A Geometric Multigrid Solver on Tsubame 2.0

Harald Köstler[1(✉)], Christian Feichtinger[1], Ulrich Rüde[1], and Takayuki Aoki[2]

[1] Chair for System Simulation, University of Erlangen-Nuremberg, Erlangen, Germany
Harald.Koestler@informatik.uni-erlangen.de
[2] Global Scientific Information and Computing Center, Tokyo Institute of Technology, Yokohama, Japan

Abstract. Tsubame 2.0 is currently one of the largest installed GPU clusters and number 5 in the Top 500 list ranking the fastest supercomputers in the world. In order to make use of Tsubame, there is a need to adapt existing software design concepts to multi-GPU environments. We have developed a modular and easily extensible software framework called waLBerla that covers a wide range of applications ranging from particulate flows over free surface flows to nano fluids coupled with temperature simulations and medical imaging. In this article we report on our experiences to extend waLBerla in order to support geometric multigrid algorithms for the numerical solution of partial differential equations (PDEs) on multi-GPU clusters. We discuss the software and performance engineering concepts necessary to integrate efficient compute kernels into our waLBerla framework and show first weak and strong scaling results on Tsubame for up to 1029 GPUs for our multigrid solver.

Keywords: GPGPU · CUDA · Parallel multigrid solver · waLBerla · Tsubame 2.0

1 Introduction

Many imaging applications exhibit high memory and compute power requirements, either due to the large amount of data being processed or runtime restrictions e.g. for real-time imaging. Graphics processing units (GPUs) typically offer hundreds of specialized compute units operating on dedicated memory and reach outstanding compute and memory performance in this way. Therefore, they are more and more used for compute-intensive applications also in imaging. GPUs are best suitable for massively-data parallel algorithms, inadequate problems, that e. g. require a high degree of synchronization or provide only limited parallelism, are left to the host CPU. For high performance computing (HPC) heterogeneous multi-GPU clusters are built up consisting of thousands of GPUs. In the Top 500 list[1] from November 2011 of the fastest machines world-wide there were three of these multi-GPU clusters in the Top 5.

[1] http://www.top500.org, Nov. 2011.

A. Bruhn et al. (Eds.): Global Optimization Methods, LNCS 8293, pp. 155–173, 2014.
DOI: 10.1007/978-3-642-54774-4_8,

However, in order to achieve good performance on these clusters, software development has to adapt to the new needs of the massively parallel hardware. As a starting point, GPU vendors offer proprietary environments for general purpose GPU computing. NVIDIA, e. g., provides the possibility to write single-source programs that execute kernels written in a subset of C and C++ on their Compute Unified Device Architecture (CUDA) [1]. An alternative would have been to use the Open Compute Language (OpenCL)[2]. Within OpenCL one can write code that runs in principle on many different hardware platforms, e. g. Intel CPUs, ATI/AMD or NVIDIA GPUs, and even the ICM Cell processor, but to achieve optimal performance the implementation has to be adapted to the specific features of the hardware. Since we are exclusively working on NVIDIA GPUs in this article and we found no considerable difference in the kernel performance if we tune OpenCL towards NVIDIA GPUs, we have done our implementations in CUDA. Both CUDA and OpenCL are low-level languages. To make code development more efficient, one either has to provide wrappers for high-level languages like e.g. OpenMP [2] and PyCUDA [3] or easy to use frameworks, where we follow the latter approach.

Our contributions in this article are specifically that

- we discuss the concepts necessary to integrate efficient GPU compute kernels for a geometric multigrid solver into our software framework waLBerla that is discussed in more detail in Sect. 3,
- and then show first weak and strong scaling results of our solver on Tsubame 2.0 located in Japan.

While waLBerla was at first developed for simulating fluid flow using the Lattice Boltzmann method on 3D structured domains, it is now also capable of solving elliptic PDEs like Poisson's equation numerically via multigrid.

One possible imaging application for our multigrid solver is high dynamic range (HDR) compression. HDR tries to allow a wide dynamic range of luminance between the lightest and darkest areas within an image. Often, HDR compression is only one step within the image acquisition pipeline and there are hard time constraints that have to be met in practical applications. In [4] one finds a state-of-the-art HDR compression algorithm in the gradient space that can be accelerated by our multigrid solver. In general, for gradient space imaging one has to transform an input image $I : \Omega \mapsto \mathbb{R}$ defined in the domain $\Omega \subset \mathbb{R}^3$ to gradient space and back. While the forward transformation to gradient space is fast by using simple finite differences to obtain the image gradient ∇I, the backward transformation requires the solution of Poisson's equation

$$\Delta u = f \quad \text{in } \Omega \tag{1a}$$

$$u = 0 \quad \text{on } \partial\Omega \tag{1b}$$

typically assuming homogeneous Dirichlet boundary conditions. Here,

$$f = \operatorname{div}(\Phi \nabla I) \ , \tag{2}$$

[2] http://www.khronos.org/opencl/, Mai 2012.

where $\Phi\nabla I$ are compressed dynamic range image derivatives and $\Phi : \mathbb{R}^3 \mapsto \mathbb{R}$ is a position-dependent attenuating function (see [4] for more details). The solution $u : \Omega \mapsto \mathbb{R}$ is the HDR compressed image.

Most of the overall runtime for HDR compression is spent in the numerical solution of (1a, 1b), where we can apply a parallel, geometric multigrid solver.

Besides HDR compression there are a variety of applications in imaging and computer vision, where multigrid methods are used. Especially for variational models the arising Euler-Lagrange equations are often treated via efficient multigrid solvers. In this way, applications ranging from image denoising, image inpainting, and image segmentation to optical flow and image registration are found (see [5] for more details about different multigrid methods and for further references).

There exists already also a variety of other implementations of different multigrid algorithms on GPU like in [6,7], conjugate gradients (CG) and multigrid on NVIDIA GeForce FX [8], mixed-precision multigrid solvers [9], finite element multigrid solvers on GPU clusters [10,11], or algebraic multigrid [12]. Parallel multigrid methods on GPUs are incorporated in software packages like OpenCurrent [13] or PETSc [14], and GPU multigrid is also used in imaging, e.g. for nonlinear denoising or variational optical flow (see e.g. [15–17]).

