Efficient Setup of Aggregation AMG for CFD on GPUs

Maximilian Emans[1] and Manfred Liebmann[2]

[1] Johann Radon Institute for Computational and Applied Mathematics and IMCC GmbH, both 4040 Linz, Austria
`maximilian.emans@ricam.oeaw.ac.at`

[2] Institute for Mathematics and Scientific Computing, University of Graz, 8010 Graz, Austria
`manfred.liebmann@uni-graz.at`

Abstract. We explore a GPU implementation of the Krylov-accelerated AMG algorithm with flexible preconditioning. We demonstrate by means of two benchmarks from industrial CFD application that the acceleration with multiple GPUs speeds up the solution phase by a factor of up to 13. In order to achieve a good performance of the whole AMG algorithm, we propose for the setup a substitution of the double-pairwise aggregation by a simpler aggregation scheme skipping the calculation of temporary grids and operators. The version with the revised setup reduces the total computing time on multiple GPUs by further 30% compared to the GPU implementation with the double-pairwise aggregation. We observe that the GPU implementation of the entire Krylov-accelerated AMG runs up to four times faster than the fastest CPU implementation.

Introduction

When the speech is on numerical simulation of physical phenomena, the notion of most engineers and scientists outside the high-performance computing community is that computers similar to the well-known desktop personal computers are used. Eventually, some of these computers might be linked by some kind of network interconnect to local networks or to clusters. In fact, the technical basis of the processing units, the vast majority of simulations in science and engineering runs on, is the same as that of a modern workstation, i.e. one or several many-core CPUs that are recognised to be multiple instruction – multiple data (MIMD) systems according to Flynn's taxonomy [1].

Since a couple of years, however, also particular single instruction – multiple data (SIMD) architectures in the form of graphics processing units (GPUs) have started to attract the attention of both, users and developers of numerical simulation software. Recent development in both, hardware design and software tools made it possible to exploit the large computational power of the GPUs for numerical calculations. However, so far not for every single algorithm a dedicated efficient GPU implementation can be devised. A prominent example is the Gauß-Seidel smoother: While on conventional CPUs this algorithm is frequently used since it can be implemented efficiently, its employment on GPU is mitigated by its inherent sequential nature. In GPU implementations it is therefore usually substituted by a ω-Jacobi smoother, although this algorithm has less favourable smoothing properties. But due to such substitutions or due to algorithmic adjustments in other cases it is today possible that many calculations in science and engineering are executed on GPU. Nevertheless, it appears that such GPU-accelerated simulations have not yet reached the significance of the CPU-based ones in relevant applications in science and engineering. Since the reduction of the computing times is the main motivation for the use of GPUs, the improvement of the performance of GPU-accelerated simulations is a major issue. We will report on a fast implementation of an algebraic multigrid (AMG) solver for linear systems that has been shown to be efficient for problems in fluid dynamics, but that can also be used in other applications.

It is known that k-cycle AMG, see Notay [2], has a particularly simple and computationally inexpensive setup since it uses double-pairwise aggregation. It is therefore well suited as linear solver within the iterative algorithms used in computational fluid dynamics (CFD). Compared to other common methods like Smoothed Aggregation, see Vaněk et al. [3], the inexpensive setup makes this algorithm also attractive for GPU calculations, since it is particularly difficult to implement the setup on the GPU efficiently. The absolute run-time of the setup of the double-pairwise aggregation is small compared to the run-time of the setup of other AMG algorithms. But it is still large in comparison to the time spent in the GPU-accelerated solution phase.

In this contribution we show that the attractive run-times of k-cycle algorithms on GPU-accelerated hardware can be even more reduced if the double-pairwise aggregation is replaced by a simple greedy aggregation algorithm that we refer to as plain aggregation. The latter method has the advantage that it does not require the computation of an intermediate (and finally discarded) grid level like the algorithm originally chosen by Notay [2] does. With GPU-acceleration, the additional cost due to the slightly worse convergence of the simpler aggregation scheme is outweighed by the dramatically reduced run-time of the setup.

1 Aggregation AMG algorithms

The AMG algorithm is here applied as a preconditioner to the pressure-correction equation in a finite volume based CFD-code, see Emans [4]. We denote this system as

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

where $A \in \mathbb{R}^{n\times n}$ is symmetric and positive definite or semi-definite, $\boldsymbol{b} \in \mathbb{R}^n$ is some right-hand side vector and $\boldsymbol{x} \in \mathbb{R}^n$ the solution where 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. [5]. 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 a neighbouring process.

