2012 11th International Symposium on Parallel and Distributed Computing

Steps Towards GPU Accelerated Aggregation AMG

Maximilian Emans
Johann Radon Institute for Computational and Applied Mathematics and IMCC GmbH
both 4040 Linz, Austria
Email: maximilian.emans@ricam.oeaw.ac.at

Manfred Liebmann
Institute for Mathematics and Scientific Computing
University of Graz
8010 Graz, Austria
Email: manfred.liebmann@uni-graz.at

Branislav Basara
Chief CFD developer
AVL GmbH
Advanced Simulation Technology
8020 Graz, Austria
Email: branislav.basara@avl.com

***Abstract*—We present an implementation of AMG with simple aggregation techniques on multiple GPUs. It supports the parallel matrix representations typically used for finite volume discretisation. We employ the ICRS sparse matrix format and the asynchronous exchange mechanism of MPI on CPUs that has been modified to make it suitable for the GPU coprocessors. We show that the solution phase of the standard v-cycle AMG with simple aggregation is accelerated by a factor of up to 12. The solution phase of the more advanced Krylov-accelerated AMG runs faster by a factor of up to 7 on Nvidia TESLA C2070 compared to calculation on Intel X5650 CPUs.**

***Keywords*-algebraic multigrid; GPGPU; finite volumes**

I. Introduction

Various physical phenomena are modelled by partial differential equations. Simulations based on such models usually require the solution of large sparse linear systems. In many cases the performance of the solver for these systems essentially determines the run-time of the simulation on the computing system. Large-scale simulations, i.e. simulations on parallel computers with execution times in the range of hours or days, are not only done in the scientific sector, but also play a role in the industrial development process. The demand for faster computations is on the one hand driven by the need to obtain the results faster and to speed up e.g. an industrial development process in this manner. On the other hand, the limited availability of hardware resources requires that the available hardware is used as efficiently as possible.

Most of the major computational tasks within a numerical simulation are done on conventional compute nodes equipped with CPUs. Recent developments in both, hardware design and software tools made it possible to exploit the large computational power of graphical processing units (GPU) for numerical calculations. Implementations of solver algorithms for linear systems, e.g. Krüger and Westermann [1], and even entire numerical simulations, e.g. Goodnight et al. [2], were pioneered some time ago. However, the restrictions in particular with respect to the programming environment and the hardware were severe at that time in a way that these approaches were limited to structured meshes and two-dimensional problems. Although multigrid algorithms could be implemented, the practical benefit of these implementations was rather small. But many of the obstacles have been overcome in recent time. It is now possible to solve large linear systems on GPU-accelerated hardware resulting in solvers with very attractive performance in terms of computing time. Parallel solvers of large-scale linear problems using multigrid algorithms on clusters with multiple GPUs are reported e.g. by Haase et al. [3] and Göddecke et al. [4].

In numerous applications, the solution of the linear systems is needed within an iterative scheme that is necessary to handle the non-linearity or the coupling of the physical variables. An accurate solution of the linear system is then usually not required, instead the reduction of some norm of the residual by only two or three orders of magnitude is sufficient. Other AMG algorithms as the ones in the mentioned publications, namely algorithms with a very cheap setup, are preferred in such situations. This is the case for typical SIMPLE-based algorithms in general purpose CFD software. In this context, the approximate solution of the pressure-correction equation is the computationally most expensive part. For this equation, simple aggregation AMG variants have been shown to be well suited, see Emans [5]. Moreover, the framework of the finite volume discretisation usually prescribes a domain decomposition with a certain overlap as parallelisation approach. This requires a parallelisation of the computational kernels that is different from that of the solvers presented by Haase et al. [3] and Göddecke et al. [4] who use parallel data structures with shared nodes that are typically used in finite element schemes.

In this contribution we demonstrate that on modern GPUs algorithms for the mentioned linear solvers can be significantly accelerated. This is shown not only to be true for the use of a single GPU, but also for calculations on multiple GPUs. This makes the solver well suited for problem sizes in the range of several million unknowns or more. Furthermore, we show that not only simple v-cycle AMG algorithms, but also advanced k-cycle AMG algorithms, i.e. algorithms with Krylov acceleration on each level, can take advantage of the GPU acceleration.

978-0-7695-4805-0/12 $26.00 © 2012 IEEE
DOI 10.1109/ISPDC.2012.19

II. Aggregation AMG algorithms

The AMG algorithm is here applied as a preconditioner of a Krylov solver for a symmetric positive definite or semi-definite system. Let us denote this system as

$$A\vec{x} = \vec{b} \tag{1}$$

where $A \in \mathbb{R}^{n\times n}$, $\vec{b} \in \mathbb{R}^n$ is some right-hand side vector, $\vec{x} \in \mathbb{R}^n$ the solution, and n is the number of unknowns. We assume that the matrix is given in Compressed Row Storage (CRS) format as e.g. described by Falgout et al. [6]. For parallel computations, a domain decomposition is used to assign a certain set of nodes to each of the parallel processes. The influences of the values associated with nodes on neighbouring processes are handled through a buffer layer, i.e. those values are calculated by the process they are associated with, but they are exchanged each time they are needed by the neighbouring processes.