In previous work, we have run a multi-GPU Lattice Boltzmann simulation on Tsubame [18] and highly scalable multigrid solvers on CPU clusters [19–21]. Furthermore, we optimized a 2D multigrid solver on GPU to do real-time HDR compression [22] for a series of X-ray images. In addition to that, we show weak and strong scaling results on an IBM Bluegene/P up to nearly 300.000 cores and an Intel CPU cluster in [23], where we used a 3D multigrid solver on a block-structured tetrahedral finite element mesh. Now we integrate a multi-GPU geometric multigrid solver in waLBerla. An alternative is to implement a finite element based multigrid solver on GPU for gradient space imaging [24], however, it is computationally more expensive than our finite difference based solver on a regular grid. Note that our multigrid solver scales also on CPU-clusters [25] and works also for more general elliptic PDEs with variable coefficients [26].

The paper is organized as follows: In Sect. 2 we briefly describe the multigrid algorithm and its parallelization on GPUs. Section 3 summarizes the MPI-parallel waLBerla framework that easily enables us to extend our code to multi-GPUs. The hardware details of the Tsubame 2.0 cluster and a simple performance model for our multigrid solver to estimate the runtime of our software are introduced in Sect. 4. In Sect. 5 we present weak and strong scaling results on Tsubame 2.0 before concluding the paper in Sect. 6.

2 Parallel Multigrid

2.1 Multigrid Algorithm

Multigrid is not a single algorithm, but a general approach to solve problems by using several levels or resolutions [27,28]. We restrict ourselves to *geometric multigrid (MG)* in this article that identifies each level with a (structured) grid.

Typically, multigrid is used as an iterative solver for large linear systems of equations that have a certain structure, e.g. that arise from the discretization of PDEs and lead to sparse and symmetric positive definite system matrices. The main advantage of multigrid solvers compared to other solvers like CG is that multigrid can reach an asymptotically optimal complexity of $\mathcal{O}(N)$, where N is the number of unknowns or grid points in the system. For good introductions and a comprehensive overview on multigrid methods, we, e.g., refer to [29,30], for details on efficient multigrid implementations see [31–33].

We assume that we want to solve the PDE (1a, 1b) with solution $\boldsymbol{u} : \mathbb{R}^3 \to \mathbb{R}$, right hand side (RHS) $f : \mathbb{R}^3 \to \mathbb{R}$, and Dirichlet boundary conditions on a rectangular domain $\Omega \subset \mathbb{R}^3$. Equation (1a, 1b) is discretized by finite differences on a structured grid. This results in a linear system

$$A^h u^h = f^h \,, \quad \sum_{j \in \Omega^h} a_{ij}^h u_j^h = f_i^h \,, i \in \Omega^h \tag{3}$$

with system matrix $A^h \in \mathbb{R}^{N \times N}$, unknown vector $u^h \in \mathbb{R}^N$ and right hand side (RHS) vector $f^h \in \mathbb{R}^N$ on a discrete grid Ω^h with mesh size h.

In order to solve the above linear system, we note that during the iteration the algebraic error $e^h = u_*^h - u^h$ is defined to be the difference between the exact solution u_*^h of Eq. (3) and the approximate solution u^h. With the residual equation $r^h = f^h - A^h u^h$ we obtain there so-called error equation

$$A^h e^h = r^h. \tag{4}$$

The multigrid idea is now based on two principles:

Smoothing Property: Classical iterative solvers like red-black Gauß-Seidel (RBGS) are able to smooth the error after very few steps. That means the high frequency components of the error are removed well by these methods. But they have little effect on the low frequency components. Therefore, the convergence rate of classical iterative methods is good in the first few steps and decreases considerably afterward.

Coarse Grid Principle: A smooth function on a fine grid can be approximated satisfactorily on a grid with less discretization points, whereas oscillating functions would disappear. Furthermore, a procedure on a coarse grid is less expensive than on a fine grid. The idea is now to approximate the low frequency error components on a coarse grid.

Multigrid combines these two principles into one iterative solver. The smoother reduces the high frequency error components first, and then the low frequency error components are approximated on coarser grids, interpolated back to the finer grids and eliminated there. In other words, on the finest grid Eq. (1a, 1b) first is solved approximately by a few smoothing steps and then an approximation to the error equation is computed on the coarser grids. This leads to recursive algorithms which traverse between fine and coarse grids in a grid hierarchy. Two successive grid levels Ω^h and Ω^H typically have fine mesh size h and coarse mesh size $H = 2h$.

One multigrid iteration, here the so-called *V-cycle*, is summarized in Algorithm 1. Note that in general the operator A^h has to be computed on each grid level. This is either done by rediscretization of the PDE or by Galerkin coarsening, where $A^H = RA^hP$.

Algorithm 1 Recursive V-cycle: $u_h^{(k+1)} = V_h(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2)$

1: **if** coarsest level **then**
2: solve $A^h u^h = f^h$ exactly or by several CG iterations
3: **else**
4: $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}(u_h^{(k)}, A^h, f^h)$ {presmoothing}
5: $r^h = f^h - A^h \bar{u}_h^{(k)}$ {compute residual}
6: $r^H = Rr^h$ {restrict residual}
7: $e^H = V_H(0, A^H, r^H, \nu_1, \nu_2)$ {recursion}
8: $e^h = Pe^H$ {interpolate error}
9: $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ {coarse grid correction}
10: $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}(\tilde{u}_h^{(k)}, A^h, f^h)$ {postsmoothing}
11: **end if**

In our node-based multigrid solver we use the following components:

- A ω-RBGS (or red-black SOR) smoother $\mathcal{S}_h^{\nu_1}, \mathcal{S}_h^{\nu_2}$ with ν_1 pre- and ν_2 postsmoothing steps.
- The restriction operator R from fine to coarse grid is full weighting.
- We apply a trilinear interpolation operator P for the error.
- The coarse grid problem is solved by a sufficient number of CG iterations.
- The discretization of the Laplacian was done via the standard 7-point stencil (cf. Eq. (1a, 1b)), on coarser grids we rediscretize the Laplacian.

Note that the required number of CG iterations on the coarsest grid is proportional to the diameter of the computational domain (see e.g. [23,34]) and thus increases linearly with growing diameter.

2.2 GPU Implementation

To implement the multigrid algorithm on GPU we have to parallelize it and write kernels for smoothing, computation of the residual, restriction, and interpolation together with coarse grid correction. In the following, we choose the ω-RBGS kernel as an example and discuss it in more detail. Algorithm 2 shows the source code of a straightforward, unoptimized CUDA RBGS kernel. Here, the `solution` and `rhs` fields are stored in global GPU memory. Due to the splitting in red and black points within the RBGS to enable parallelization, only every second solution value is written back, whereas the whole solution vector is processed.

