPU with ECC is about a factor of 2.3 more power-efficient compared to standard CPUs.

eight-GPU cluster and on the Fermi GPU with and without ECC. During the simulation, we attach a power measuring tool to each of the machines and note the power consumption after each of the systems performed the calculation of the first 20 time steps including data field initialization. The resulting numbers are presented in Figure 10. The maximum power consumption is required with the CPU system. In contrast, the results for the GPU cluster are already significantly better. The numbers for the Fermi card is quite impressive. It is by a factor of 2.3 more power efficient for calculations with ECC and even a bit more efficient for non-ECC calculations. These results are a clear indicator for the superiority of GPU computations. This may be important for large computing centers which are looking for energy efficient systems.

5. Discussion and Conclusions

We presented our results on a fully multi-GPU-based parallel double-precision fluid solver for the two-phase incompressible Navier-Stokes equations which uses the level-set method. Other GPU-oriented publications for two-phase incompressible flows [10, 11] are not able to achieve our performance results, because they only support *single-GPU* computations and partly do not fully run on the GPU. In contrast, we were able to discuss the first published results for a *fully multi-GPU-based* solver of this kind, at least to our knowledge. Furthermore, our solver is designed for engineering applications and can now be applied for our various projects and cooperations. For example, we can easily simulate the crown formation of a liquid after a droplet impact, cf. Figure 11. Note that we can also use our solver for simulations with arbitrarily complex geometries.

In this paper, we could give very promising benchmarking results which included a fully realistic three fold speed-up in comparison to equally priced CPU hardware and a more than doubled power efficiency for a real world application. Moreover, we did a standard multi-GPU parallelization benchmark on two GPU clusters, which showed a profound strong and an even better weak scaling. Overall, these results emphasize that our two-phase Navier-Stokes solver is capable to get good performance on parallel multi-GPU clusters.

Acknowledgements

This work was supported in parts by the Sonderforschungsbereich 611 *Singular phenomena and scaling in mathematical models* funded by the *Deutsche Forschungsgemeinschaft*. We also like to thank the *Ressourcenverbund NRW* and the *RWTH Aachen University* for giving us access to the GPU cluster of the *Center for Computing and Communication*. Furthermore, we acknowledge *Nvidia Corporation* for the provided Fermi card.

References

[1] C. Feichtinger, J. Habich, H. Köstler, G. Hager, U. Rüde, G. Wellein, A flexible patch-based lattice Boltzmann parallelization approach for heterogeneous GPU-CPU clusters, Parallel Computing, In Press, Corrected Proof (2011) .

[2] X. Wang, T. Aoki, Multi-GPU performance of incompressible flow computation by lattice Boltzmann method on GPU cluster, Parallel Computing, In Press, Corrected Proof (2011) .

[3] A. Corrigan, R. Löhner, Porting of FEFLO to multi-GPU clusters, Proceedings of the 49th AIAA Aerospace Sciences Meeting, Orlando, Florida, USA (2011) .

[4] T. Brandvik, G. Pullan, An accelerated 3d Navier–Stokes solver for flows in turbomachines, Journal of Turbomachinery 133 (2) (2011) .

[5] T. Shimokawabe, T. Aoki, C. Muroi, J. Ishida, K. Kawano, T. Endo, A. Nukada, N. Maruyama, S. Matsuoka, An 80-fold speedup, 15.0 tflops full GPU acceleration of non-hydrostatic weather model ASUCA production code, Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, Washington, DC, USA (2010) 1–11.

[6] J. Cohen, M. Molemaker, A Fast Double Precision CFD Code using CUDA, in: Proceedings of Parallel CFD 2009, Moffett Field, California, USA, 2009.

[7] D. A. Jacobsen, J. C. Thibault, I. Senocak, An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters, Proceedings of the 48th AIAA Aerospace Sciences Meeting and Exhibit, Orlando, Florida, USA (2010) .

