Computer Science - Research and Development manuscript No.
(will be inserted by the editor)

A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations

Michael Griebel · **Peter Zaspel**

Received: date / Accepted: date

Abstract The use of graphics hardware for general purpose computations allows scientists to enormously speed up their numerical codes. We presently investigate the impact of this technology on our computational fluid dynamics solver for the three-dimensional two-phase incompressible Navier-Stokes equations, which is based on the level set technique and applies Chorin's projection approach. To our knowledge, this is the first time, that a two-phase solver for the Navier-Stokes equations profits from the computation power of modern graphics hardware. As part of our project, a Jacobi preconditioned conjugate gradient solver for the pressure Poisson equation and the reinitialization of the level set function of our CPU based code were ported to the graphics processing unit (GPU). They are implemented in double precision and parallelized by the Message Passing Interface (MPI). We obtain speedups of 16.2 and 8.6 for the Poisson solver and the reinitialization on one GPU in contrast to a single CPU. Our implementation scales close to perfect on multiple GPUs of a distributed memory cluster. This results in excellent speedups of 115.8 and 53.7 on eight GPUs of our cluster. Furthermore our whole multi-GPU accelerated solver achieves an impressive speedup of 69.6 on eight GPUs/CPUs.

Keywords Computational fluid dynamics · Graphics hardware · Navier-Stokes equations · Multi-GPU · Two-phase flows

M. Griebel
Institute for Numerical Simulation - University of Bonn
Wegelerstr. 6, 53115 Bonn, Germany
E-mail: griebel@ins.uni-bonn.de
Tel.: +49-228-733437, Fax: +49-228-737527

P. Zaspel
Institute for Numerical Simulation - University of Bonn
Wegelerstr. 6, 53115 Bonn, Germany
E-mail: zaspel@ins.uni-bonn.de
Tel.: +49-228-732748, Fax: +49-228-737527

1 Introduction

The numerical solution of the three-dimensional incompressible Navier-Stokes equations is a computationally expensive task. So far, high resolution fluid simulations are performed on large parallel clusters only. General purpose computations on graphics cards promise to perform the same calculations on small cost-effective workstations with low energy requirements.

Current graphics processors, i.e. GPUs, can be used as highly parallel computation units. Computer graphics and image processing were among the first fields that utilized computations on graphics hardware [10], [11]. For now, GPU parallelizations follow the single instruction multiple data (SIMD) idea. Depending on the hardware, tens or even hundreds of *streaming processor cores* compute fully parallel on large data sets. Since the first generation of programmable GPUs was limited to single precision floating point arithmetics, they could not be effectively used in numerical fields with high accuracy requirements, such as computational fluid dynamics for engineering applications. With the introduction of the latest graphics architectures, e.g. the NVIDIA Tesla platform, double precision calculations are now available on the GPU. As a result, this type of hardware becomes increasingly popular for scientific applications.

So far, there are still only few applications of GPUs to solve the full[1] non-stationary incompressible three-dimensional Navier-Stokes equations with an Eulerian grid based approach. Krüger [14] was one of the first to publish results in this field with the target of realtime applications for fluid dynamics. His solver uses Chorin's projection approach [2]

[1] Note, that we call the standard Navier-Stokes equations *full*, to be able to distinguish them from the *inviscid* Navier-Stokes equations frequently found in computer graphics.

on a staggered grid, finite differences with forward and central differencing, velocity advection as proposed by Stam [18], a conjugate gradient (CG) solver for the pressure Poisson equation and vorticity confinement [19] to reduce the numerical diffusion introduced by Stam's method. Due to hardware, this solver was limited to single precision.

Furthermore, Thibault and Senocak [21] implemented the first multi-GPU solver for the full incompressible Navier-Stokes equations. Instead of Stam's advection approach they use a first order explicit Euler scheme. The pressure Poisson equation is solved by a Jacobi iterative solver. To compute on multiple GPUs, they use a shared-memory parallelization by Posix threads and a standard domain decomposition approach. Due to hardware limitations, they also compute in single precision. This way, they obtain a speedup factor of 33 on one GPU compared to a single CPU and a speedup factor of 100 on four GPUs.

Cohen and Molemaker [3] implemented a *double* precision solver for the Navier-Stokes equations. They included temperature into their model via the Boussinesq approximation [6]. The discretization employs a second order finite volume approach on a staggered regular grid, the pressure projection method and a second order Adams-Bashforth time integration. A multigrid solver handles the Poisson equation. This way, a maximum speedup factor of 8.5 is obtained on the latest available graphics hardware (i.e. an NVIDIA C1060) compared to an eight-core multithreaded Fortran fluid solver.