Algorithm 2 ω-red-black Gauß-Seidel smoother kernel in CUDA.

```
__global__ void kernel_RBGS(Real* solution, Real* rhs,
                            const Uint xSize, const Uint ySize,
                            const Uint zSize, const Uint red_black)
{

  unsigned int x = threadIdx.x;
  unsigned int y = blockIdx.x;
  unsigned int z = blockIdx.y;
  Real w = 1.15;

  if ((x > 0) && (y > 0) && (z > 0) &&
      (x < xSize-1) && (y < ySize-1) && (z < zSize-1) )
  {

    Real new_val = (1-w)*GET3D(solution,x,y,z)
    + w*((GET3D(rhs,x,y,z) + GET3D(solution,x-1,y,z)
    + GET3D(solution,x+1,y,z) + GET3D(solution,x,y+1,z)
    + GET3D(solution,x,y-1,z) + GET3D(solution,x,y,z+1)
    + GET3D(solution,x,y,z-1)) / Real(6.));

    if (((x+y+z)%2) == red_black)
      GET3D(solution,x,y,z) = new_val;
  }
}
```

Possible optimizations are e.g. to split the red and black points into separate arrays in memory, or blocking techniques (see [22] for a detailed performance analysis in 2D). Additionally, the thread block size depends on the number of grid points in x-direction. Best performance can be achieved for larger thread block sizes, e.g. 256 or 512, therefore the kernel becomes inefficient for a smaller number of grid points in x-direction and 2D thread blocks become necessary.

For multi-GPU, the distributed memory parallelization is simply done by decomposing each grid into several smaller sub-grids and introducing a layer of ghost cells between them. Now the sub-grids can be distributed to different MPI processes and only the ghost cells have to be communicated to neighboring sub-grids. The function calling the kernel handles the ghost cell exchange. Buffers are sent to neighboring processes via communication routines provided by the waLBerla framework introduced in the next section. Within Algorithm 1 one has to exchange the ghost layer of the solution resp. the error after smoothing and interpolation (steps 4, 8, and 10), the ghost layer of the residual after step 5. On the coarsest level we have only a few grid points left per sub-grid and thus we transfer the whole RHS from GPU to CPU and do the parallel CG iterations on CPU. After that, the solution is transfered back from CPU to GPU.

3 Walberla

WaLBerla is a massively parallel software framework developed for HPC applications on block-structured domains [35]. It has been successfully used in many multi-physics simulation tasks ranging from free surface flows [36] to particulate flows [37] and fluctuating lattice Boltzmann [38] for nano fluids.

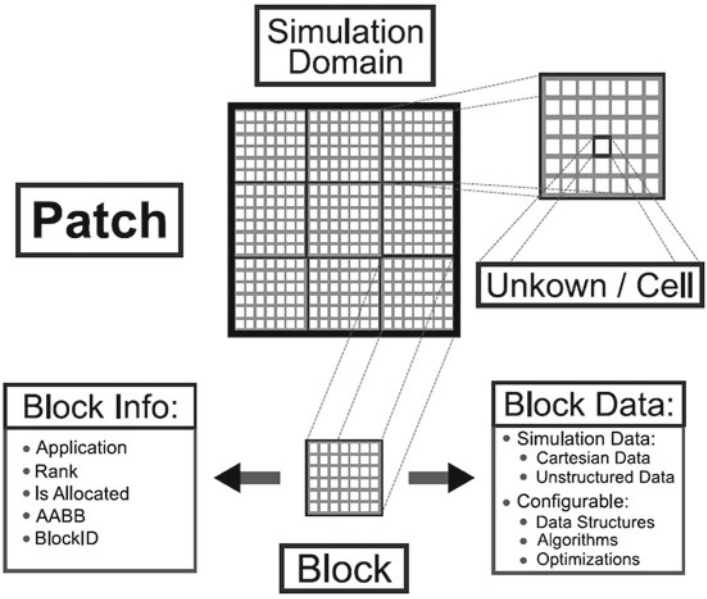


Fig. 1. Patches and Blocks in waLBerla [39].

The main design goals of the waLBerla framework are to provide excellent application performance across a wide range of computing platforms and the easy integration of new algorithms. The current version waLBerla 2.0 is capable of running heterogeneous simulations on CPUs and GPUs with static load balancing [39].

3.1 Patch, Block, and Sweep Concept

A fundamental design concept of waLBerla is to rely on block-structured grids, what we call our Patch and Block data structure. We restrict ourselves to block-structured grids in order to support efficient massively parallel simulations.

In our case a Patch denotes a cuboid describing a region in the simulation that is discretized with the same resolution (see Fig. 1) . This Patch is further subdivided into a Cartesian grid of Blocks consisting of cells. The actual simulation data is located on these cells. In parallel one or more Blocks can be assigned to each process in order to support load balancing strategies. Furthermore, we may specify for each Block, on which hardware it is executed. Of course, this requires also to be able to choose different implementations that run on a certain Block, what is realized by our functionality management.

The functionality management in waLBerla 2.0 controls the program flow. It allows to select different functionality (e.g. kernels, communication functions) for different granularities, e.g. for the whole simulation, for individual processes, and for individual Blocks.

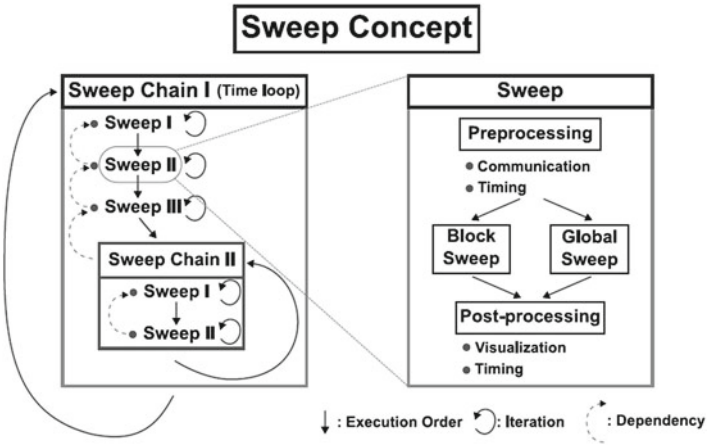


Fig. 2. Sweep concept in waLBerla [39].

