Chapter 26
A Geometric Multigrid Solver on GPU Clusters

Harald Koestler, Daniel Ritter and Christian Feichtinger

Abstract Recently, more and more GPU HPC clusters are installed and thus 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. 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 object-oriented software and performance engineering concepts necessary to integrate efficient compute kernels into our WaLBerla framework and show that a large fraction of the high computational performance offered by current heterogeneous HPC clusters can be sustained for geometric multigrid algorithms.

Keywords MPI parallelization · GPGPU · CUDA · Multigrid solver

26.1 Introduction

The multi-disciplinary field of computational science and engineering (CSE) deals with large scale computer simulations and optimization of mathematical models. CSE is used successfully, e.g. by aerospace, automotive, and processing industries, as well as in medical technology. In order to obtain physically meaningful results, many of these simulation tasks can only be done on HPC clusters due to the high memory and compute power requirements. Therefore, software development in CSE is dominated by the need for efficient and scalable codes on current compute clusters.

H. Koestler (✉) · D. Ritter · C. Feichtinger
System Simulation Group, University of Erlangen-Nuremberg,Erlangen, Germany
e-mail: harald.koestler@informatik.uni-erlangen.de
http://www10.informatik.uni-erlangen.de

Graphics processing units (GPUs) typically offer hundreds of specialized compute units operating on dedicated memory. In this way they reach outstanding compute and memory performance and are more and more used for compute-intensive applications, often called general purpose programming on graphics processing units (GPGPU). 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. Recently, GPUs are more and more used to build up heterogeneous multi-GPU HPC clusters. In the current Top 500 list[1] of the fastest machines world-wide there are three of these clusters amongst the Top 5.

In order to achieve good performance on these GPU 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 GPGPU. 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) (NVIDIA 2010). Since we are exclusively working on NVIDIA GPUs in this article we have done our implementations in CUDA. 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, but to achieve good performance the implementation has to be adapted to the specific features of the hardware. 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 (Ohshima et al. 2010) and PyCUDA (Klöckner et al. 2009) or easy to use frameworks.

Our contributions in this article are that we first discuss the concepts necessary to integrate efficient GPU compute kernels into our software framework called WaLBerla and second that we show scaling results for an exemplary multigrid solver on multi-GPU clusters.

Various other implementations of different multigrid algorithms on GPU exist (e.g. Bolz et al. 2003; Goddeke et al. 2008; Haase et al. 2010), and multigrid is also incorporated in software packages like OpenCurrent (Cohen 2011) or PETSc (Balay et al. 2009).

The paper is organized as follows: In Sect. 26.2 we briefly describe the multigrid algorithm and its parallelization on GPUs. Section 26.3 summarizes the MPI-parallel WaLBerla framework that easily enables us to extend our code to multi-GPUs, and in Sect. 26.4 we present performance results for different CPU and GPU platforms before concluding the paper in Sect. 26.5.

[1] http://www.top500.org

[2] see http://www.khronos.org/opencl/

26.2 Parallel Multigrid

26.2.1 Multigrid Algorithm

Multigrid is not a single algorithm, but a general approach to solve problems by using several levels or resolutions (Brandt 1977; Hackbusch 1985). 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 Conjugate Gradients (CG) is that multigrid can reach an asymptotically optimal complexity of $\mathcal{O}(N)$, where N is the number of unknowns in the system. For good introductions and a comprehensive overview on multigrid methods, we, e.g., refer to Briggs et al. (2000) and Trottenberg et al. (2001), for details on efficient multigrid implementations see Douglas et al. (2000), Hülsemann et al. (2005), Stürmer et al. (2008) and Köstler (2008).