A fundamental effort to implement three-dimensional finite difference methods on GPUs has been made by Micikevicius [15]. He introduced base patterns for the fast computation of high order finite difference stencils. Additionally, he presented a scalable and fast multi-node / multi-GPU parallelization using MPI.

The currently available literature on the solution of the Navier-Stokes equations on graphics hardware already shows quite promising performance gains. However, the applied numerical methods are often not designed for real-world engineering applications. Engineers and researchers need a parallel high order double precision fluid solver that scales on large clusters. Our in-house CPU Navier-Stokes solver NaSt3DGPF [5], [4] meets these requirements. It solves the full two-phase non-stationary incompressible Navier-Stokes equations in three dimensions. Similar to [21], we apply the pressure projection approach of Chorin and discretize the equations on a staggered grid with finite differences. A fifth order Weighted Essentially Non-Oscillatory (WENO) [12] space discretization is used for the convective terms. Time integration is discretized either by a second order Adams-Bashforth or a third order Runge-Kutta method. The Poisson equation is solved with a Jacobi-preconditioned conjugate gradient iterative method or with an algebraic multigrid method [7]. Arbitrary geometries as well as a variety of boundary conditions are handled. A large-eddy turbulence model, temperature and passive transport of species concentration is implemented, as well. An MPI-parallelization based on the domain decomposition method of Schwarz [4] allows our solver to scale on large clusters.

There are a lot of applications where the solution of the Navier-Stokes equations for one phase does not fulfill the requirements of a realistic simulation model. Studies of droplet behavior [5], gas bubble dynamics or the physics of rivers crossing a dam construction [20] make the simultaneous simulation of two fluid phases (e.g. air and water) necessary. Our solver NaSt3DGPF uses a level set function [17] for the distinction of the two phases. Additionally, surface tension effects of the free surface between the two phases are included by the Continuum Surface Force (CSF) scheme [1].

We presently port parts of our CPU fluid solver to the GPU. After investigating the performance bottlenecks of our code, we first implemented the Jacobi-preconditioned conjugate gradient solver for the Poisson equation and the reinitialization process for the level set function on the GPU. The level set reinitialization is performed by solving a partial differential equation of Hamilton-Jacobi type in artificial time. For its spacial discretization, we implemented the fifth order WENO method, the time discretization is performed by a Runge-Kutta method of third order. The GPU code is integrated with the domain decomposition MPI parallelization framework. We thus obtain a substantially multi-GPU accelerated, fully parallel and scalable fluid solver.

The remainder of this article is organized as follows. In Section 2, we introduce our model for the solution of the two-phase Navier-Stokes equations as well as parallelization techniques. Section 3 outlines the necessary implementation details of our GPU code. In Section 4, we measure and analyze the performance of our implementation. Section 5 summarizes the results and gives a short outlook.

The contribution of this publication is as follows:

- We show the first application of multi-node / multi-GPU computations in a high order engineering targeted solver for the full[2] non-stationary incompressible Navier-Stokes equations.
- For the first time, a *two-phase* solver for the incompressible Navier-Stokes equations profits from the power of current graphics hardware.
- We present a successful example for the integration of GPU computations into a legacy CPU code.

[2] See footnote 1.

2 Two-Phase Incompressible Navier-Stokes Equations

The simulation of incompressible two-phase flows utilizes the level set function and is based on the two-phase Navier-Stokes equations with surface tension [5]. They consist of the momentum equation

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

and the continuity equation

$$\nabla\cdot\mathbf{u} = 0\,.$$

The tensor S is given by $\mathsf{S} := \nabla\mathbf{u} + \{\nabla\mathbf{u}\}^T$ and the material derivative $\frac{D\mathbf{u}}{Dt}$ can be expressed by $\frac{D\mathbf{u}}{Dt} := \partial_t\mathbf{u} + (\mathbf{u}\cdot\nabla)\mathbf{u}$. The fluid velocity $\mathbf{u}$ and pressure p are space- and time-dependent with time $t\in[0,t_{end}]$. The level set function ϕ is a signed distance function with

$$\phi(\mathbf{x},t)\begin{cases} <0 & \text{if } \mathbf{x}\in\Omega_1 \\ =0 & \text{if } \mathbf{x}\in\Gamma_f \\ >0 & \text{if } \mathbf{x}\in\Omega_2 \end{cases} \qquad \text{and} \qquad |\nabla\phi| = 1$$

and defines the free surface $\Gamma_f(t) = \{\mathbf{x}:\phi(\mathbf{x},t)=0\}$ between the two fluid phases. We set the density ρ and the dynamic viscosity μ on the two disjoint fluid phase domains Ω_1 and Ω_2 as

$$\begin{aligned}\rho(\phi) &:= \rho_2 + (\rho_1-\rho_2)H(\phi) \\ \mu(\phi) &:= \mu_2 + (\mu_1-\mu_2)H(\phi)\end{aligned}, \qquad H(\phi) := \begin{cases} 0 & \text{if } \phi<0 \\ \frac{1}{2} & \text{if } \phi=0 \\ 1 & \text{if } \phi>0 \end{cases}.$$