One of the simplest AMG schemes, the v-cycle, is shown in Algorithm 1. This scheme as well as any other AMG scheme requires the definition of a grid hierarchy with L levels. The matrices representing the problem on grid level l are $A_l \in \mathbb{R}^{n_l\times n_l}$ $(l = 1, ..., L)$ with system size n_l where $n_{l+1} < n_l$ holds for $l = 1, ..., L-1$ and $A_1 = A$ as well as $n_1 = n$. As it is common practice in algebraic multigrid, the coarse-grid operators, starting with the finest grid, are defined recursively by

$$A_{l+1} = P_l^T A_l P_l \qquad (l = 1, ..., L-1). \tag{2}$$

where the prolongation operator P_l has to be determined for each level l while the restriction operator is defined as P_l^T. It is the choice of the coarse-grid selection scheme that determines the elements of P_l and consequently the entire grid hierarchy. The definition of the elements of P_l for all levels and the computation of the operators A_l $(l = 2, ..., L)$ are referred to as the setup phase of AMG.

A. Aggregation scheme

Simple aggregation methods have the advantage that the setup is computationally inexpensive compared to the setup of more complex aggregation methods like Smoothed Aggregation of Vaněk et al. [7] or to splitting based techniques that otherwise proved to be very efficient. Methods based on the pairwise aggregation, used by Notay [8], are simple to implement and reliable. They have been demonstrated to be appropriate for linear solvers in CFD simulation tools, see Emans [5]. With regard to GPU computing, this advantage is even more relevant: Due to the structure of the algorithm there is little hope that the setup phase can be accelerated to the same extend as the solution phase.

The prolongation operator P_l maps a vector on the coarse grid $\vec{x}_{l+1}$ to a vector on the fine grid $\vec{x}_l$:

$$\vec{x}_l = P_l\vec{x}_{l+1} \tag{3}$$

Algorithm 1 v-cycle AMG

$\vec{x}_l^{(3)}$ = v-cycle(l, $\vec{b}_l$, $\vec{x}_l^{(0)}$)
Input: level l, right-hand side $\vec{b}_l$, initial guess $\vec{x}_l^{(0)}$
Output: approximate solution $\vec{x}_l^{(3)}$

1: pre-smoothing: $\vec{x}_l^{(1)} = S_l(\vec{b}_l, A_l, \vec{x}_l^{(0)})$
2: compute residual: $\vec{r}_l = \vec{b}_l - A_l\vec{x}_l^{(1)}$
3: restriction: $\vec{r}_{l+1} = P_l^T\vec{r}_l$
4: **if** $l + 1 = L$ **then**
5: solution of coarse-grid system: $\vec{x}_{l+1} = A_{l+1}^{-1}\vec{r}_{l+1}$
6: **else**
7: recursive solution of coarse-grid system: $\vec{x}_{l+1}$ = v-cycle($l + 1$, $\vec{r}_{l+1}$, $\vec{0}$)
8: **end if**
9: prolongation of coarse-grid solution and update: $\vec{x}_l^{(2)} = \vec{x}_l^{(1)} + P_l\vec{x}_{l+1}$
10: post-smoothing: $\vec{x}_l^{(3)} = S_l(\vec{b}_l, A_l, \vec{x}_l^{(2)})$

For the definition of this operator, a pairwise aggregation considering the strength of the connection between two nodes is used. This results in prolongation operators with a particularly simple structure: They have only one non-zero element with value one in each row. The calculation of the coarse-grid operator by means of equation (2) is therefore essentially an addition of rows. The parallel version of the method restricts the aggregates to nodes belonging to the same parallel domain. We use this aggregation technique in a v-cycle as preconditioner to a standard conjugate gradient (CG) method. The algorithm is referred to as V-PA2.

Since there are only two nodes per aggregate, the coarsening of this method is quite slow, which results in high operator complexities and has computational disadvantages, see Emans [9]. By a second pass of the pairwise aggregation to the grid and the corresponding system that has been constructed by the first pairwise aggregation, aggregates of at most four nodes are obtained and a corresponding coarse-grid operator is calculated. With this scheme, the coarsening produces only half as many levels as that of V-PA2, but the convergence with the same cycle is significantly slower. However, it is well suited for the k-cycle (see section II-B). We will use this double-pairwise aggregation therefore together with the k-cycle based on flexible conjugate gradients, see Notay [10], and denote the scheme by K-PA4.

B. Cycling strategy

Most of the AMG algorithms described in the literature, e.g. Vaněk et al. [7], Henson and Yang [11], Emans [12], or Haase and Reitzinger [13], among many others, use a fixed cycling scheme. This means that the order in which the different levels of the grid hierarchy are visited, has been determined before the algorithm is started. A simple but still effective and frequently applied cycle is the v-cycle,

see Algorithm 1, in which each grid level is visited once per iteration.