We assume that we want to solve the PDE

$$-\nabla c \nabla \boldsymbol{u} + \alpha u = f \quad \text{in } \Omega \tag{26.1a}$$

$$\langle \nabla \boldsymbol{u}, \boldsymbol{n} \rangle = 0 \quad \text{on } \partial\Omega \tag{26.1b}$$

with $\alpha > 0$, smoothly varying or constant coefficients $c : \mathbb{R}^3 \to \mathbb{R}^+$, solution $\boldsymbol{u} : \mathbb{R}^3 \to \mathbb{R}$, right hand side (RHS) $f : \mathbb{R}^3 \to \mathbb{R}$, and (natural) Neumann boundary conditions on a rectangular domain $\Omega \subset \mathbb{R}^3$. Equation 26.1 is discretized by finite volumes on a structured grid. This results in a linear system

$$A^h u^h = f^h, \quad \sum_{j \in \Omega^h} a^h_{ij} u^h_j = f^h_i, \quad i \in \Omega^h \tag{26.2}$$

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. 26.2 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

$$A^h e^h = r^h. \tag{26.3}$$

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 afterwards. *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 now 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. 26.1 is first 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 many smoothing 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 cell-based multigrid solver we use the following components:

- A RBGS 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 simple averaging over the neighboring cells.
- We apply a nearest neighbor interpolation operator P for the error.
- The coarse grid problem is solved by a sufficient number of RBGS steps.
- The discretization of the Laplacian was done via the standard 7-point stencil (case $c = 1$ in Eq. 26.1).
- For varying c we apply Galerkin coarsening with a variable 7-point stencil on each grid level.

26.2.2 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 RBGS kernel as an example and discuss it in more detail. Algorithm 25.2 shows the source code of the CUDA kernel. It is called from Algorithm 25.3.

The kernel can handle arbitrary variable seven-point stencils. The `GET3D_tex` and `GET3D_ST_tex` functions are macros that provide access to the solution resp. stencil field that is stored in global or texture 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. Note that the outer if statement to check if the point is not a boundary point can be dropped on the new Fermi GPUs since they are much less sensitive to data alignment than older GPUs.

Algorithm 2 Red-black Gauss-Seidel smoother kernel in CUDA.

```
__global__ void kr_RBGS(double * stencil, double* solution, double* 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;

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

    double fak = 1./GET3D_ST_tex(tex_stencil,x,y,z,C);
    double v = 0.0;
    v += GET3D_tex(tex_solution,x,y+1,z)*GET3D_ST_tex(tex_stencil,x,y,z,N);
    v += GET3D_tex(tex_solution,x,y-1,z)*GET3D_ST_tex(tex_stencil,x,y,z,S);
    v += GET3D_tex(tex_solution,x-1,y,z)*GET3D_ST_tex(tex_stencil,x,y,z,W);
    v += GET3D_tex(tex_solution,x+1,y,z)*GET3D_ST_tex(tex_stencil,x,y,z,E);
    v += GET3D_tex(tex_solution,x,y,z+1)*GET3D_ST_tex(tex_stencil,x,y,z,T);
    v += GET3D_tex(tex_solution,x,y,z-1)*GET3D_ST_tex(tex_stencil,x,y,z,B);

    double new_val = (GET3D(rhs,x,y,z)- v)/fak;

    if (((x+y+z)&1) == red_black)
      GET3D(solution,x,y,z) = new_val;
  }
}
```

The distributed memory parallelization is simply done by decomposing the finest 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. In case of multi-GPU processing, the function calling this kernel (shown in Algorithm 25.3) has to handle the ghost cells. The solution values at the borders of the sub-grids have to be initialized with the values that were already communicated via MPI. This transfer from the MPI buffer in main memory to the memory of the GPU is the only part of communication that is not done implicitly by the WaLBerla framework. After that, the stencil, solution and right-hand side values are put to texture arrays to have a more efficient read access to them later. This is not necessary for the new NVIDIA Fermi GPUs, since they offer a built-in cache for the global GPU memory. Now, the actual red and the black sweep are done. After the sweeps, the Neumann boundary conditions are set by copying the border values to a ghost layer. Finally, the values to be communicated are transferred again to the MPI buffers, before the mapping of the texture objects is released.