The surface tension coefficient is denoted by σ, volume forces (such as gravity) are given by $\mathbf{g}$, the curvature of the free surface Γ_f is called κ and δ is the Dirac δ-functional.

In our CPU code [5], we apply Chorin's projection method and the level set method to solve the two-phase unsteady Navier-Stokes equations. This results in Algorithm 1 which describes the time-discrete computation of the velocity, the pressure and the level set function for a new time step t^{n+1} with time step size δt in our simulation.[3]

After the level set transport in step 2 of Algorithm 1, the distance property $|\nabla\phi| = 1$ may be destroyed. By performing the reinitialization in step 3, this property can be recovered. We solve the reinitialization equation by a time stepping method in artificial time τ. For $\tau = 0$ the distance field d is initialized with the transported level set function ϕ^*. After convergence, the distance field d, becomes the level set function ϕ^{n+1} of the new time step.

Our CPU parallelization follows the domain decomposition approach of Schwarz. We thus decompose our uniform grid into subblocks of similar size. Each parallel process stores the simulation data of its own subblock and computes on this data. For stencil operations, we add several layers

[3] Here, for ease of notation, we describe a forward Euler time integration scheme. The application of higher order schemes is straightforward.

Algorithm 1 (Computation of a new time step)

1. compute intermediate velocity field $\mathbf{u}^*$

$$\frac{\mathbf{u}^*-\mathbf{u}^n}{\delta t} = -(\mathbf{u}^n\cdot\nabla)\mathbf{u}^n + \mathbf{g} + \frac{1}{\rho(\phi^n)}\nabla\cdot(\mu(\phi^n)\mathsf{S}^n) - \frac{1}{\rho(\phi^n)}\sigma\kappa(\phi^n)\delta(\phi^n)\nabla\phi^n$$

2. transport level set function

$$\phi^* = \phi^n + \delta t\,(\mathbf{u}^n\cdot\nabla\phi^n)$$

3. reinitialize level set function by solving

$$\partial_\tau d + sign(\phi^*)(|\nabla d|-1) = 0$$

4. solve the pressure Poisson equation

$$\nabla\cdot\left(\frac{\delta t}{\rho(\phi^{n+1})}\nabla p^{n+1}\right) = \nabla\cdot\mathbf{u}^*$$

5. apply velocity correction

$$\mathbf{u}^{n+1} = \mathbf{u}^* - \frac{\delta t}{\rho(\phi^{n+1})}\nabla p^{n+1}$$

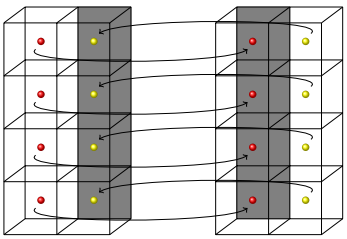

Fig. 1 Data exchange between ghost cells of two parallel processes

of *ghost cells* at the boundaries between the parallel blocks. The layer count depends on the order of the stencil. Ghost cells contain the necessary field values of the neighbor process. This data is exchanged when necessary (see Fig. 1).

3 GPU Implementation Details

The GPU implementation of the Poisson solver and the reinitialization process were performed using the Compute Unified Device Architecture (CUDA) framework [16] for parallel programming, which has been introduced by the graphics hardware company NVIDIA. CUDA defines a parallel hardware architecture as well as an extension to the C programming language for SIMD parallelizations. As part of the language extension, the programmer is able to implement C functions called *kernels*, which are executed on the GPU. In addition, there are user methods for hardware initialization, memory handling and thread management. For reasons of brevity, we do not describe the the basic concepts of CUDA and its parallel programming framework. We re-

fer the reader to [3], [21] or [8] for short introductions to CUDA.

Poisson Solver Our implementation of the preconditioned conjugate gradient (PCG) solver utilizes double precision calculations, because our engineering applications impose very high accuracy requirements. Since we discretize the left-hand side of the pressure Poisson equation with a standard seven-point stencil, the matrix of the discretized linear system is sparse. By choosing the appropriate variable ordering, we get a matrix with seven non-zero diagonals. We thus have to implement a sparse PCG solver.