A variable cycling scheme tries to adjust the effort spent on each grid level to efficiently reduce the error component on this level. There are many ways to introduce flexibility into the cycles; the most important parameter that can be chosen in a flexible manner depending on some residual-related criterion is the number of visits to the next coarser grid such that e.g. either a v-cycle or a w-cycle is effectively applied. A concrete formulation of such a flexible cycling strategy that has been shown to work extraordinarily well for fluid flow problems like the ones we discuss in this article, is the so-called k-cycle algorithm that traces back to Axelsson and Vassilevski [14] and that has been put in a compact form by Notay [8] recently. In this algorithm, the coarse-grid solution is not only obtained by applying the multi-grid recursively, but by applying a multigrid-preconditioned Krylov solver for the coarse-grid problem. The algorithm is shown in algorithm 2.

Algorithm 2 k-cycle AMG with FCG, see Notay [8]

$\vec{x}_l^{(3)}$ = k-cycle(l, $\vec{r}_l$)
Input: level l, right-hand side $\vec{r}_l$
Output: approximate solution $\vec{x}_l^{(3)}$

pre-smoothing: $\vec{x}_l^{(1)} = S_l(\vec{r}_l, A_l, \vec{x}_l^{(0)})$
restriction: $\vec{r}_{l+1} = P^T(\vec{r}_l - A_l\vec{x}_l^{(1)})$
if $l+1 = L$ **then**
  solution of coarse-grid system: $\vec{x}_{l+1} = A_{l+1}^{-1}\vec{r}_{l+1}$
else
  first recursive preconditioning: $\vec{z}^{(1)}$= k-cycle($l+1$, $\vec{r}_{l+1}$); $\vec{v}^{(1)} = A_{l+1}\vec{z}^{(1)}$
  $\rho_1 = \vec{z}^{(1)T}\vec{v}^{(1)}$; $\alpha_1 = \vec{z}^{(1)T}\vec{r}_{l+1}$
  $\tau_1 = \frac{\alpha_1}{\rho_1}$
  calculate residual of updated solution: $\vec{p} = \vec{r}_{l+1} - \tau_1 \cdot \vec{v}^{(1)}$
  if $||\vec{p}|| < \varepsilon \cdot ||\vec{r}_{l+1}||$ **then**
    update coarse-grid solution: $\vec{x}_{l+1} = \vec{x}_{l+1} + \tau_1 \cdot \vec{v}^{(1)}$
  else
    second recursive preconditioning: $\vec{z}^{(2)}$= k-cycle($l+1$, $\vec{p}$); $\vec{v}^{(2)} = A_{l+1}\vec{z}^{(2)}$
    $\gamma = \vec{z}^{(2)T}\vec{v}^{(1)}$; $\beta = \vec{z}^{(2)T}v^{(2)}$; $\alpha_2 = \vec{z}^{(2)T}\vec{p}$
    $\tau_1 = \frac{\alpha_1}{\rho_1} - \frac{\gamma}{\rho_1} \cdot \frac{\alpha_2}{\beta - \gamma^2/\rho_1}$; $\tau_2 = \frac{\alpha_2}{\beta - \gamma^2/\rho_1}$
    update coarse-grid solution: $\vec{x}_{l+1} = \vec{x}_{l+1} + \tau_1 \cdot \vec{z}^{(1)} + \tau_2 \cdot \vec{z}^{(2)}$
  end if
end if
prolongate coarse-grid solution and update: $\vec{x}_l^{(2)} = \vec{x}_l^{(1)} + P_l\vec{x}_{l+1}$
post-smoothing: $\vec{x}_l^{(3)} = S_l(\vec{r}_l, A_l, \vec{x}_l^{(2)})$

C. Other aspects of AMG

We employ a Gauß-Seidel smoother or a Jacobi smoother. The parallel implementation of the Gauß-Seidel algorithm uses the values associated with points of a neighbouring domain all from the previous iteration, i.e. in a Jacobi-like fashion. The algorithms using the Gauß-Seidel smoother are denoted by G, e.g. k-cycle with double pairwise aggregation K-P4 is then K-P4-G. The Jacobi smoother is fully parallel. The underrelaxation parameter is set to 0.67. The Jacobi smoother is denoted analogously by J.

For the solution of the coarsest system, we apply an iterative block-Jacobi technique where the blocks are the matrices associated with the points assigned to each of the processes. For this coarse-grid solver we compute in the setup an LU factorisation of these blocks; this can be done entirely in parallel since no external influences have to be considered. In the solution phase, in each iteration of this block-Jacobi scheme, the values at the boundaries are exchanged and the factorisation is used to compute the solution considering these values of the neighbouring domains. 2 iterations for the block-Jacobi solver are done for $p \leq 4$, 3 iterations for $p > 4$.