Algorithm 3 Cuda kernel wrapper

```
void RBGSSweepVarGPU(double * st, double* sol,
                         double* rhs, double** buffers,
                         const Uint xSize, const Uint ySize,
                         const Uint zSize, const Uint * blockID,
                         const Uint * numBlocks)
{

  dim3 dimblock(xSize,1);
  dim3 dimgrid(ySize,zSize);

  //COPY MPI_BUFFER -> GPU
  SetBuffers(sol,buffers,xSize,ySize,zSize,blockID, numBlocks);

  LoadTextureFrom1DArray(st, xSize*ySize*zSize*StenCellSize, tex_stencil);
  LoadTextureFrom1DArray(sol, xSize*ySize*zSize, tex_solution);
  LoadTextureFrom1DArray(rhs, xSize*ySize*zSize, tex_rhs);

  kr_RBGS<<<dimgrid, dimblock>>>(st, sol, rhs, xSize, ySize, zSize,0.);
  kr_RBGS<<<dimgrid, dimblock>>>(st, sol, rhs, xSize, ySize, zSize,1.);

  Treatboundary(sol,xSize, ySize, zSize, blockID, numBlocks);

  //COPY GPU -> MPI_BUFFER
  CopyBuffers(sol,buffers,xSize,ySize,zSize,blockID, numBlocks);

  UnloadTextureFrom1DArray(tex_stencil);
  UnloadTextureFrom1DArray(tex_solution);
  UnloadTextureFrom1DArray(tex_rhs);

}
```

26.3 WaLBerla

WaLBerla is a massively parallel multi-physics software framework developed for HPC applications on block-structured domains (Feichtinger et al. 2010). It has been successfully used in many simulation tasks ranging from free surface flows (Donath et al. 2009) to particulate flows (Götz et al. 2010) and fluctuating lattice Boltzmann (Dünweg et al. 2007) for nano fluids.

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 (Feichtinger et al. 2010).

26.3.1 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. 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.

When the simulation runs, all tasks are broken down into several basic steps, so-called *Sweeps*. A Sweep consists of two parts: 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.

26.3.2 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.

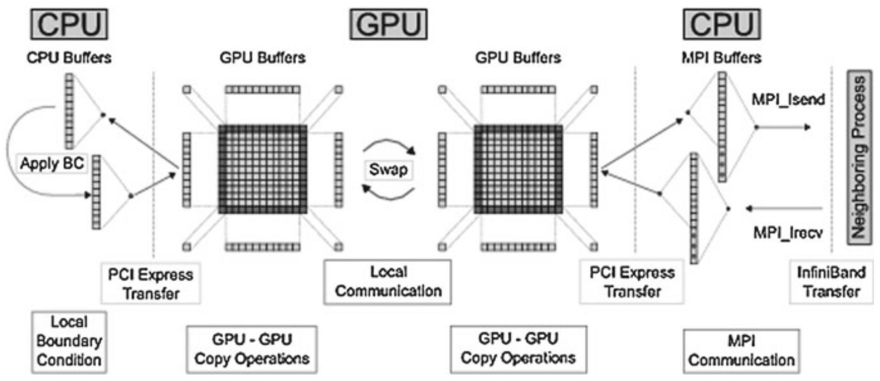

Fig. 26.1 Communication concept within WaLBerla (Feichtinger et al. 2010)

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 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.

26.3.3 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. 26.1.

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 the function *cudaMemcpy* is used to 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.

Table 26.1 Technical hardware specifications

	Xeon 5550	Tesla M1060	Tesla C2070
Processor frequency	2.66 GHz	1.3 GHz	1.15 GHz
Memory frequency	1.3 GHz	800 MHz	1.5 GHz
Memory size	12 GB	4 GB	6 GB
# Streaming units/cores	4	240	448
Floating-point performance (SP)	85.1 GFLOP/s	933 GFLOP/s	1030 GFLOP/s
Floating-point performance (DP)	42.6 GFLOP/s	78 GFLOP/s	515 GFLOP/s
Memory bandwidth	32 GB/s	102 GB/s	144 GB/s

26.4 Performance Results

The main focus within this article is put on parallel efficiency of our multigrid implementation on multi-GPU clusters. As a baseline we also evaluate the single GPU runtime and identify the performance bottlenecks.