When the simulation runs, all tasks are broken down into several basic steps, so-called Sweeps. A Sweep consists of two parts as shown in Fig. 2: a communication step fulfilling the boundary conditions for parallel simulations by nearest neighbor communication and a communication independent work step traversing the process-local Blocks and performing operations on all cells. The work step usually consists of a kernel call, which is realized for instance by a function object or a function pointer. As for each work step there may exist a list of possible (hardware dependent) kernels, the executed kernel is selected by our functionality management.

3.2 MPI Parallelization

The parallelization of waLBerla can be broken down into three steps:

1. a data extraction step,
2. a MPI communication step, and
3. a data insertion step.

During the data extraction step, the data that has to be communicated is copied from the simulation data structures of the corresponding Blocks. Therefore, we distinguish between process-local communication for Blocks lying on the same and MPI communication for those on different processes.

Local communication directly copies from the sending Block to the receiving Block, whereas for the MPI communication the data has first to be copied into buffers. For each process to which data has to be sent, one buffer is allocated. Thus, all messages from Blocks on the same process to another process are serialized. To extract the data to be communicated from the simulation

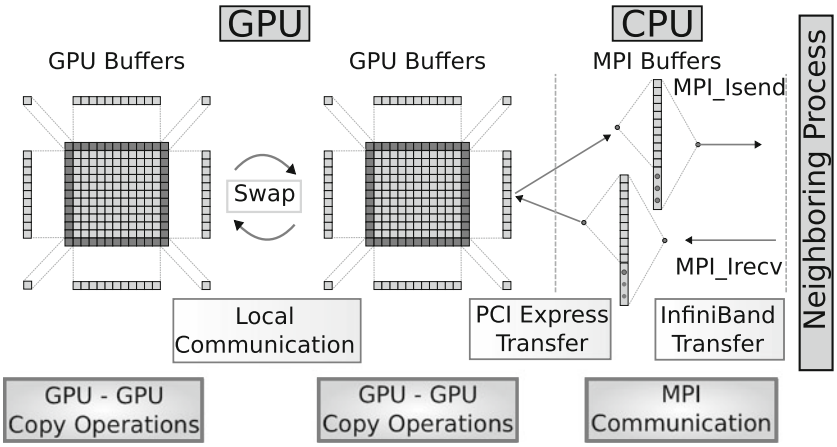


Fig. 3. Communication concept within WaLBerla [39]. Depicted is a process having two Blocks. Communication between the process-local Blocks is realized by swapping of the corresponding buffers, whereas MPI communication involves PCIe transfers of the GPU buffers. GPU-GPU copy operations are required to extract and insert data from the data fields to and from the buffers.

data, extraction function objects are used that are again selected via the functionality management. The data insertion step is similar to the data extraction, besides that we traverse the block messages in the communication buffers instead of the Blocks.

3.3 Multi-GPU Implementation

For parallel simulations on GPUs, the boundary data of the GPU has first to be copied by a PCIe transfer to the CPU and then be communicated via MPI routines. Therefore, we need buffers on GPU and CPU in order to achieve fast PCIe transfers. In addition, on-GPU copy kernels are added to fill these buffers. The whole communication concept is depicted in Fig. 3.

The only difference between parallel CPU and GPU implementation is that we need to adapt the extraction and insertion functions. For the local communication they simply swap the GPU buffers, whereas for the MPI communication we copy the data directly from the GPU buffers into the MPI buffers and vice versa. To support heterogeneous simulations on GPUs and CPUs, we execute different kernels on CPU and GPU and also define a common interface for the communication buffers, so that an abstraction from the hardware is possible. Additionally, the work load of the CPU and the GPU processes can be balanced e.g. by allocating several Blocks on each GPU and only one on each CPU-only process. In addition to that it is also possible to divide a Block into several Sub-Blocks of different sizes to enable load balancing on heterogeneous compute nodes containing e.g. GPUs and CPUs.

Table 1. Specifications of the Tsubame 2.0 cluster.

Compute Nodes	1408
Processor	Intel Xeon X5670
GPU	NVIDIA Tesla M2050
GPUs per Compute Node	3
LINPACK[a] Performance	1192 TFLOPS
Power Consumption	1398.61 KW
Flops per Watt	852.27 FLOPS/W
Network Type	Fat Tree
Interconnect	QDR Infiniband

[a] http://www.netlib.org/linpack

4 Hardware and Performance Model

4.1 Tsubame 2.0

We perform all numerical tests in this article on Tsubame 2.0[3] that is currently (Nov. 2011) number 5 in the TOP 500 list. The detailed hardware specifications of this multi-GPU cluster are listed in Table 1.

All 1408 compute nodes are equipped with three NVIDIA Tesla M2050 GPU accelerators each having 3 GB of GPU memory. NVIDIA Tesla M2050 has a floating-point performance (single precision) of 1030 GFLOP/s and 515 GFLOP/s (double precision) coming from 448 CUDA streaming processors capable of doing 2 floating point operations per cycle and a processor frequency of 575 MHz. Thus, most of Tsubame's 2.4 PFlops peak performance comes from its 4224 GPUs. The GPU memory frequency is 1550 MHz with DDR5 (factor 2) RAM and a 384 Bit memory bus what results in 148 GB/s peak memory bandwidth.

4.2 Performance Model

Next we derive a very simple performance model for our multigrid solver in order to identify performance bottlenecks and to estimate the overall runtime for a given problem size on Tsubame 2.0. In general, we can split the runtime t into the time for the compute kernels, e.g. for the smoother, restriction or interpolation, and the time for communicating data between neighboring processes, mainly exchanging ghost layers after smoothing, residual, and interpolation. An important decision is, if one overlaps computation and communication. If we do not overlap them, the runtime is just the sum

$$t = t_{\text{kernel}} + t_{\text{buffer}} + t_{\text{PCI}} + t_{\text{MPI}} \tag{5}$$

of runtime of all kernels t_{kernel}, the time for copying data from ghost layers to send and receive buffers t_{buffer}, the time for PCIe transfers t_{PCI}, and the time for MPI communication t_{MPI}.

[3] http://www.gsic.titech.ac.jp/en/tsubame2, Nov. 2011.