III. Implementation

A. Parallelisation of AMG by domain decomposition

The parallel implementation of the setup phase of the AMG algorithm has been discussed in detail by Emans [15]. We will focus on the solution phase in this section. In the solution phase, three groups of operations have to be considered: The first group are vector operations like additions, vector updates, and copying of data in arrays. The parallelisation of these operations does not require any communication and is straight forward to implement. The second group, scalar products and norm computations, require global communication between the processes. The third group are matrix-vector operations like the standard matrix-vector product or the smoother operations where the matrices are sparse. With regard to the computation time required for the solver, the last group dominates. It is therefore important that the parallel implementation of this type of operations is efficient. Using the message passing interface (MPI) standard, the asynchronous point-to-point communication mechanism is employed in order to overlap the overhead for the data exchange with the computational work that has to be done on the internal nodes. The algorithm and its properties have been explained in detail in Emans [9]. The domain decomposition approach assigns each node to one of the parallel processes. This is expressed by the disjoint index sets I_d that contain the indices of the nodes assigned to process d. If we consider the matrix-vector product $\vec{t} = M\vec{x}$, we first note that the vectors $\vec{t}$ and $\vec{x}$ are split into the vectors $\vec{t}_d$ and $\vec{x}_d$, respectively, where the vector with the subscript d contains the values associated

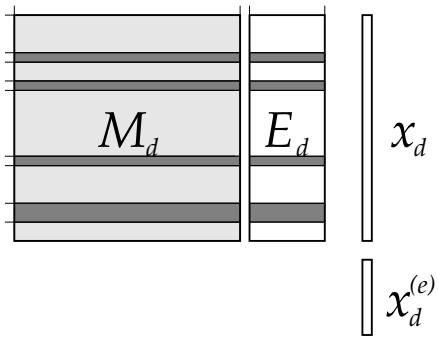

Figure 1. Matrix vector multiplication for process d, schematic: zero blocks (white rows), rows with influence from neighbouring domains (dark grey), rows without influence from neighbouring domains (light grey)

Algorithm 3 Parallel matrix-vector product $\vec{t} = M\vec{x}$, tasks on process d

1:	start sending of $\vec{x}_d^{(b)}$
2:	compute internal part of $\vec{t}_d$: $\vec{t}_d = M_d\vec{x}_d$
3:	receive values of neighbours as $\vec{x}_d^{(e)}$
4:	add contribution of neighbours: $\vec{t}_d \leftarrow \vec{t}_d + E_d\vec{x}_d^{(e)}$

with the nodes of the set I_d. The matrix M is also distributed to the parallel processes. The matrices M_d contain the rows and columns that are associated with the nodes in I_d. The matrices E_d contain the entries with column indices not in the local index set I_d. The principle is illustrated in Figure 1. For a matrix vector multiplication, process d needs to access the matrices M_d and E_d in order to evaluate $\vec{t}_d$. It is further necessary to distinguish nodes assigned to process d that are connected by the graph of the global matrix to nodes assigned to another process, from nodes that have only internal connections. Let us denote the index set of the first type of nodes as I_d^b. I_d^b is a subset of I_d. The components of $\vec{x}_d^{(b)}$ are the values associated with the nodes in I_d^b. The selection of values from $\vec{x}$ for $\vec{x}_d^{(b)}$ is represented by the operator B_d:

$$\vec{x}_d^{(b)} = B_d\vec{x}_d \tag{4}$$

The matrix E_d acts on the vector $\vec{x}_d^{(e)}$ that contains the values associated with the nodes assigned to other processes. Before any matrix-vector operation or a similar operation can be executed, the vector $\vec{x}_d^{(e)}$ on process d has to be filled with values from neighbouring processes. This exchange can be done at the same time as the internal work is done, i.e. $M_d\vec{x}_d$ is evaluated. The algorithm of this parallel computation is shown for a matrix-vector product in Algorithm 3. Note that, if process d has more than one neighbour, the components of $\vec{x}_d^{(e)}$ are assigned to different neighbours.

B. Implementation of matrix-vector operations on GPU

In the algorithms we consider for this contribution, two types of sparse matrices occur: The first type are matrices that have a few non-zero elements per row with arbitrary real value. The matrices A_l $(l = 1, ..., L-1)$ are of this type. In order to implement the matrix-vector product for these matrices on the GPU, we use the Interleaved Compressed Row Storage format (ICRS) of Liebmann [16] and the corresponding algorithm described in the same publication. The second type of matrices represents mappings, i.e. these matrices contain, aside from the zero elements, only elements with the value one. Clearly it is not economical to store all the values of the matrix elements; the data stored is only the number of elements per row, the displacement and the column indices for each element. The fill-in elements (due to the different number of elements per row) are identified by a negative column index and ignored in the matrix-vector multiplication kernel. The restriction operators P_l^T $(l = 1, ..., L-1)$ are stored in the same way. The prolongation operators P_l $(l = 1, ..., L-1)$ have exactly one non-zero element per row, thus the row count vector is also not necessary. For this third type of matrices we store only the displacement and the column indces. The matrices B_d for the extraction of the boundary values and the associated transpose matrices are stored in the latter way. The implementation of the matrix-vector products of the second and third type of matrices is a simplified version of the algorithm for the matrices of the first type.

The setup phase that is implemented conventionally on the CPU, uses the standard CRS format for the matrices; after each matrix has been defined or calculated, it is translated into the ICRS format on the CPU and then transferred to GPU memory.