26.4.1 Platforms for Tests

The numerical tests were performed on three different platforms provided by our local computing center,[3] an Intel Core i7 CPU (Xeon 5550), an NVIDIA G80 (Tesla M1060) and an NVIDIA Fermi (Tesla C2070) platform. The detailed hardware specifications are depicted in Table 26.1.

The cluster for our test is the *TinyGPU* cluster of the RRZE. It consists of 8 dual-socket nodes, hosting two Xeon 5500 processors and two Tesla M1060 boards. The nodes are connected via Infiniband. Additionally, we had access to one of those nodes with two Tesla C2070 GPUs instead of the M1060.

All numerical tests run with double floating-point precision.

26.4.2 Scaling Experiments

In the following, we show how the runtime code behaves with increasing problem size on one or several compute nodes. Baseline is the performance on one node, i.e. two Teslas resp. two Xeons.

We distinguish two types of experiments: *Weak scaling* relates to experiments were the problem size is increased linearly with the number of involved devices, 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

[3] http://www.rrze.de

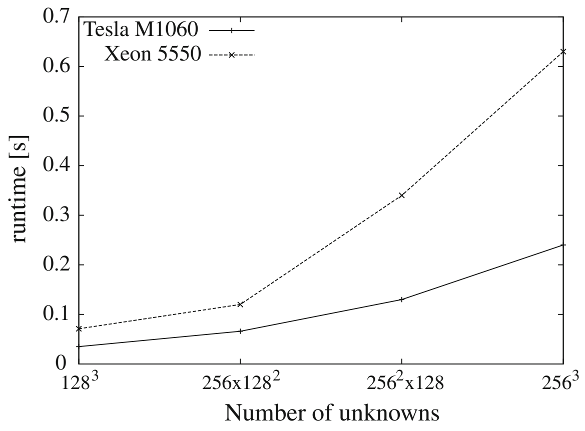

Fig. 26.2 Single node (i.e. two Teslas resp. two Xeons) performance of one multigrid V(2, 2)-cycle

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. We measure the runtime of one V(2,2)-cycle (i.e. a V-cycle with 2 RBGS iterations for pre- and postsmoothing each) on four grid levels with parameters from Sect. 26.2. On the coarsest grid 50 RBGS steps are performed to obtain a solution.

Single-node performance Here, the problem sizes vary between $128^3 = 2{,}097{,}152$ and $256^3 = 16{,}777{,}216$ unknowns and the runtimes for one V(2, 2)-cycle on Xeon 5550 and Tesla M1060 machines are depicted in Fig. 26.2.

Since the performance of the multigrid algorithm is memory-bandwidth bounded and there is roughly a factor of three in theoretical peak memory bandwidth between CPU and GPU we expect the same factor in the runtime of both platforms. Indeed, the GPU shows a speedup factor of about three for larger problem sizes. For smaller problems this factor shrinks down due to several reasons: first, the GPU overhead e.g. for CUDA kernel calls becomes visible and there is not enough work to be done in parallel, especially on the coarse grids. Furthermore, the CPU can profit from its big caches.

Weak scaling Figure 26.3 shows the weak scaling behavior of the code for problem size 256^3. On the Xeon—having eight physical cores per node on two sockets—tests are performed with four and eight MPI instances per node. We did not pin the MPI processes to fixed cores and thus the runtimes slightly vary between 0.6 and 0.9 seconds for one V(2, 2)-cycle. In average the results do not differ much for four and eight MPI instances per node. In case of the Tesla we have only two MPI instances per node and the runtime is quite stable. However, it increases by approximately 20 % for eight nodes compared to the baseline performance. This slightly worse scaling factor on GPU is mainly due to the effect of additional intra-node memory transfers of ghost layers between CPU and GPU.