As initialization process of the PCG solver, we precalculate the non-zero coefficients of the matrix and the right-hand side of the system on the CPU and transfer them to the GPU. This approach reduces the implementation effort and is still efficient, because the calculation time for the coefficients is dominated by the time needed for the PCG iterative solution.

The three most important building blocks of a sparse PCG solver are a fast sparse matrix-vector product, a fast scalar product and the preconditioner. We have implemented GPU kernels for all of them.

The diagonal structure of the sparse matrix allows us to implement a very efficient matrix-vector product, which is based on the SIMD parallel summation of the seven stencil points. Note, that we do not assemble the full matrix. To reduce the usage of GPU registers, we save the memory access addresses for the pressure field in shared memory. The register reduction leads to a higher occupancy of the GPU's multiprocessors which results, in this case, in an optimized data throughput for our matrix-vector product kernel.

To avoid the implementation of an additional kernel for the enforcement of the Poisson equation's Neumann boundary conditions, we integrate the boundary handling directly into the matrix-vector product. Instead of a summation of all of the seven stencil values, we omit those values which would be normally removed by a separate boundary treatment. To find these points, we calculate a flag field of boundary positions on the CPU and transfer it to the GPU. It is then passed to the matrix-vector product kernel and used to decide whether to use a stencil value. Extensive profilings have shown that the necessary additional conditional statements in the kernel have only slight effect on the performance.

For the implementation of an efficient scalar product, we follow the lines of [9], and implement a highly optimized, shared memory based scalar product kernel.

The Jacobi preconditioner is perfectly suited for the SIMD parallelism of the GPU. The pointwise product can be easily implemented in a CUDA kernel and shows excellent performance.

Level Set Reinitialization The reinitialization process of the level set function was implemented with double precision GPU kernels for the calculation of the regularized *sign* function, boundary conditions, the gradient of the distance field d and the Runge-Kutta time integration. We focus here on the details of the WENO based ∇d computation as this turned out to be most challenging when ported to the GPU.

The partial derivatives d_x, d_y and d_z of the gradient are computed with an upwind method based on a fifth order WENO scheme. We will briefly sketch the idea of the WENO scheme. For further reading, we suggest [12] and [13].

WENO is a finite difference discretization scheme for derivatives, which is stable even in the case of the existence of large gradients or shocks in the discretized field. The main concept of the scheme is to weight the grid points of the stencil according to their smoothness. The smoother the derived function in a grid point, the higher its weight. Thereby, discontinuities or large gradients are ignored. A weighting can be achieved by so-called *smoothness indicators* which are based on the first order forward differences of each neighboring stencil grid point. The WENO scheme computation is summed up in Algorithm 2.

Algorithm 2 (Sketch of WENO computation)

1. compute first order forward differences between involved grid points
2. compute smoothness indicators
3. build weighted sum of forward differences according to smoothness indicators

$\Rightarrow$ high order approximation of derivative

A first straight-forward GPU implementation of the gradient in *one* computation kernel turned out to be very slow, as the WENO scheme requires a lot of computations per grid point and many variables. Consequently the register usage increased dramatically, so that we had to limit the register number and force the compiler to put most of the local variables into the slow local memory.

Our fastest gradient computation kernel is now based on the idea to divide the computation into several small kernels. We compute each partial derivative separately. Additionally, the first order forward differences (step 1 of Algorithm 2) are first precomputed in one kernel for the whole level set field and stored in a temporary data field. A second kernel than loads this data to compute the smoothness indicators and the approximation of the partial derivative. By following this approach, we could more than triple our computation performance.

Efficient Multi-GPU Parallelization The multi-GPU parallelization has been fully integrated with the distributed memory MPI parallelization of our CPU code. Thus, each

GPU holds exactly the same decomposed subblocks of the data fields which belong to the respective CPU.

The major challenge of the multi-GPU parallelization is an efficient implementation of the data exchange process. In the case of a parallel CPU implementation, data exchanges involve the data transport from CPU memory to the network interface, the data transmission over the network and a final data copy from the network interface to the CPU memory. A distributed memory multi-GPU parallelization requires additional data transfers from the GPU to the CPU and vice versa. As a matter of fact, memory transfers between the GPU and the CPU are one major performance bottleneck of GPU computations. It is therefore necessary to find a fast way to exchange the ghost cells of the decomposed data blocks between the GPU and the CPU.