C. Parallelisation using multiple GPUs

The reasons why multiple GPUs are used for a single calculation are the same as for conventional CPUs: Limited memory on a single GPU or CPU and speed-up through parallel execution of computational tasks on several GPUs or CPUs. Independent of the number of GPUs used, each GPU (device) is controlled by one process running on a CPU core (host). In our GPU implementation, the underlying parallelisation scheme, i.e. the domain decomposition, is kept the same for both. The vectors and matrices associated with the domains are stored on the GPUs. The concept of overlapping the data exchange with the arithmetic operations has to be modified: While the computational work is done on the device, the host manages the data exchange and computes $\vec{t}_d^{(b)} = E_d\vec{x}^{(e)}$, see Algorithm 4. This is efficient since $E_d\vec{x}^{(e)}$ is computationally significantly less expensive than $M_d\vec{x}_d$ and it can be overlapped with the computation of $M_d\vec{x}_d$ on the device. This implements a parallelism between computations on the host and on the device.

Algorithm 4 Parallel matrix-vector product $\vec{t} = M\vec{x}$, tasks on process d, with GPU acceleration

	host	device
1:		$\vec{x}_d^{(b)} = B_d\vec{x}$
2:	transfer of $\vec{x}_d^{(b)}$ from device to host	
3:	start sending of $\vec{x}_d^{(b)}$	launch $\vec{t}_d = M_d\vec{x}_d$
4:	receive $\vec{x}_d^{(e)}$	
5:	calculate $\vec{t}_d^{(b)} = E_d\vec{x}_d^{(e)}$	
6:		synchronise
7:	transfer of $\vec{t}_d^{(b)}$ from host to device	
8:		add: $\vec{t}_d \leftarrow \vec{t}_d + B^T\vec{t}_d^{(b)}$

IV. Benchmark

The solver algorithms described above are integrated into the program FIRE 2011, developed and distributed by AVL GmbH, Graz. As a general purpose fluid dynamics software, this program is used to solve various steady and unsteady flow problems in a wide range of technically relevant flow regimes. The available space will not permit to discuss all details of the numerical procedure the AMG solver is embedded in. The FIRE code employs the finite volume discretization method, which rests on the integral form of the general conservation law applied to the polyhedral control volumes. All dependent variables, such as momentum, pressure, density, turbulence kinetic energy, dissipation rate, and passive scalar are evaluated at the cell center. The cell-face based connectivity and interpolation practices for gradients and cell-face values are introduced to accommodate an arbitrary number of cell faces. A second-order midpoint rule is used for integral approximation and a second order linear approximation for any value at the cell-face. The convection is solved by a variety of differencing schemes (upwind, central differencing, MINMOD, and SMART), see Pržulj and Basara [17]. The rate of change is discretized by using implicit schemes, namely Euler implicit scheme and three time level implicit scheme of second order accuracy. With regard to technically relevant flows, the program provides a number of additional features: E.g., moving walls as well as sliding meshes are supported, see Basara et al. [18] and advanced turbulence models are provided, see e.g. Basara [19]. The overall solution procedure is iterative and is based on the Semi-Implicit Method for Pressure-Linked Equations algorithm (SIMPLE), see Patankar and Spalding [20]. The solution of the pressure-correction equation that reflects the mass conservation is one of the most time consuming partial tasks within this SIMPLE algorithm. The efficiency of the algorithm for the numerical solution of this system is therefore of particular importance for the performance of the whole simulation. Equally important is the robustness: Since in one iteration typically thousands of different linear systems are solved and a single failure to converge can lead to the divergence of the whole simulation, the reliability of the solver is particularly important.

The pressure-correction equation is sparse and definite or semi-definite. Essentially, it reflects the discretisation of a Poisson-problem on an unstructured mesh, see Emans [12]. The employed discretisation leads to a matrix structure where each row is associated with a cell and the only non-zero off-diagonal elements are associated with the nearest neighbours of this cell. Since the predominant shape of the cells are hexahedra, the number of non-zero elements per row is six for most of the cells.

The solver part of FIRE is coded in FORTRAN 90 and compiled by the hp FORTRAN compiler, version 11.1. The MPI library is Platform MPI, version 7.01. The GPU related code is written in CUDA and was compiled with the Nvidia nvcc compiler version 4.0, using the compute capability 1.3.

The benchmark considered is a steady state simulation of the internal flow through a water-cooling jacket of an engine block. The geometry of the computational domain is discretised by an unstructured mesh of about $5 \cdot 10^6$ cells; about 88% the cells are hexagonal. In this geometry, the incompressible Navier-Stokes equations are solved by a finite-volume based SIMPLE scheme. We apply our AMG algorithms to the pressure-correction equation only. The SIMPLE iteration is terminated after 50 iterations, i.e. in the benchmark we consider the solution of 50 linear systems with different matrices and right-hand sides. For the parallel runs, the domain decomposition is obtained by METIS [21]. The fluid is cooling water, i.e. a 50% water/glycol mixture; it flows at a rate of 2.21 kg/s into the geometry through an inlet area of $0.61 \cdot 10^{-3}\text{m}^2$ and leaves though an outlet area of $0.66 \cdot 10^{-3}\text{m}^2$. The maximum fluid velocity is 4.3 m/s. The area of the walls in total is 2.94 m^2, the volume of the cooling jacket is 1.14 m^{-3}. One of the goals of such a simulation is the prediction of the heat transfer into the cooling water. Since the fluid behaves essentially as an incompressible fluid, the temperature is treated as a passive scalar and has no influence on the flow field. Computationally the most demanding task is the prediction of the flow field.