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

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

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 $(l = 2, ..., l_{max})$ are referred to as the setup phase of AMG.

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

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

The aggregation methods split the number of nodes on the fine grid into a lower number of disjoint sets of nodes, the so-called aggregates. The mapping from the coarse grid to the fine grid is than achieved by simply assigning the coarse-grid value of the aggregate to all the fine-grid nodes belonging to this aggregate. This corresponds to a constant interpolation. The prolongation operator of this scheme has only one non-zero entry in each row with the value one such that the evaluation of eqn. (2) is greatly simplified to an addition of rows of the fine-grid operator. In the following we restrict ourselves to this group of methods.

Double-pairwise aggregation The first step of the double-pairwise aggregation that has been used by Notay [2] is the aggregation of the set of nodes into aggregates of pairs of nodes. For this we use algorithm 1.

For the pairwise aggregation the output of this algorithm, i.e. the set of aggregates G_i $(i = 1, ..., n_{l+1})$, is used to define the prolongation operator P_l. The calculation of the elements of the coarse-grid matrix $A_l = P_l^T A_l P_l$ with eqn. (2) is implemented as the addition of two rows in two steps: First, the rows are extracted from the matrix storage structure in a way that the corresponding elements of the data array are put in a single array and the row pointers in another array of the same size. Second, the column pointers are replaced by the indices of the corresponding coarse-grid aggregates; then both arrays are sorted with respect to the new column pointers where matrix elements with the same column pointer are added. The parallel version of the method restricts the aggregates to nodes belonging to the same parallel domain.

For the double-pairwise aggregation, the set of aggregates obtained with algorithm 1 to A_l is used to define the intermediate prolongation operator P_{l1}.

Algorithm 1 Pairwise aggregation (by Notay [2], simplified version)

```
Input:          Matrix $A = (a_{ij})$ with $n$ rows.
Output:         Number of coarse variables $n_c$ and aggregates $G_i$, $i = 1, ..., n_c$
                (such that $G_i \cap G_j = \emptyset$ for $i \neq j$).
Initialisation: $U = [1, n]$
                for all $i$: $S_i = \left\{ j \in U \setminus \{i\} \mid a_{ij} < -0.25\max_{(k)} |a_{ik}| \right\}$,
                for all $i$: $m_i = | \{ j | i \in S_j \} |$,
                $n_c = 0$.
Algorithm:      While $U \neq \emptyset$ do:
                1. select $i \in U$ with minimal $m_i$; $n_c = n_c + 1$.
                2. select $j \in U$ such that $a_{ij} = \min_{k \in U} a_{ik}$
                3. if $j \in S_i$: $G_{n_c} = \{i, j\}$, otherwise $G_{n_c} = \{i\}$
                4. $U = U \setminus G_{n_c}$
                5. for all $k \in G_{n_c}$: $m_l = m_l - 1$ for $l \in S_k$
```

With this, the elements of the corresponding intermediate coarse-grid matrix $A_{l+1/2} = P_{l1}^T A_l P_{l1}$ are calculated in the same way as for the pairwise aggregation. In order to obtain the matrix A_{l+1}, the double-pairwise aggregation applies the same procedure a second time, this time with input $A_{l+1/2}$ instead of A_l for algorithm 1 which gives rise to the prolongation operator P_{l2}. A_{l+1} is calculated as $A_{l+1} = P_{l2}^T A_{l+1/2} P_{l2}$. The final prolongation operator is formally $P_l = P_{l2} P_{l1}$. Since it contains only the information to which coarse-grid element or aggregate a fine-grid node is assigned, it is sufficient to store it as an array of size n_l carrying the index of the coarse-grid node. The operators $A_{l+1/2}$, P_{l1}, and P_{l2} are discarded after A_{l+1} has been calculated. We refer to the method as P4.