A broadly known solution to this problem lies in an efficient data reordering for a fast GPU $\leftrightarrow$ CPU transfer. If each ghost cell slice is copied separately from GPU to CPU and vice versa, this would be a badly coalesced copy operation and would result in a massive performance lost. We therefore copy the ghost cells to one contiguous memory block on the GPU and transfer this block between the GPU and the CPU. The reordering or packing on the GPU is fast, since data transfers on the GPU memory are fast. Additionally, a GPU $\leftrightarrow$ CPU data transfer of a contiguous data block is the fastest available data exchange. A technical realization of the data reordering involves the implementation of GPU kernels for data packing and unpacking and corresponding CPU functions for unpacking and packing.

By using the latest generation of GPUs (e.g. an NVIDIA Tesla S1070), we can often even go further and, similar to the work done in [15], completely hide the total time necessary for GPU $\leftrightarrow$ CPU and network data transfers. This can be achieved by the so-called *concurrent copy and execution* capability: The GPU is now able to perform an asynchronous data transfer while it is calculating on another piece of data. Controlled by *CUDA Streams*, we can also launch CPU methods while the GPU is performing calculations. Consequently we are able to reorder and transfer data while a GPU kernel works on another data package.

Let us exemplify this technique by the sparse matrix-vector product of our PCG solver. In a straight-forward implementation, one would first exchange the ghost cell data and then calculate the whole matrix-vector product. In contrast, by the approach of parallelized computations and communications described above, we can hide the time of data transfers behind the time-consuming matrix-vector product. The main idea is to calculate the matrix-vector product on the *inner* variables of the decomposed field blocks while the ghost cell variables are exchanged. In a last serial step, the matrix-vector product is performed on the ghost cells. This can be done very fast. Details of this approach are given in Algorithm 3.

Note, that the reinitialization process of the level set function could also profit from the application of overlapped data transfers and calculations. However, this is still work in progress.

Algorithm 3 (Concurrent data transfer and calculation)

Stream 1	**Stream 2**
1. pack ghost cell data on GPU	
	2.1 GPU $\rightarrow$ CPU transfer of packed ghost cells
	2.2 unpack ghost cell data on CPU
2. perform matrix-vector product on inner cells	2.3 exchange data over network connection
	2.4 pack received ghost cell data on CPU
	2.3 CPU $\rightarrow$ GPU transfer of packed ghost cells
3. unpack ghost cell data on GPU	
4. perform matrix-vector product on ghost cells	

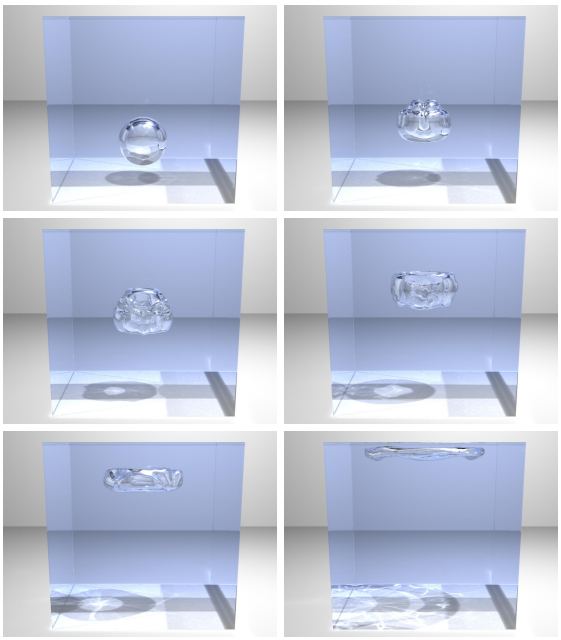

Fig. 2 Photorealistic visualization of the benchmark problem of a large air bubble rising in water (time: $0.0s$ – $0.34s$)

4 Performance Results

We have measured the performance of our implementation on an eight GPU cluster, which is based on two separate workstations. Both workstations use a four core Intel Core i7-920 CPU with 2.66 GHz and are equipped with 12GB DDR3-RAM (1333 MHz). Each node is connected to an NVIDIA Tesla S1070 GPU computing system. The 10 gigabit network interconnect Myri-10G is used for parallel communication. As software setup, we employ the Ubuntu Linux 9.10 operating system as 64 bit version, a GCC 4.2, MPICH-MX 1.2.7 and CUDA 2.3. Our fluid solver was compiled with the optimization flag `-O3`.

The correctness of our GPU implementation was checked against the CPU code. We use identical numerical methods in both codes. Consequently, we can expect equal results up to machine accuracy and up the order of convergence. However, it is a well-known fact, that the parallelized scalar product used in our GPU based PCG solver leads to a slightly different numerical result in contrast to a sequential scalar product, since the different parallel summation order introduces modified rounding errors.