In order to enable overlapping, we have to split the kernels into inner kernels and outer kernels, where the latter are just processing the points lying near boundary layers. After the outer kernels are finished and the GPU buffers are filled, we can communicate the ghost layers via several CUDA streams and asynchronous PCIe transfers. In parallel run the inner kernels, i.e.

$$t = t_{\text{o,kernel}} + t_{\text{buffer}} + \max(t_{\text{i,kernel}}, t_{\text{PCI}} + t_{\text{MPI}}). \tag{6}$$

Kernel Performance. First we take a closer look at the kernel performance on a single GPU. From previous work we already know that our multigrid algorithm is bounded by memory bandwidth (this also can be easily checked e.g. by a profiler). For the most time consuming part, the smoother, where we basically do a sparse matrix (stencil) vector product, we have to load per grid point one value of the right hand side, seven values of the solution, and we store one value of the solution. The loads of the solution can be partly cached (at least three rows in one grid plane), such that we can assume to require only one load per plane in the solution array, i.e. instead of seven we have three loads. Since we do not split the red and black points of the solution into separate arrays in memory, we assume that we must load and store the full array twice, once within the update iteration of the red and once of the black points. Table 2 summarizes the estimated load and store instructions for the different multigrid components. We denote the number of grid points on the finest grid $l = 0$ by $N = N_0$. On the coarser grids we have $N_l = \frac{N_{l-1}}{8}$ grid points in 3D. Thus, the overall number of grid points on $L-1$ grid levels is roughly $N_{\text{mg}} = N_0 \cdot (1 + \frac{1}{8} + \ldots + \frac{1}{8^L}) \approx N_0 \cdot 1.14$.

Table 2. Number of load and store instructions for different multigrid components per (fine) grid point. Additionally we list the required number of ghost layer exchanges in the multi-GPU case.

Component	Loads	Stores	Ghost layer exchanges
Smoothing	$2 \cdot (3 + 1) = 8$	2	2
Residual	$3 + 1 = 4$	1	1
Restriction	3	$\frac{1}{8}$	0
ProlongationAdd	$1 + \frac{3}{8}$	1	1

Table 3. Memory bandwidths on Tsubame 2.0 for data transfers within one GPU (DDR5), between two GPUs on one compute node (PCIe), and between two compute nodes (Infiniband).

	Theoretical Memory Bandwidth [GB/s]	Attainable Memory Bandwidth [GB/s]
DDR5	148	95
PCIe	8	6 (shared by 3 GPUs)
Infiniband (bidirectional)	5	5

Since we know that our multigrid solver is bandwidth limited we can estimate the kernel time and communication time from the maximum attainable memory and network bandwidth that we measured in Table 3 via standard streaming benchmarks. Note that we neglect the fact that the PCIe bandwidth for GPU to CPU resp. GPU to CPU transfers differs and that the bandwidth depends on the message size, i.e. for smaller message sizes the bandwidth is much lower and the latency dominates.

As an example one ω-RBGS iteration for problem size $N = 512^2 \times 256$ takes 60.5 ms what corresponds to approximately 89 GB/s memory bandwidth on one GPU. All our numerical tests run with double floating-point precision. Our performance model with data from Tables 2 and 3 estimates

$$t_{\text{RBGS}} = \frac{8 \cdot N \cdot 10}{95\,\text{GB/s}} \approx 56.5\,\text{ms}\,. \tag{7}$$

Thus, our model is quite accurate for the RBGS smoother. However, this holds only for larger problem sizes. In order to show the dependency of our smoother on the problem size we depict the runtimes and bandwidth of one ω-RBGS iteration with varying sizes in Fig. 4.

For smaller problems, the GPU overhead e.g. for CUDA kernel calls becomes visible and there is not enough work to be done in parallel and thus most of the compute cores idle.

For one multigrid V(2,2)-cycle with 6 grid levels we measure 364 ms on one GPU (corresponding to approximately 85 GB/s memory bandwidth), our performance model predicts 325 ms.

In order to give more insight in the runtime behavior of the different parts of the multigrid solver, Fig. 5 shows the portions of predicted and measured runtime spent in different components of a V(2,2)-cycle on one GPU. The problem size shrinks by a factor of 8 for each grid level, thus one expects the coarse grid (this includes all the previous components on all coarser grids plus solving the

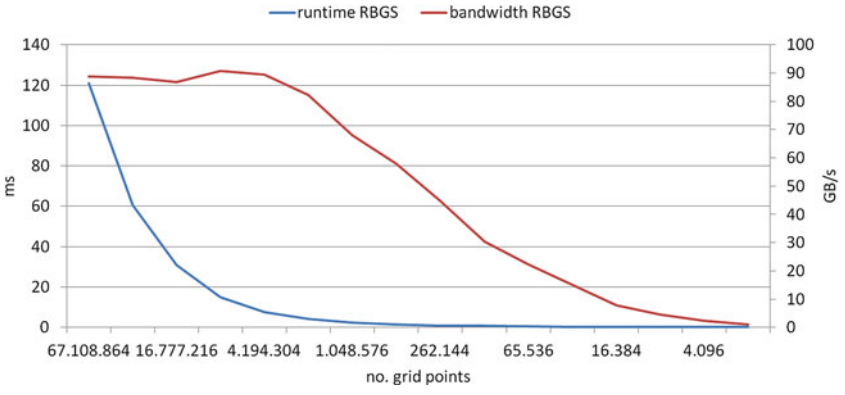
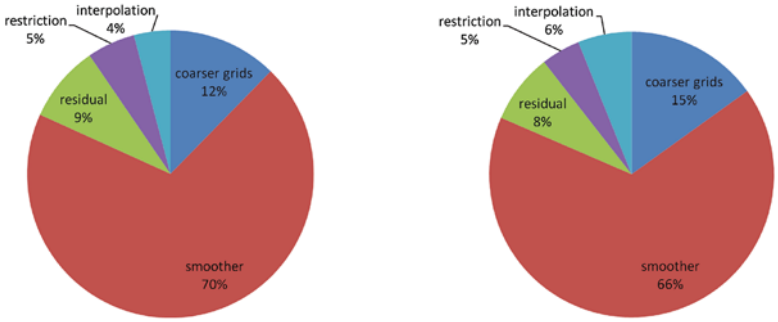


Fig. 4. Single GPU ω-RBGS runtime and bandwidth for different problem sizes.



(a) (b)

Fig. 5. Runtime percentage for different components predicted by our performance model (a) and measured (b) on one GPU (problem size $512^2 \times 256$).

problem on the coarsest grid with CG iterations) to require about 1/8 of the runtime. The measurement lies a little bit higher especially for GPUs, because the smaller sizes are not very efficient on GPU as seen before.

Summarizing, our predictions on one GPU are quite accurate and the model error is typically below 10 %.