For our benchmark two different machines were available: machine 1 with four Nvidia Tesla C2070 GPUs per node and machine 2 with two Nvidia Tesla M2050. Machine 1 is equipped with two Intel Xeon X5650 CPUs, machine 2 has two Intel E5650 CPUs. The hardware specification of the nodes is compiled in Table I. Machine 1 has four nodes that are connected through an 40 Gb/s QDR Infiniband interconnect. The operating system of both machines is a 64-bit Linux.

In the following we consider the time that is spent in the solver for the solution of the pressure-correction equation only. The time measurements are accumulated wall clock times for the solution of the 50 linear systems. We measure

Table I
HARDWARE SPECIFICATION

machine 1	
CPU	2 Intel X5650
cores	2·6
main memory, per node	96 GB
l3-cache, per CPU	2·6 MB, shared
clock frequency	2.67 GHz
memory bus	QPI, 26.7 GB/s
GPU	4 Nvidia Tesla C2070
multiprocessors	4·14
global memory, per GPU	5375 MB
memory clock rate	1.49 GHz
memory bus	136.8 GB/s

machine 2	
CPU	2 Intel E5640
cores	2·6
main memory, per node	48 GB
l3-cache, per CPU	2·6 MB, shared
clock frequency	2.67 GHz
memory bus	QPI, 26.7 GB/s
GPU	2 Nvidia Tesla M2050
multiprocessors, per GPU	14
global memory, per GPU	2687 MB
memory clock rate	1.55 GHz
memory bus	141.5 GB/s

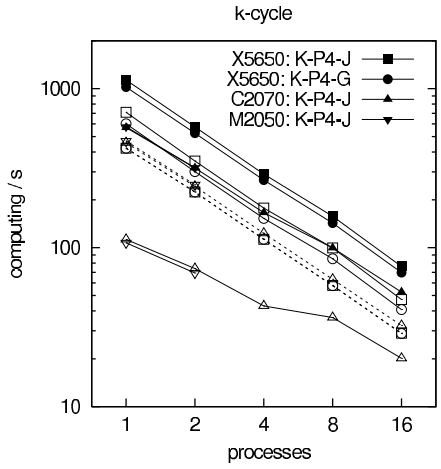

Figure 2. Computing times k-cycle algorithms: solid line, filled symbols: total time in AMG solver, solid line, empty symbols: AMG solution phase, dashed line, empty symbols: AMG setup phase

the setup and solution phase separately. The measurements comprise all operations including data transfers. During the measurement no data is written to the hard disk. With the option `-CPU_bind` of the `mpirun` command we disable the automatic mapping of processes to the cores. We bind all processes symmetrically to the cores on the sockets that are connected directly to the GPUs. The ECC memory protection of the GPUs is activated. Since at most four GPUs are available per node, we have also limited the number of used CPU cores to four; higher number of parallel processes are placed in groups of four to the avaliable nodes.

The measured computing times are shown in Figure 2

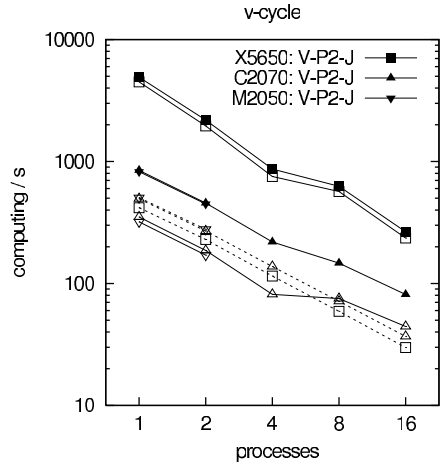

Figure 3. Computing times v-cycle algorithms: solid line, filled symbols: total time in AMG solver, solid line, empty symbols: AMG solution phase, dashed line, empty symbols: AMG setup phase

and 3 for the k-cycle and v-cycle algorithms, respectively. Due to the sequential nature of the Gauß-Seidel algorithm, an efficient implementation of this algorithm on GPU for a matrix representing the discretisation of an unstructured mesh seems not to be feasible. But the comparison between the Gauß-Seidel and Jacobi algorithms on CPUs for the k-cycle algorithm demonstrates that the difference between both algorithms in terms of run-time is only about 10%. The overhead time in the GPU implementation for the setup (translation of the matrix into ICRS format and transfer to GPU) that is done on the CPU is around 5% of the total setup cost. The solution phase of the k-cycle algorithm is accelerated through the use of GPUs by a factor between 7 (sequential run) and 5 (4 parallel processes on the same node) or 2.5 (16 parallel processes on 4 nodes). The solution of the v-cycle algorithm is accelerated by a factor between 12, 9, and 4, respectively. The difference is due to the structure of both algorithms. Whereas the v-cycle requires atomic operations that are generally expensive on GPUs only on the top level of the grid hierarchy (for the scalar products of the CG and for the calculation of vector norms to terminate the iteration), the k-cycle algorithm requires up to seven of those operations on each level of the grid hierarchy per visit, and it might perform due to the adaptivity even a w-cycle which results in up to l^2 number of visits on level l per iteration. Considering the entire AMG algorithm including the setup that still resides on the CPU, the acceleration of the k-cycle algorithm is around 50%, and that of the v-cycle algorithm between 65% and 75%.