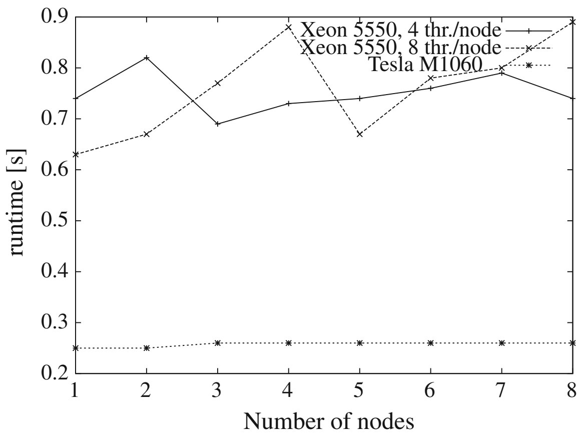

Fig. 26.3 Weak scaling behavior per computer node of one multigrid V(2,2)-cycle

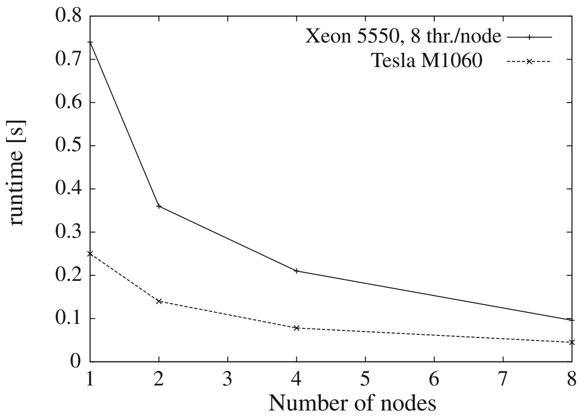

Fig. 26.4 Strong scaling behavior of one multigrid V(2,2)-cycle with 256^3 unknowns

Table 26.2 Speedup factors for strong scaling experiment in Fig. 26.4

#Nodes	2	4	8
Tesla M1060	1.79	3.21	5.56
Xeon 5550	2.06	3.52	7.71

Strong scaling Next, we scale the number of involved processing units, but leave the total problem size, i.e. the number of unknowns, constant. Figure 26.4 shows the runtimes on the Xeon and the Tesla for $2 \cdot 256^3$ unknowns are shown for one to eight nodes of the cluster and Table 26.2 the corresponding relative speedup factors.

For the Xeon tests the parallel efficiency is relatively high: The speedup factor on 8 nodes of 7.71 stays only slightly below the ideal one, which is linear with the

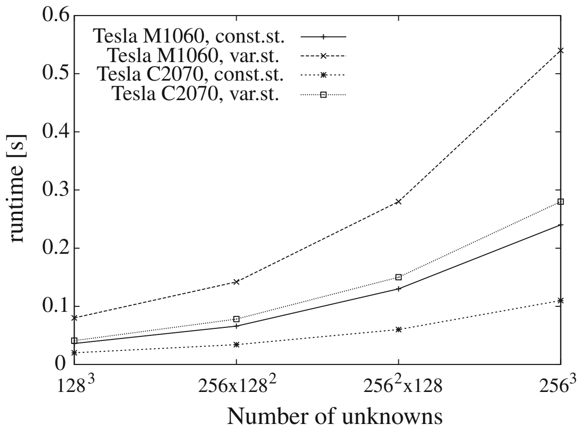

Fig. 26.5 Performance comparison of constant and variable 7-point stencil within one multigrid V$(2,2)$-cycle

number of nodes and thus 8. For two nodes even super-linear scaling is reached with the CPU code, this could be caused for example by cache effects.

The speedup on the Tesla is just 5.56, which is a result of different factors: on the one hand the problems for small size mentioned when discussing the single-node performance and on the other hand the communication overhead addressed within the weak scaling experiments.

26.4.3 CUDA Compute Capability 1.3 Versus 2.0

Next we evaluate the performance of a newer NVIDIA Fermi Tesla C2070 graphics board that implements CUDA compute capability 2.0. Its technical specifications are listed in Table 26.1 and the difference to the previous generation (CUDA compute capability 1.3) is, beneath a higher bandwidth, a tremendous increase in double-precision floating-point performance and real caches. Furthermore, it is easier to program: Alignment requirements are weakened and C++ support is enabled.