For overlapping computation and communication we split the smoother kernel into an inner and outer kernel. This increases the runtime e.g. for one ω-RBGS iteration for problem size $512^2 \times 256$ by approximately 6 % on one GPU. Therefore, we assume $t_{\text{i,kernel}} = t_{\text{kernel}}$ and $t_{\text{o,kernel}} = 0.06 \cdot t_{\text{kernel}}$ in the following. A simple bandwidth based runtime estimate for $t_{\text{o,kernel}}$ is not feasible because of the relatively small number of boundary points and the non-contiguous memory accesses for four of the six boundary layers.

Communication Time. The same problems as for $t_{\text{o,kernel}}$ we also have when trying to estimate t_{buffer} for the multi-GPU case. Thus, we also fall back to measured times here that depend on the number of neighboring processes. In worst case, six ghost layers have to be copied into and from buffers. From this we measure $t_{\text{buffer}} \approx 0.05 \cdot t_{\text{kernel}}$. t_{PCI} and t_{MPI} we are able to predict using information about the number of ghost layer exchanges from Table 2 and the bandwidths from Table 3. Note that within one smoothing iteration we have two ghost layer exchanges, one after updating the red points and one after updating the black points. The PCIe transfer time is the sum of the transfer from GPU to CPU and back (if more than one of the three GPUs on a compute node is used the attainable PCIe bandwidth is shared and thus reduces to 3 resp. 2 GB/s). We neglect that the MPI communication time differs from within one node and between two nodes.

For problem size $N = 512^2 \times 256$ the number of ghost points on the six boundary planes is $512^2 + 4 \cdot 256 \cdot 512$, i.e. the surface to volume ratio is $1 : 64$. On the

coarser grids the ratio goes down to 1 : 2 on grid level 6. In this setting we measure 89 ms on 48 GPUs for one ω-RBGS iteration on the finest grid level, if we do not overlap computation and communication. Our communication model predicts $t_{\text{PCI}} = 16.8\,\text{ms}$ and $t_{\text{MPI}} = 3.4\,\text{ms}$, i.e. $t_{\text{RBGS}} = 56.5 + 16.8 + 3.4 + 5 = 81.7\,\text{ms}$.

To estimate the overall time for overlapping computation and communication, we observe that the sum $t_{\text{PCI}} + t_{\text{MPI}}$ is much lower than the time for an inner smoothing kernel, therefore the communication time should not be visible for the parallel smoother, i.e. $t = t_{\text{o,kernel}} + t_{\text{buffer}} + t_{\text{i,kernel}}$.

5 Scaling Experiments

Next we check the achievable parallel efficiency and speedup of our multigrid multi-GPU implementation on Tsubame 2.0. Baseline is the performance on one GPU.

We distinguish two types of experiments: *Weak scaling* relates to experiments were the problem size is increased linearly with the number of involved GPUs, whereas the term *strong scaling* implies that we have a constant global problem size and vary only the number of processes. Assuming a perfect parallelization, we expect the runtime to be constant in weak scaling experiments, while we expect the runtime to be reciprocally proportional to the number of parallel processes in strong scaling experiments. To estimate the quality of our parallelization we compute speedup $S_p = \frac{t_1}{t_p}$ and parallel efficiency $E_p = \frac{S_p}{p}$ given the runtimes t_1 and t_p on one and on p GPUs.

We measure the runtime of one V(2,2)-cycle (i.e. V-cycles with two ω-RBGS iterations for pre- and postsmoothing each) on six grid levels with parameters from Sect. 2, if not stated otherwise. On the coarsest grid between 15 and 20 CG iterations are performed. All our experiments are done with double floating point accuracy.

5.1 Weak Scaling

Figure 6 shows the weak scaling behavior of the code for problem size $512^2 \times 256$ for non-overlapping communication and computation and when overlapping communication and computation within the smoother. Here, we measure the time spent to do pre- and postsmoothing (step 4 and 10 in Algorithm 1) on the finest grid level (Smoother fine), the time spent to solve the problem on all coarser grid levels (Coarser Grids), and the overall time for one V(2,2)-cycle. In addition to that the efficiency for one V-cycle is shown. In contrast to nearly perfect weak scaling also on large CPU clusters (cf. [23]) the overall parallel efficiency drops to 35 % in the first case and to 42 % in the second case on 1029 GPUs, what was the maximum number of GPUs available for us on Tsubame 2.0. This has mainly two reasons: first the effect of additional intra-node memory transfers of ghost layers between CPU and GPU via the PCIe bus when using GPUs, and second the CG iterations on the coarsest grid that are done on CPU and require a global all-to-all communication. Overlapping of computation and