It is worthwhile examining the parallel efficiency of the calculations using multiple GPUs since in this case the exchange of information from one GPU to another needs to be considered. For this purpose we define the parallel

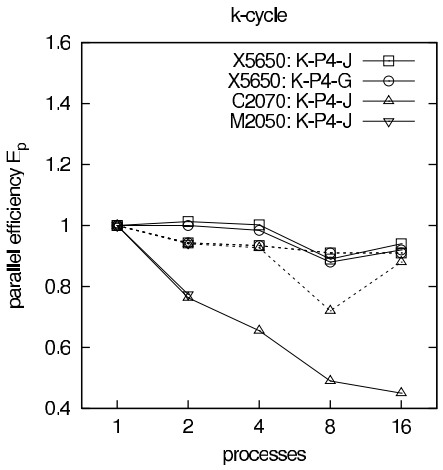

Figure 4. Parallel efficiency k-cycle algorithms: solid line: AMG solution phase, dashed line: AMG setup phase

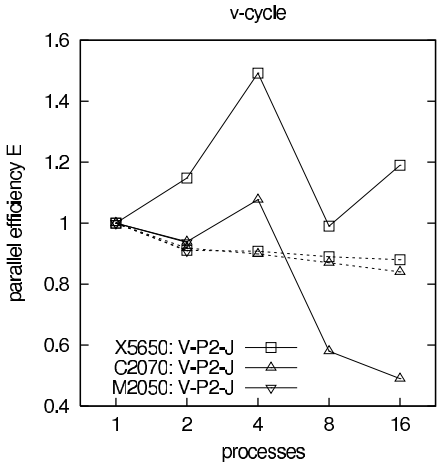

Figure 5. Parallel efficiency v-cycle algorithms: solid line: AMG solution phase, dashed line: AMG setup phase

efficiency as

$$E_p := \frac{t_1}{p \cdot t_p} \tag{5}$$

where t_p is the run-time with p parallel processes. The values for E_p are shown in Figure 4 and 5. The parallel efficiency of the k-cycle calculations are well below that of the corresponding CPU implementation, but they are still on an acceptable level for the target applications. An obvious reason for the retardation of the parallel k-cycle algorithm is again the high number of global communication events due to the numerous scalar products: these require not only global communication between the controlling CPU processes, but also, before these communication events, the transfer of the partial sums from the GPUs to the CPUs, and after these communication events, the transfer back to the GPUs. Regarding the parallel efficiency of the v-cycle algorithms, the CPU calculations show values much higher

Table II
PARALLEL EFFICIENCIES FOR DIFFERENT DISTRIBUTION OF THE PROCESSES TO THE NODES: ALGORITHM K-P4-J

CPU	processes per node			GPU	processes per node		
1 node	1.00	1.15	1.49	1 node	1.00	0.94	0.92
2 nodes	1.14	1.49	0.99	2 nodes	0.94	1.10	0.78
4 nodes	1.56	1.05	1.19	4 nodes	1.11	0.69	0.65

Table III
PARALLEL EFFICIENCIES FOR DIFFERENT DISTRIBUTION OF THE PROCESSES TO THE NODES: ALGORITHM V-P2-J

CPU	processes per node			GPU	processes per node		
1 node	1.00	1.01	1.00	1 node	1.00	0.76	0.65
2 nodes	1.01	1.00	0.89	2 nodes	0.78	0.68	0.49
4 nodes	1.04	0.89	0.74	4 nodes	0.69	0.53	0.45

than one which indicates that the algorithms greatly profit from the caches. The GPU implementation is not able to exploit such effects, but other than for the k-cycle algorithms, the v-cycle algorithms do not suffer from additional communication overhead that cannot be overlapped with other operations. The high parallel efficiency of these GPU calculations indicates that the implementation of the data exchange of the values in the buffer layer is efficient.

Finally, we are interested in the performance of the exchange mechanism. If the communicating processes (for both, CPU and GPU computation) run on a single node, then the MPI exchange mechanism exploits the available common memory which results in a very fast exchange with low latency. If the processes run on different nodes, then the communication is required to go through the network interconnect which is in general significantly slower. It is known that the principle of the overlap of communication and computation is appropriate for CPU calculations on clusters relying on network interconnect, see Emans [5]. In order to show that our parallel GPU implementation has the same advantages also if the processes communicate only through the relatively slow network interconnect, we have compiled the parallel efficiencies of the runs with up to four parallel processes that are distributed differently to the available four nodes of the computing system in tables II and III. The observations are, on the one hand, that the parallel efficiency of the GPU-implementation is generally weaker than that of the conventional CPU implementation. The reasons for this are less favourable implementations of atomic operations like scalar product and norms on GPU and the fact that the GPU cannot profit from cache effects since there is no significant cache on the GPU. Both effects have been discussed above. On the other hand, the data shows that the parallel efficiency is also on GPUs essentially independent of the distribution of the processes to the nodes: compare e.g. four processes on one node and four processes on four nodes. This allows the conclusion that our parallelisation appears to be appropriate.