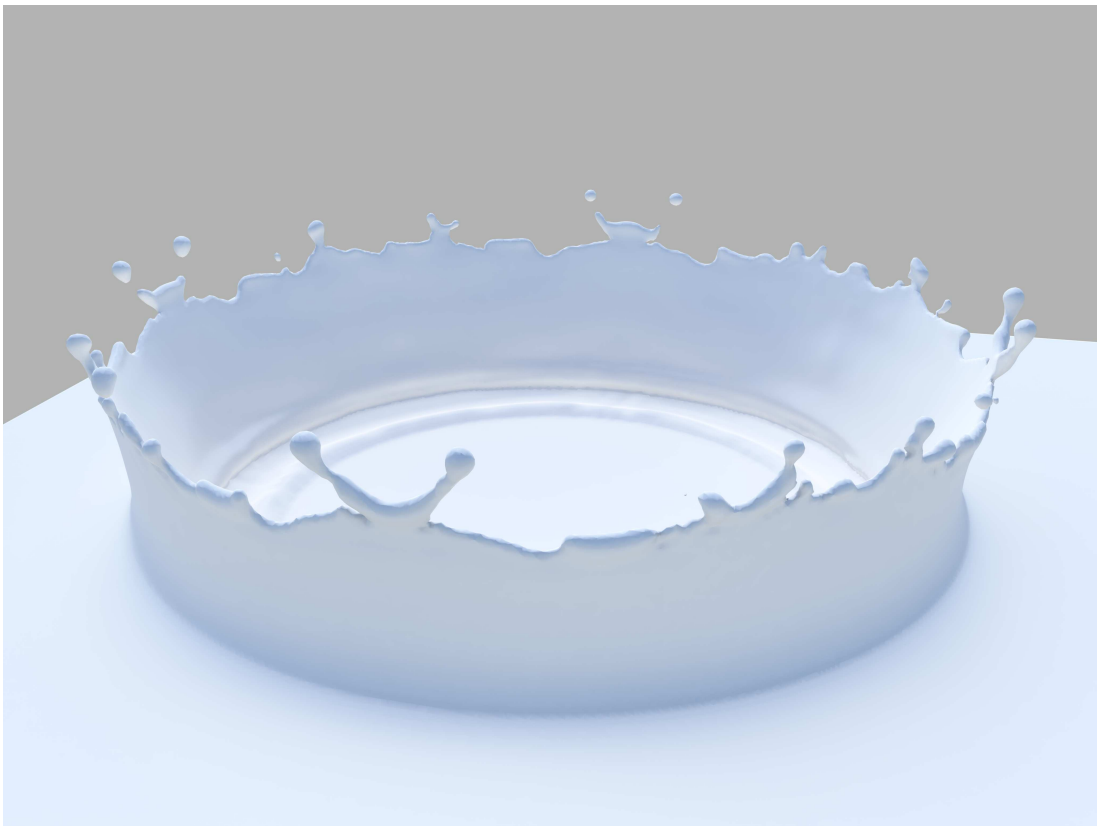

Figure 11: The simulation of the crown formation after a droplet impact in a liquid is a real-world application which was computed with our multi-GPU code.

[8] D. A. Jacobsen, I. Senocak, A full-depth amalgamated parallel 3d geometric multigrid solver for GPU clusters, Proceedings of the 49th AIAA Aerospace Sciences Meeting, Orlando, Florida, USA (2011) .

[9] D. Göddeke, S. H. Buijssen, H. Wobker, S. Turek, GPU acceleration of an unmodified parallel finite element Navier-Stokes solver, in: W. W. Smari, J. P. McIntire (Eds.), High Performance Computing & Simulation 2009, 2009, pp. 12–21.

[10] J. Kelly, GPU-accelerated simulation of two-phase incompressible fluid flow using a level-set method for interface capturing, ASME Conference Proceedings 2009 (43826) (2009) 2221–2228.

[11] S.-H. Kuo, P.-H. Chiu, R.-K. Lin, Y.-T. Lin, GPU implementation for solving incompressible two-phase flows, World Academy of Science, Engineering and Technology (2011) 878–886.

[12] M. Griebel, P. Zaspel, A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations, Computer Science - Research and Development 25 (1–2) (2010) 65–73.

[13] T. Dornseifer, M. Griebel, T. Neunhoeffer, Numerical Simulation in Fluid Dynamics, a Practical Introduction, SIAM, Philadelphia, 1998.

[14] J. Strybny, C. Thorenz, R. Croce, M. Engel, A parallel 3d free surface Navier-Stokes solver for high performance computing at the german waterways administration, in: The 7th Int. Conf. on Hydroscience and Engineering (ICHE-2006), Philadelphia, USA, 2006.

[15] B. Verleye, R. Croce, M. Griebel, M. Klitz, S. Lomov, G. Morren, H. Sol, I. Verpoest, D. Roose, Permeability of textile reinforcements: Simulation, influence of shear, validation, Composites Science and Technology 68 (13) (2008) 2804–2810.

[16] R. Croce, M. Griebel, M. A. Schweitzer, Numerical simulation of bubble and droplet-deformation by a level set approach with surface tension in three dimensions, International Journal for Numerical Methods in Fluids 62 (9) (2009) 963–993.

[17] S. Osher, J. A. Sethian, Fronts propagating with curvature dependent speed: Algorithms based on Hamilton-Jacobi formulations, Journal of Computational Physics 79 (1988) 12–49.

[18] M. Sussman, P. Smereka, S. Osher, A level set approach for computing solutions to incompressible two-phase flow, J. Comput. Phys. 114 (1994) 146–159.

[19] J. U. Brackbill, D. B. Kothe, C. Zemach, A continuum method for modeling surface tension, J. Comput. Phys. 100 (2) (1992) 335–354.

[20] A. J. Chorin, Numerical solution of the Navier-Stokes equations, Mathematics of Computation 22 (104) (1968) 745–762.

[21] P. Micikevicius, 3D Finite Difference Computation on GPUs using CUDA, in: GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, ACM, New York, NY, USA, 2009, pp. 79–84.

[22] M. Griebel, P. Zaspel, Photorealistic visualization and fluid animation - coupling of Maya with a two-phase Navier-Stokes fluid solver, Computing and Visualization in Science, submitted 2011.

[23] J. K. Walters, J. F. Davidson, The initial motion of a gas bubble formed in an inviscid liquid - Part 2. The three-dimensional bubble and the toroidal bubble, J. Fluid. Mech. 17 (1963) 312–336.

[24] Web page of the Top500 list of supercomputers, `http://www.top500.org` (last access: 08/09/2011).

[25] NVIDIA Corporation, NVIDIA's next generation CUDA compute architecture: Fermi, Whitepaper (2009).