Plain aggregation algorithm Our plain aggregation algorithm comprises the following steps:

1. **Determine strong connectivity:** Edges of the graph of the matrix A_l for which the relation
$$|a_{ij}| > \beta \cdot max_{(j)} |a_{ij}| \quad (4)$$
holds, are marked as strong connections. The criterion β depends on the level of the grid hierarchy l and it is defined according to Vaněk et al. [3] as
$$\beta := 0.08 \left(\frac{1}{2} \right)^{l-1} \quad (5)$$
2. **Start-up aggregation:** All nodes are visited in the arbitrary order of their numeration. Once a certain node i is visited in this process, a new aggregate is built if this node is not yet assigned to another aggregate. Each of the neighbours of node i that is strongly connected to this node and that is not yet assigned to another aggregate is grouped into this aggregate as long as the number of nodes is lower than the maximum allowed aggregate size.

3. **Enlarging the decomposition sets:** Remaining unassigned nodes are joint to aggregates containing any node they are strongly connected to as long as the number of nodes in this aggregate is lower than twice the maximum allowed aggregate size. If there is more than one strongly connected node in different aggregates, the one with the strongest connection determines the aggregate this node is joint with.
4. **Handling the remnants:** Unassigned nodes are grouped into aggregates of a strongly connected neighbourhood. Twice the maximum allowed aggregate size is allowed.

This algorithm follows closely the one proposed by Vaněk et al. [3] with the essential difference that we restrict the number of nodes per aggregate which gives rise to a parameter of this algorithm. In step (3) we allow twice the maximum number of nodes in order to avoid a large number of single-point aggregates. Usually only a few such enlarged aggregates are formed. In parallel, only nodes assigned to the same process are grouped into aggregates.

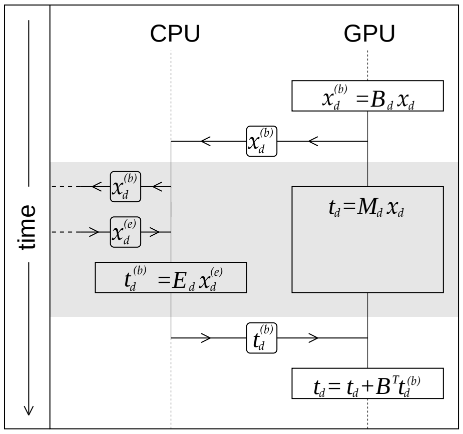
The aggregates generated this way are used in a scheme with constant interpolation, i.e. in a way that all fine-grid nodes assigned to a certain aggregate receive the value of the coarse-grid node this aggregate forms on the coarse-grid. The corresponding prolongation operator will have a similarly simple structure as that of the described pairwise aggregation method. The coarse-grid operator is again obtained by adding the rows of the fine-grid matrix that are associated with the nodes assigned to an aggregate. This is done exactly as for the pairwise aggregation. The coarsening scheme that is defined by this procedure will be only useful, if the maximum number of nodes per aggregate is kept relatively low. In preliminary experiments we found that a maximum number of nodes per aggregate of 6 results in an efficient and robust algorithm with good convergence properties. We denote this aggregation scheme in a k-cycle scheme as K-R6.

Smoothed Aggregation The Smoothed Aggregation algorithm of Vaněk et al. [3] is derived from this algorithm: It refines the aggregation scheme by applying a ω-Jacobi smoothing step (along the paths of the fine-grid matrix) to the prolongation operator to obtain the final prolongation operator. This way the quality of the interpolation is improved, but the structure of the operator is now similarly complex as the structure of the operators of classical AMG with the consequence, that the calculation of the coarse-grid operator with eqn. (2) can no longer be simplified in the described way. For the use as Smoothed Aggregation scheme, the number of nodes per aggregate is not limited. The Smoothed Aggregation is used in a v-cycle scheme; the algorithm is denoted as V-SA.

Implementation on GPU

The setup phase is implemented conventionally on the CPU as it has been described in Emans [6]. The standard CRS format, see e.g. Falgout et al. [5], is used for the matrices. After any matrix has been defined or calculated, it is translated into the Interleaved Compressed Row Storage (ICRS) format on the CPU and then transferred to GPU memory. The definition of this format is found e.g. in Haase et al. [7]. The corresponding algorithm devised in the same publication is used to carry out matrix-vector operations. This applies to the system matrices on all levels. The particularly simple structure of the restriction and prolongation operators of the aggregation algorithms P4 and R6 gives rise to a simplified version of the ICRS format: Since the value of all non-zero matrix elements is the same (one), it does not make sense to store these values. Therefore, only the number of elements per row, the displacement and the column index for each element is stored. 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.