V. Conclusion and future work

Using the ICRS sparse matrix format and an asynchronous exchange mechanism that has been extended to handle calculations with multiple GPUs, we show that the solution phase of the standard v-cycle AMG with simple aggregation is accelerated by a factor of up to 12. The solution phase of the more advanced k-cycle AMG runs faster by a factor of up to 7. Future activities are directed towards the improvement of the performance of the total algorithm including the setup phase: For this, it is necessary to accelerate the setup, either by employment of GPUs, or by algorithmic improvements, or by both.

Acknowledgement

Part of the contribution of Maximilian Emans has been supported in the framework of "Industrielle Kompetenzzentren" by the Austrian Ministerium für Wirtschaft, Jugend und Familie and by the government of Upper Austria.

References

[1] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Trans. Graphics*, vol. 22, no. 3, pp. 908–916, 2003.

[2] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys, "A multigrid solver for boundary value problems using programmable graphics hardware," in *Eurographics/SIGGRAPH Workshop on Graphics Hardware 2003*, M. Doggett, W. Heidrich, W. Mark, and A. Schilling, Eds., 2003, pp. 102–135.

[3] G. Haase, M. Liebmann, C. Douglas, and G. Plank, "A parallel algebraic multigrid solver on graphical processing unit," in *High Performance Computing and Applications 2010*, ser. Lecture Notes in Computer Science, W. Zhang, Z. Chen, C. Douglas, and W. Tong, Eds. Springer-Verlag Berlin Heidelberg, 2010, vol. 5938, pp. 38–47.

[4] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek, "Using GPUs to improve multigrid solver performance on a cluster," *International Journal of Computational Science and Engineering*, vol. 4, no. 1, pp. 36–55, 2008.

[5] M. Emans, "AMG for linear systems in engine flow simulations," in *PPAM2009, Part II*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds. Springer-Verlag Berlin Heidelberg, 2010, vol. 6068, pp. 350–359.

[6] R. Falgout, J. Jones, and U. Yang, "Conceptual interfaces in hypre," *Future Generation Computer Systems*, vol. 22, pp. 239–251, 2006.

[7] P. Vaněk, J. Mandel, and M. Brezina, "Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems," *Computing 56*, pp. 179–196, 1996.

[8] Y. Notay, "An aggregation-based algebraic multigrid method," *Electronic Transactions on Numerical Analysis*, vol. 37, pp. 123–146, 2010.

[9] M. Emans, "Aggregation AMG for distributed systems suffering from large message sizes," in *EuroPar2010, Part II*, ser. Lecture Notes in Computer Science, P. D'Ambra, M. Guarracino, and D. Tallia, Eds. Springer-Verlag Berlin Heidelberg, 2010, vol. 6272, pp. 89–100.

[10] Y. Notay, "Flexible conjugate gradients," *SIAM Journal on Scientific Computing*, vol. 22, no. 4, pp. 1444–1460, 2000.

[11] V. Henson and U. Yang, "BoomerAMG: a parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, pp. 155–177, 2002.

[12] M. Emans, "Efficient parallel amg methods for approximate solutions of linear systems in CFD applications," *SIAM Journal on Scientific Computing*, vol. 32, pp. 2235–2254, 2010.

[13] G. Haase, M. Kuhn, and S. Reitzinger, "Parallel algebraic multigrid methods on distributed memory computers," *SIAM Journal on Scientific Computing*, vol. 24, no. 2, pp. 410–427, 2002.

[14] O. Axelsson and P. Vassilevski, "Variable-step multilevel preconditioning methods, I: Self-adjoint and positive definite elliptic problems," *Numerical Linear Algebra with Applications*, vol. 1, pp. 75–101, 1994.

[15] M. Emans, "Performance of parallel AMG-preconditioners in CFD-codes for weakly compressible flows," *Parallel Computing*, vol. 36, pp. 326–338, 2010.

[16] M. Liebmann, "Efficient PDF solvers on modern hardware with applications in medical and technical sciences," Ph.D. dissertation, University of Graz, 2009.

[17] V. Pržulj and B. Basara, "Bounded convection schemes for unstructured grids," *AIAA-paper*, vol. 2001-2593, 2001.

[18] B. Basara, A. Alajbegovic, and D. Beader, "Simulation of single- and two-phase flows on sliding unstructured meshes using finite volume method," *International Journal for Numerical Methods in Fluids*, vol. 45, pp. 1137–1159, 2004.

[19] B. Basara, "Employment of the second-moment turbulence closure on arbitrary unstructured grids," *International Journal for Numerical Methods in Fluids*, vol. 44, pp. 377–407, 2004.

[20] S. Patankar and D. Spalding, "A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows," *International Journal Heat Mass Transfer*, vol. 15, pp. 1787–1806, 1972.

[21] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, pp. 359–392, 1998.