We now test also variable 7-point stencils, i.e. c from Eq. 26.1 is no more constant within the domain Ω. In this case we have in contrast to the constant coefficient stencil to store additionally the stencil at each grid point. We then use the RBGS smoother as sketched in Algorithm 25.2. The runtimes for constant and variable stencils are shown in Fig. 26.5 on both GPU platforms. With the constant stencil the C2070 is 2.2 times faster than the M1060, whereas for the variable stencil the speedup factor is 1.9. The performance gain cannot be a single effect of the increased memory bandwidth. In addition to that we also profit from the new features of the Fermi, especially from the cache and the weaker alignment requirements.

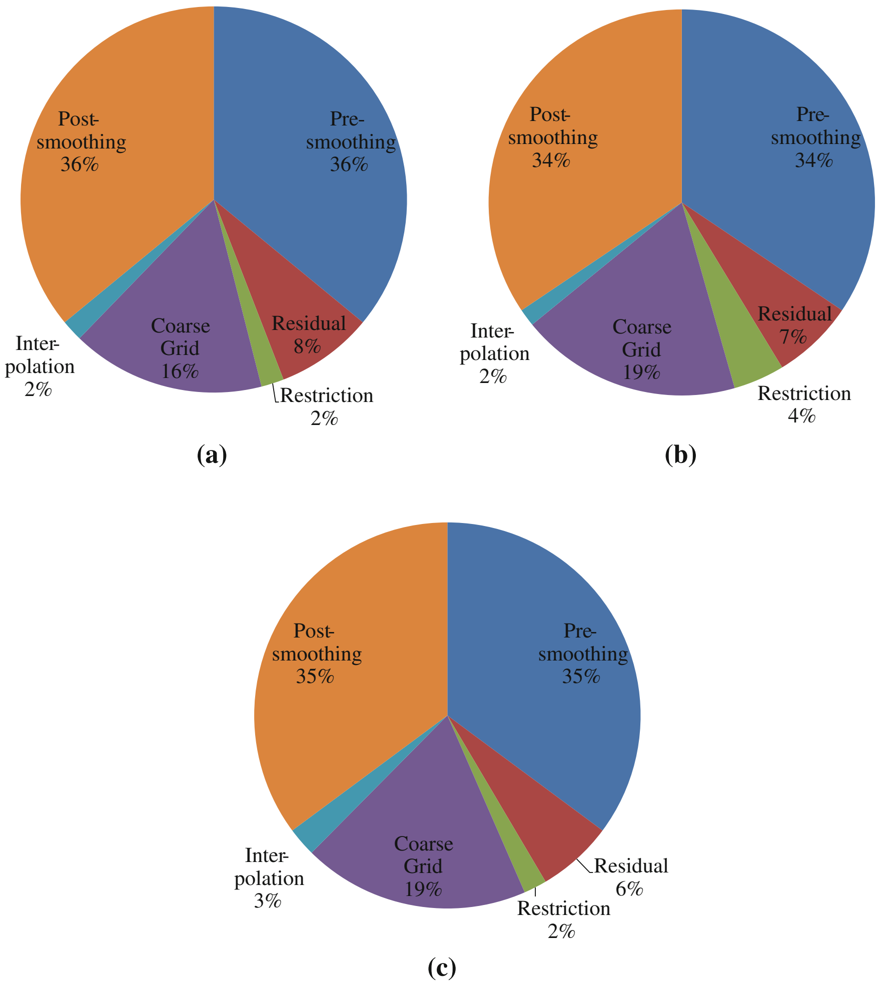

Fig. 26.6 Runtime percentage for different components on **a** one and **b** eight Tesla M1060 and **c** one Tesla C2070 (problem size 256^3, constant stencil)

26.4.4 Runtime of Components

In this section we try to give more insight in the runtime behavior of the different parts of the multigrid solver in order to identify the performance bottlenecks.