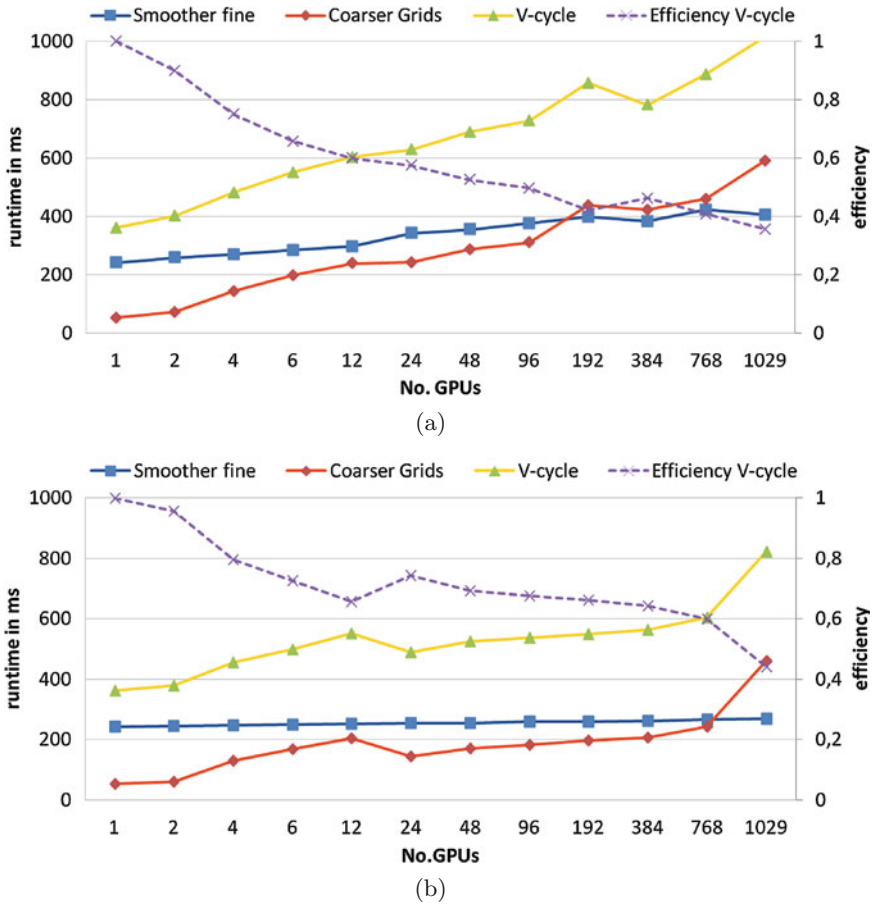


Fig. 6. Weak scaling behavior and parallel efficiency for non-overlapping communication (a) and overlapping communication in the smoother (b) from one to 1029 GPUs on Tsubame performing one multigrid V(2,2)-cycle.

communication within the smoother improves the parallel efficiency and the overall runtime on 1029 GPUs is about 870 ms in this case, where 40 % of the time are spent within the smoother and about 30 % on the coarsest grid doing CG iterations.

5.2 Strong Scaling

Next, we scale the number of involved processing units, but leave the total problem size, i.e. the number of grid points, constant. In this subsection we do not overlap communication and computation. Figure 7 shows the runtimes for $512^2 \times 256$ solved on up to 16 GPUs. The maximal speedup is just 2.3 achieved on 8 GPUs, which is a result of different factors: on the one hand the problems

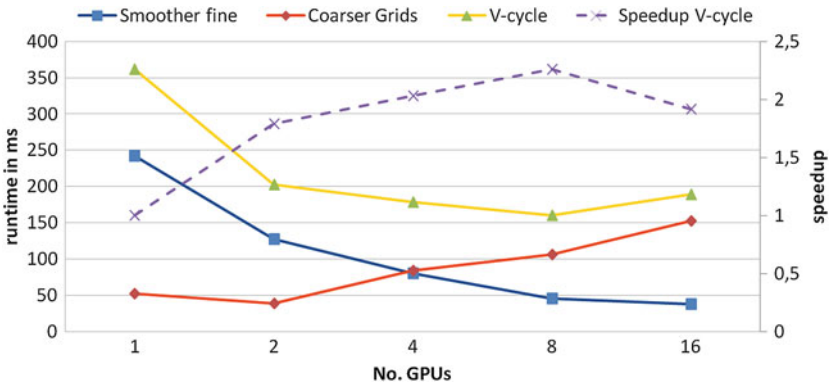


Fig. 7. Strong scaling and speedups for one V(2,2)-cycle with 512×256^2 grid points per GPU.

Table 4. Runtimes of one V(2,2)-cycle for varying problem sizes and 5 or 6 grid levels on 1029 GPUs

Unknowns (in million)	No. of levels	Runtime in ms
69055	6	1025
34528	6	583
17264	6	**322**
8632	5	461
4316	5	261
2158	5	171
1079	5	127
539	5	**125**
270	5	130

for small size mentioned when discussing the single-node performance and on the other hand the communication overhead addressed within the weak scaling experiments.

Table 4 shows runtime results for different problem sizes on 1029 GPUs in order to determine the optimal problem size on this number of GPUs. For the largest run taking 1025 ms our performance model predicts only 445 ms with a ratio computation to communication of 2.7 : 1, the model error is mainly caused by the coarse grid solver. The minimal runtime on 1029 GPUs we find for 539 million grid points, here one V(2,2)-cycle takes 125 ms and communications dominates computation roughly 4 : 1 due to our performance model.

6 Conclusions and Future Work

We have implemented a geometric multigrid solver on GPU and integrated it into the waLBerla framework. First results show acceptable scalability on Tsubame 2.0 up to 1029 GPUs.

One of the next steps is a performance optimization of our code. On one GPU, one obvious improvement would be to use an optimized data layout by splitting the red and black grid points into two separate arrays in memory. In the multi-GPU case we next implement overlapping communication also for the remaining multigrid components besides the smoother. We will also further investigate the CG coarse grid solver and possible alternative parallel (direct) solvers. It is possible to refine the performance model, e.g. to take into account different bandwidths for each grid level like in [22] or to model the performance of the CG solver as done in [23]. The next major change in waLBerla will be to support local grid refinement within the computational domain. Besides adaptive multigrid methods this allows us to reduce the number of processes on coarser grids to achieve a better scalability.

Acknowledgment. We are grateful to have the opportunity to test our multigrid solver on Tsubame 2.0.

References

1. NVIDIA Cuda Programming Guide 4.2. http://developer.nvidia.com/nvidia-gpu-computing-documentation (2012)
2. Ohshima, S., Hirasawa, S., Honda, H.: OMPCUDA : OpenMP execution framework for CUDA based on omni OpenMP compiler. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 161–173. Springer, Heidelberg (2010)
3. Klöckner, A., Pinto, N., Lee, Y., Catanzaro, B., Ivanov, P., Fasih, A.: PyCUDA and PyOpenCL: a scripting-based approach to GPU run-time code generation. Parallel Comput. **38**, 157–174 (2012)
4. Fattal, R., Lischinski, D., Werman, M.: Gradient domain high dynamic range compression. ACM Trans. Graph. **21**(3), 249–256 (2002)
5. Köstler, H.: A Multigrid Framework for Variational Approaches in Medical Image Processing and Computer Vision. Verlag Dr. Hut, München (2008)
6. Goodnight, N., Woolley, C., Lewin, G., Luebke, D., Humphreys, G.: A multigrid solver for boundary value problems using programmable graphics hardware. In: ACM SIGGRAPH 2005 Courses, p. 193. ACM Press, New York (2005)
7. Feng, Z., Li, P.: Multigrid on GPU: tackling power grid analysis on parallel simt platforms. In: IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2008, pp. 647–654. IEEE Computer Society, Washington, DC (2008)
8. Bolz, J., Farmer, I., Grinspun, E., Schröder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In: ACM SIGGRAPH 2003 Papers, pp. 917–924. ACM (2003)
9. Göddeke, D., Strzodka, R.: Cyclic reduction tridiagonal solvers on gpus applied to mixed-precision multigrid. IEEE Trans. Parallel Distrib. Syst. **22**(1), 22–32 (2011)
10. Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Wobker, H., Becker, C., Turek, S.: Using GPUs to improve multigrid solver performance on a cluster. Int. J. Comput. Sci. Eng. **4**(1), 36–55 (2008)
11. Göddeke, D.: Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters. Logos Verlag, Berlin (2011)