Our error measurements totally agree with our expectations. The PCG solver reduces the residual of the linear system to machine accuracy. For the reinitialization, we also get equal results up to machine accuracy and convergence order.

CPU and GPU performance benchmarks were performed on the same machines to be comparable. Time measurements were made using the `gettimeofday` command and include, in the case of GPU computations, the time necessary for the CPU precalculations and the GPU $\leftrightarrow$ CPU data transfers. We measured *wall clock* times.

Our benchmark problem is the simulation of a large air bubble rising in a box filled with water. Table 1 provides the parameters of the test problem and Figure 2, shows a photorealistic visualization of the simulation result. The measured benchmark times reflect the time consumption of the level set reinitialization and the Poisson solver methods during the first 50 time steps of the simulation. In the case of the PCG solver, we fix the number of iterations to one thousand.

domain size:	$20\,cm \times 20\,cm \times 20\,cm$
liquid phase:	water at 20^oC
gas phase:	air at 20^oC
volume forces:	standard gravity
initial air bubble radius:	$3\,cm$
initial center position of bubble:	$(10\,cm, 6\,cm, 10\,cm)$

Table 1 Parameters of the benchmark problem of a rising air bubble in water

Figure 3 shows the GPU speedup results for the Poisson solver and the level set reinitialization compared to the computation on one CPU core. The grid resolution of the benchmark simulation is 300^3. We get an excellent speedup of 16.2 for the double precision Poisson solver on one GPU. The double precision GPU reinitialization process achieves a major speedup of 8.6. When using our full eight GPU cluster, we are able to achieve impressive speedups of 115.8 for the PCG solver and 53.7 for the reinitialization.

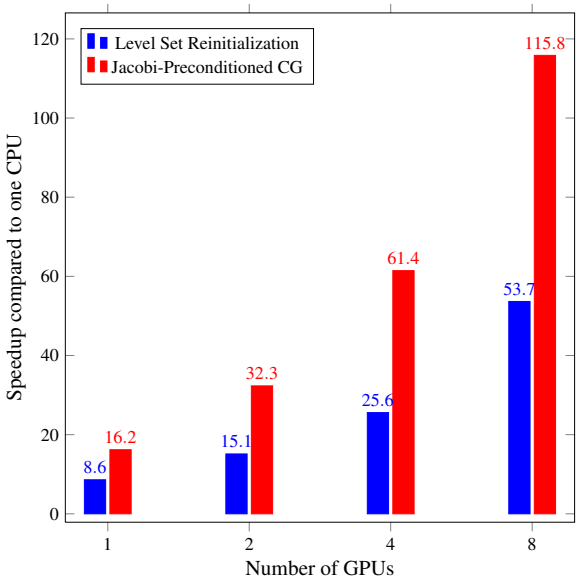

Fig. 3 GPU speedup compared to one CPU core for a grid resolution of 300^3

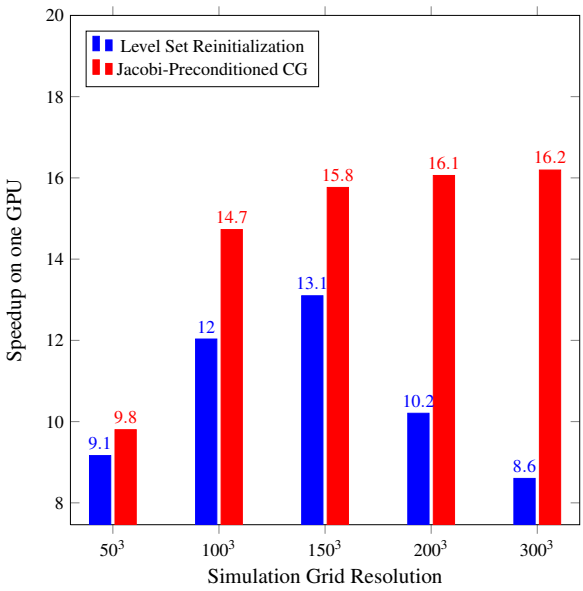

Fig. 4 Speedup of one GPU relative to one CPU for different resolutions

In [21], it is observed that, with growing problem sizes, the per-GPU efficiency increases up to optimality. Figure 4 presents speedup results for growing problem sizes. When it comes to the PCG solver, we can agree with this observation. However, the reinitialization process of the level set function shows a somewhat different behavior: for problem sizes of