Therefore, Fig. 26.6 shows the portion of runtime spent in different components of a V(2, 2)-cycle on one resp. eight Tesla M1060 and one Tesla C2070. Since the overall performance of the multigrid solver is bounded by the memory bandwidth it is not astonishing that in each case smoothing on the finest grid takes around 70 % of the runtime. The problem size shrinks by a factor of 8 for each grid level, thus

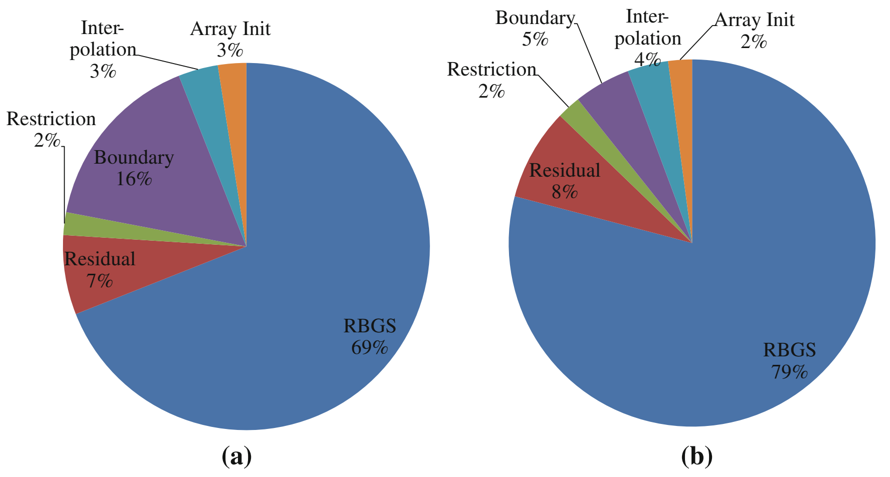

Fig. 26.7 Runtime percentage of different CUDA kernels on one Tesla C2070 for problem size **a** 128^3 and **b** 256^3 (constant stencil)

one expects the coarse grid (this includes all the previous components on all three coarser grids plus solving the problem on the coarsest grid with 50 RBGS iterations) to require about 1/8 of the runtime. This lies a little bit higher especially for the Tesla C2070, because the smaller sizes are not very efficient on the GPU as seen before. In the multi-GPU case the cost for the communication between the single processes is an important issue. In contrast to the calculations that are simply distributed, the communication has to be done extra when switching from sequential to parallel code. Therefore, it has to be ensured that this extra work does only consume little time and does not dominate the whole runtime. Within the multigrid algorithm the communication is part of pre- and postsmoothing, where the solution ghost layer has to be communicated to the neighboring processes, and of the restriction, where the residual ghost layer has to be distributed. In Fig. 26.6 the communication effects are an increase of the restriction runtime including the residual ghost layer transfer and the increase of the portion spent on the coarser grids, because of the worse ratio between local points and ghost layer points there and the bigger influence of latencies in case of smaller transfers. Altogether, runtime distribution in the one-GPU and multi-GPU case looks quite similar. To explain this, note that the difference in runtimes on one and eight Tesla M1060 is 37 ms or 13 % of the total runtime (for 256^3 unknowns per GPU and a constant stencil). This means, at least for that number of GPUs, the communication does not have a dominating effect on the runtime.

To provide another point of view we also measured the performance of our CUDA kernels for the same setting as above on one Tesla C2070 with the profiler *computeprof* provided by NVIDIA. The results are summarized in Fig. 26.7. In contrast to the previous measurements these do not contain the overhead from thread creation and cleanup. Additionally, they show more details since some of the components in Fig. 26.6 call more than one CUDA kernel. In total, we observe that the RBGS kernel dominates the calculation. For 256^3 unknowns it requires almost 80 % of the runtime and for 128^3 unknowns still almost 70 %. For the small problem size the boundary treatment, which is included in the smoothing in Fig. 26.6, plays quite a role with 15 % of the total runtime.

26.5 Conclusions and Future Work

We have implemented a geometric multigrid solver for Eq. 26.1 on GPU and integrated it into the WaLBerla framework. We observed that the runtimes decrease on the current NVIDIA Fermi architecture due to its new hardware features like an incorporated memory cache. The speedup factor between CPU and GPU implementations of our multigrid algorithm corresponds roughly to the ratio of their memory bandwidths in the single-node case. Our experiments on a small compute cluster showed good scaling behavior for CPUs and slightly worse for GPUs.