12. Haase, G., Liebmann, M., Douglas, C.C., Plank, G.: A parallel algebraic multigrid solver on graphics processing units. In: Zhang, W., Chen, Z., Douglas, C.C., Tong, W. (eds.) HPCA 2009. LNCS, vol. 5938, pp. 38–47. Springer, Heidelberg (2010)
13. Cohen, J.: OpenCurrent, Nvidia research. http://code.google.com/p/opencurrent/ (2011)
14. Balay, S., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Web page. http://www.mcs.anl.gov/petsc (2009)
15. Grossauer, H., Thoman, P.: GPU-based multigrid: real-time performance in high resolution nonlinear image processing. In: Gasteratos, A., Vincze, M., Tsotsos, J.K. (eds.) ICVS 2008. LNCS, vol. 5008, pp. 141–150. Springer, Heidelberg (2008)
16. Gwosdek, P., Zimmer, H., Grewenig, S., Bruhn, A., Weickert, J.: A highly efficient GPU implementation for variational optic flow based on the Euler-Lagrange framework. In: Kutulakos, K.N. (ed.) ECCV 2010 Workshops, Part II. LNCS, vol. 6554, pp. 372–383. Springer, Heidelberg (2012)
17. Zimmer, H., Bruhn, A., Weickert, J.: Optic flow in harmony. Int. J. Comput. vis. **93**(3), 368–388 (2011)
18. Wang, X., Aoki, T.: Multi-GPU performance of incompressible flow computation by lattice boltzmann method on GPU cluster. Parallel Comput. **37**, 512–535 (2011)
19. Gradl, T., Rüde, U.: High performance multigrid in current large scale parallel computers. In: 9th Workshop on Parallel Systems and Algorithms (PASA), vol. 124, pp. 37–45 (2008)
20. Gradl, T., Freundl, C., Köstler, H., Rüde, U.: Scalable multigrid. In: High Performance Computing in Science and Engineering, Garching/Munich 2007, pp. 475–483 (2009)
21. Bergen, B., Gradl, T., Hülsemann, F., Rüde, U.: A massively parallel multigrid method for finite elements. Comput. Sci. Eng. **8**(6), 56–62 (2006)
22. Köstler, H., Stürmer, M., Pohl, T.: Performance engineering to achieve real-time high dynamic range imaging. J. Real-Time Image Proc., pp. 1–13 (2013)
23. Gmeiner, B., Köstler, H., Stürmer, M., Rüde, U.: Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters. Practice and Experience, Concurrency and Computation (2012)
24. Kazhdan, M., Hoppe, H.: Streaming multigrid for gradient-domain operations on large images. ACM Trans. Graph. (TOG) **27**, 21 (2008). (ACM Press, New York)
25. Bartuschat, D., Ritter, D., Rüde, U.: Parallel multigrid for electrokinetic simulation in particle-fluid flows. In: 2012 International Conference on High Performance Computing and Simulation (HPCS), pp. 374–380. IEEE (2012)
26. Köstler, H., Ritter, D., Feichtinger, C.: A geometric multigrid solver on GPU clusters. In: Yuen, D.A., Wang, L., Chi, X., Johnsson, L., Ge, W., Shi, Y. (eds.) GPU Solutions to Multi-scale Problems in Science and Engineering. Lecture Notes in Earth System Sciences, pp. 407–422. Springer, Heidelberg (2013)
27. Brandt, A.: Multi-level adaptive solutions to boundary-value problems. Math. Comput. **31**(138), 333–390 (1977)
28. Hackbusch, W.: Multi-Grid Methods and Applications. Springer, Heidelberg (1985)
29. Briggs, W., Henson, V., McCormick, S.: A Multigrid Tutorial, 2nd edn. Society for Industrial and Applied Mathematics (SIAM), Philadelphia (2000)
30. Trottenberg, U., Oosterlee, C., Schüller, A.: Multigrid. Academic Press, San Diego (2001)
31. Douglas, C., Hu, J., Kowarschik, M., Rüde, U., Weiß, C.: Cache optimization for structured and unstructured grid multigrid. Electron. Trans. Numer. Anal. (ETNA) **10**, 21–40 (2000)

32. Hülsemann, F., Kowarschik, M., Mohr, M., Rüde, U.: Parallel geometric multigrid. In: Bruaset, A., Tveito, A. (eds.) Numerical Solution of Partial Differential Equations on Parallel Computers. Lecture Notes in Computational Science and Engineering, vol. 51, pp. 165–208. Springer, Heidelberg (2005)
33. Stürmer, M., Wellein, G., Hager, G., Köstler, H., Rüde, U.: Challenges and potentials of emerging multicore architectures. In: Wagner, S., Steinmetz, M., Bode, A., Brehm, M., eds.: High Performance Computing in Science and Engineering, Garching/Munich 2007, LRZ, KONWIHR, pp. 551–566. Springer, Heidelberg (2008)
34. Shewchuk, J.: An introduction to the conjugate gradient method without the agonizing pain (1994)
35. Feichtinger, C., Donath, S., Köstler, H., Götz, J., Rüde, U.: WaLBerla: HPC software design for computational engineering simulations. J. Comput. Sci. **2**(2), 105–112 (2011)
36. Donath, S., Feichtinger, Ch., Pohl, T., Götz, J., Rüde, U.: Localized parallel algorithm for bubble coalescence in free surface lattice-boltzmann method. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 735–746. Springer, Heidelberg (2009)
37. Götz, J., Iglberger, K., Feichtinger, C., Donath, S., Rüde, U.: Coupling multibody dynamics and computational fluid dynamics on 8192 processor cores. Parallel Comput. **36**(2–3), 142–151 (2010)
38. Dünweg, B., Schiller, U., Ladd, A.J.C.: Statistical mechanics of the fluctuating lattice boltzmann equation. Phys. Rev. E **76**(3), 036704 (2007)
39. Feichtinger, C., Habich, J., Köstler, H., Hager, G., Rüde, U., Wellein, G.: A flexible patch-based lattice boltzmann parallelization approach for heterogeneous GPU-CPU clusters. J. Parallel Comput. **37**(9), 536–549 (2011)