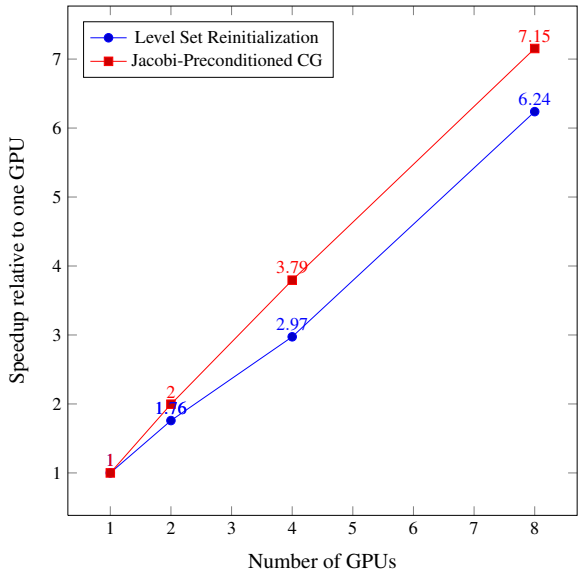

Fig. 5 Multi-GPU speedup relative to one single GPU for a grid resolution of 300^3

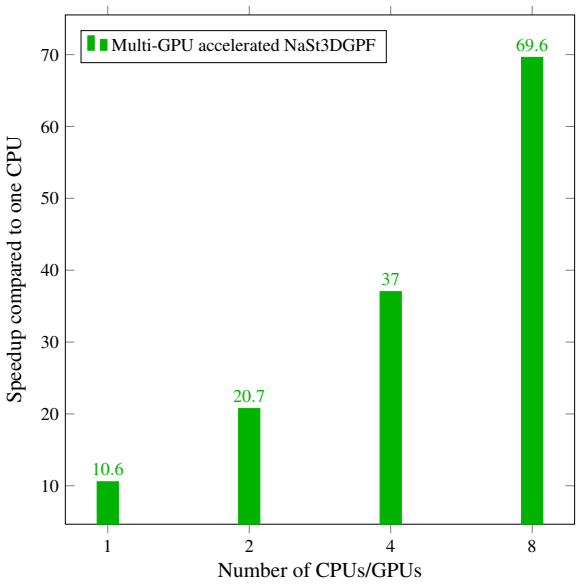

Fig. 6 Overall speedup of our multi-GPU accelerated fluid solver compared to one CPU core for a grid resolution of 300^3

Throughput in GFLOPS			
1 GPU	2 GPUs	4 GPUs	8 GPUs
4.279	8.599	16.452	31.221

Table 2 Throughput measurements for the double precision PCG solver at a resolution of 300^3

50^3, 100^3 and 150^3 the speedup compared to one CPU increases similar to [21], but breaks down for simulation grid sizes of 200^3 and 300^3. We could think of multiple reasons for this kind of behavior. Possible explanations might be the GPU occupancy or uncoalesced memory accesses. A rigorous analysis of this result is subject to further research.

The high quality of our parallel multi-GPU implementation can be seen in Figure 5. The multi-GPU scaling of our PCG solver meets our expectations to be close to perfect on all eight GPUs. This results from the highly optimized data transfer with fully parallel computations and communication. A nearly linear scaling with high parallel efficiency can be observed for the reinitialization process. In this case, the lack of a parallelized computation and communication is visible. The bend in the graph can be explained by the limited performance of the second generation PCI Express 8x slots on the mainboards of our workstations. The more than doubled speedup, which can be seen between four and eight GPUs, agrees with the observations made for the reinitialization process in Figure 4: Since we halve the number of grid points per GPU, we get a better per-GPU efficiency on eight GPUs.

The floating point operation throughput of a numerical algorithm can act as an indicator for implementation efficiency. This is why we calculated it for our PCG solver. After figuring out the required number of double precision floating point operations for a thousand iterations we can calculate the throughput in *floating points per second* (FLOPS) based on our time measurements. Please note again, that we include the time necessary for the CPU precalculations and the CPU $\leftrightarrow$ GPU data transfers into our measurements. The real throughput of our GPU kernels is even higher. Table 2 sums up the performance in double precision gigaFLOPS (GFLOPS). We get a remarkable maximum of 4.3 GFLOPS for one GPU and impressive 31.2 GFLOPS of throughput on eight GPUs.

We finally had a look at the overall speedup of our multi-GPU accelerated Navier-Stokes solver. Figure 6 shows the speedup results of the whole CPU fluid solver when using multi-GPU computations. Again, we compare our results with computation times on one CPU core. Even though we just ported two components of our solver to the GPU we get a major speedup of 10.6 by using one GPU and an excellent speedup of 69.6 on our full eight GPU cluster.