Next steps would be a performance optimization of our code and a comparison of CUDA and OpenCL. One obvious improvement would be to use an optimized data layout, by splitting the red and black grid points into two separate fields. Finally, we currently investigate further applications for the parallel multigrid solver within the WaLBerla framework.

References

Balay S, Buschelman K, Gropp WD, Kaushik D, Knepley MG, McInnes LC, Smith BF, Zhang H (2009) PETSc web page. http://www.mcs.anl.gov/petsc

Bolz J, Farmer I, Grinspun E, Schröder P (2003) Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In: ACM SIGGRAPH 2003 papers, pp 917–924

Brandt A (1977) Multi-level adaptive solutions to boundary-value problems. Math. Comput. 31(138):333–390

Briggs W, Henson V, McCormick S (2000) A multigrid tutorial, 2nd edn. Society for Industrial and Applied Mathematics (SIAM), Philadelphia

Cohen J (2011) OpenCurrent. NVIDIA research. http://code.google.com/p/opencurrent/

Donath S, Feichtinger C, Pohl T, Götz J, Rüde U, (2009) Localized parallel algorithm for bubble coalescence in free surface lattice-Boltzmann method. In: Sips H, Epema D, Lin H-X (eds) Euro-Par, (2009) Lecture notes in computer science, vol 5704. Springer, Berlin, pp 735–746

Douglas C, Hu J, Kowarschik M, Rüde U, Weiß C (2000) Cache optimization for structured and unstructured grid multigrid. Elect Trans Numer Anal 10:21–40

Dünweg B, Schiller U, Ladd AJC (Sep 2007) Statistical Mechanics of the Fluctuating Lattice Boltzmann Equation. Phys. Rev. E 76(3):036704

Feichtinger C, Donath S, Köstler H, Götz J, Rüde U (2010) WaLBerla: HPC software design for computational engineering simulations. J Comput Sci (submitted)

Feichtinger C, Habich J, Köstler H, Hager G, Rüde U, Wellein G (2010) A flexible patch-based lattice Boltzmann parallelization approach for heterogeneous GPU-CPU clusters. J Parallel Comput. Arxiv, preprint arXiv:1007.1388 (submitted)

Goddeke D, Strzodka R, Mohd-Yusof J, McCormick P, Wobker H, Becker C, Turek S (2008) Using GPUs to improve multigrid solver performance on a cluster. Int J Comput Sci Eng 4(1):36–55
Götz J, Iglberger K, Feichtinger C, Donath S, Rüde U (2010) Coupling multibody dynamics and computational fluid dynamics on 8192 processor cores. Parallel Comput 36(2–3):142–151
Haase G, Liebmann M, Douglas C, Plank G (2010) A parallel algebraic multigrid solver on graphics processing units. In: Zhang W et al (eds) High performance computing and applications. Springer, Berlin, pp 38–47
Hackbusch W (1985) Multi-grid methods and applications. Springer, Berlin
Hülsemann F, Kowarschik M, Mohr M, Rüde U (2005) 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. Springer, Berlin, pp 165–208
Klöckner A, Pinto N, Lee Y, Catanzaro B, Ivanov P, Fasih A (2009) PyCUDA: GPU run-time code generation for high-performance computing. Arxiv preprint arXiv 911. http://mathema.tician.de/software/pycuda
Köstler H (2008) A multigrid framework for variational approaches in medical image processing and computer vision. Verlag Dr, Hut, München
NVIDIA Cuda Programming Guide 3.2 (2010). http://developer.nvidia.com/object/cuda_3_2_downloads.html
Ohshima S, Hirasawa S, Honda H (2010) OMPCUDA: OpenMP execution framework for CUDA based on omni OpenMP compiler. In: Beyond loop level parallelism in OpenMP: accelerators, tasking and more, pp 161–173
Stürmer M, Wellein G, Hager G, Köstler H, Rüde U, (2008) 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. Springer, Berlin, pp 551–566
Trottenberg U, Oosterlee C, Schüller A (2001) Multigrid. Academic Press, San Diego