5 Conclusions and Outlook

We have presented the progress of porting our parallel CPU-based in-house solver for the three-dimensional two-phase Navier-Stokes equations to the GPU. The first two implemented GPU components are a Jacobi-preconditioned conjugate gradient solver for the pressure Poisson equation and the reinitialization of the level set function. Both GPU implementations were sketched. Additionally, an efficient and scalable distributed memory multi-GPU parallelization has been proposed. Our optimization techniques are quite gen-

eral, and can be applied to a variety of numerical algorithms. The performance measurements underline the quality of our GPU implementation as well as the perfect applicability of GPU computations in the field of computational fluid dynamics.

Encouraged by our first GPU computation results, we plan to port our whole fluid solver to the GPU. Since we have already implemented WENO space and Runge-Kutta time discretizations for the level set reinitialization process on graphics hardware, it should be very little effort to apply the GPU kernels to the advection part of our fluid solver. The porting of the remaining CPU code should be straightforward. After a full GPU port of the whole fluid solver, a lot of CPU $\leftrightarrow$ GPU data transfers will become redundant and can be removed. This will result in a completely multi-GPU based high-performance two-phase incompressible Navier-Stokes solver.

Acknowledgements This work was supported in parts by the Sonderforschungsbereich 611 *Singular phenomena and scaling in mathematical models* funded by the *Deutsche Forschungsgemeinschaft*.

References

1. Brackbill JU, Kothe DB, Zemach C (1992) A continuum method for modeling surface tension. J Comput Phys 100(2):335–354, DOI http://dx.doi.org/10.1016/0021-9991(92)90240-Y
2. Chorin AJ (1968) Numerical Solution of the Navier-Stokes Equations. Math Comp 22(104):745–762
3. Cohen J, Molemaker M (2009) A Fast Double Precision CFD Code using CUDA. In: Proceedings of Parallel CFD 2009, Moffett Field, California, USA
4. Croce R, Griebel M, Schweitzer MA (2004) A Parallel Level-Set Approach for Two-Phase Flow Problems with Surface Tension in Three Space Dimensions. Preprint 157, Sonderforschungsbereich 611, Universität Bonn
5. Croce R, Griebel M, Schweitzer MA (2009) Numerical simulation of bubble and droplet deformation by a level set approach with surface tension in three dimensions. Int J Numer Methods Fluids, accepted
6. Griebel M, Dornseifer T, Neunhoeffer T (1998) Numerical Simulation in Fluid Dynamics, a Practical Introduction. SIAM, Philadelphia
7. Griebel M, Metsch B, Oeltz D, Schweitzer MA (2006) Coarse Grid Classification: A Parallel Coarsening Scheme For Algebraic Multigrid Methods. Numer Linear Algebra Appl 13(2–3):193–214
8. Halfhill TR (2008) Parallel Processing with CUDA. Microprocessor Report
9. Harris M (2007) Optimizing Parallel Reduction in CUDA. Tech. rep., NVIDIA Corporation
10. Hoff KE III, Keyser J, Lin M, Manocha D, Culver T (1999) Fast computation of generalized Voronoi diagrams using graphics hardware. In: SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, pp 277–286
11. Hopf M, Ertl T (1999) Hardware Based Wavelet Transformations. In: Workshop '99 on Vision, Modeling and Visualization, Erlangen, Germany
12. Jiang GS, Peng D (1999) Weighted ENO Schemes for Hamilton-Jacobi Equations. SIAM J Sci Comput 21:2126–2143
13. Jiang GS, Shu CW (1996) Efficient implementation of weighted ENO schemes. J Comput Phys 126(1):202–228, DOI http://dx.doi.org/10.1006/jcph.1996.0130
14. Krüger J (2006) A GPU Framework for Interactive Simulation and Rendering of Fluid effects. PhD thesis, Technische Universität München
15. Micikevicius P (2009) 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, pp 79–84, DOI http://doi.acm.org/10.1145/1513895.1513905
16. NVIDIA (2008) CUDA programming guide, Version 2.3. NVIDIA Corporation
17. Osher S, Sethian JA (1988) Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. J Comput Phys 79(1):12–49, DOI http://dx.doi.org/10.1016/0021-9991(88)90002-2
18. Stam J (1999) Stable fluids. In: SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, pp 121–128, DOI http://doi.acm.org/10.1145/311535.311548
19. Steinhoff J, Underhill D (1994) Modification of the Euler equations for "vorticity confinement": Application to the computation of interacting vortex rings. Phys Fluids 6:2738–2744, DOI 10.1063/1.868164
20. Strybny J, Thorenz C, Croce R, Engel M (2006) 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
21. Thibault JC, Senocak I (2009) CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows. In: Proceedings of the 47th AIAA Aerospace Sciences Meeting, Orlando, Florida, USA