For an efficient parallel implementation, the concept of overlapping the data exchange with the internal operations, implemented by means of the asynchronous point-to-point exchange mechanism of MPI, see e.g. Emans [8], needed to be modified: While the internal work is done on the device, the host manages the data exchange by means of the same asynchronous point-to-point exchange mechanism of MPI, and computes $\boldsymbol{t}_d^{(b)} := E_d \boldsymbol{x}^{(e)}$, see figure 1.



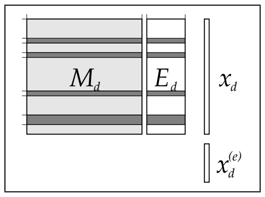


Fig. 1. Flow chart of parallel matrix-vector product on machines with multiple GPUs, the parallel execution on CPU and GPU is marked grey (left), notation (right)

The solver algorithms described above are integrated into the program FIRE 2011, developed and distributed by AVL GmbH, Graz. The solver part of this program is coded in FORTRAN 90 and compiled by the hp FORTRAN compiler, version 11.1. The used MPI library is Platform MPI, version 7.01. The GPU related code is written in Cuda and compiled by the Nvidia compiler version 4.0. The link between the MPI library and the Cuda part is ensured by C-binding.

Benchmarks

Problem 1 is the unsteady simulation of the flow of cold air into the cylinder of an automotive gasoline engine. The cylinder has a diameter of 0.08m. During the observed time the piston head is moving from its top position downwards and air at a temperature of 293K flows at a rate of around 1 kg/s into the cylinder. The mass flow at the boundary is prescribed as a function of time according to experimental data for this engine. The volume of the computational domain is initially 0.18l. The mesh consists of around $1.4 \cdot 10^6$ finite volumes 80% of which are hexagonal.

Problem 2 is the steady simulation of the internal flow through a water-cooling jacket of an engine block. Cooling water, i.e. a 50% water/glycol mixture, flows at a rate of 2.21 kg/s into the geometry through an inlet area of $0.61 \cdot 10^{-3}\mathrm{m}^2$ and leaves it though an outlet area of $0.66 \cdot 10^{-3}\mathrm{m}^2$. The maximum fluid velocity is 4.3m/s. The volume of the cooling jacket is 1.14l. The turbulence is modelled by a k-ζ-f model according to Hanjalic et al. [9]. The computational domain is discretised by an unstructured mesh of about $5 \cdot 10^6$ cells of which around 88% are hexagonal.

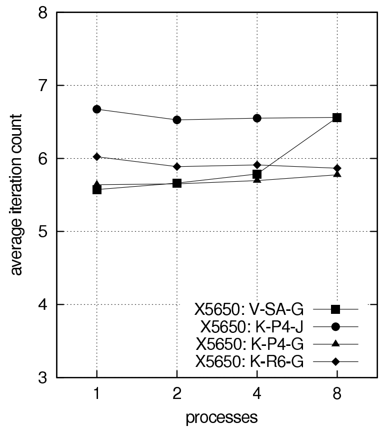
For both problems, the Navier-Stokes equations are solved by the finite-volume based SIMPLE scheme with collocated variable arrangement, see Patankar and Spalding [10]. We apply our AMG algorithms to the pressure-correction equation only. This system is symmetric and positive definite and its solution is usually the most time consuming part of the whole simulation. In the case of problem 1, the SIMPLE iteration is terminated after 50 iterations, i.e. for this problem we consider the solution of 50 systems with different matrix and different right-hand side. For problem 2 three time steps with together 89 SIMPLE iterations are calculated, i.e. here 89 different systems are solved.

The benchmarks were run on two different computers. Both computers had four Intel X5650 CPUs. Additionally, both computers were equipped with four graphics boards by Nvidia: computer 1 with four Tesla C2070, computer 2 with four GeForce GTX480. The most important specifications of the hardware are compiled in Table 1.

Table 1. Hardware specification

	computer 1 & 2
CPUs	Intel X5650
cores	2·6
main memory	96 GB
L3-cache	4·6 MB, shared
clock rate	2.67 GHz
memory bus	QPI, 26.7 GB/s

	computer 1	computer 2
GPUs (Nvidia)	Tesla C2070	GeForce GTX480
multiprocessors	14	15
L2-cache	–	768 kB
global memory	5375 MB	1535 MB
memory clock rate	1.49 GHz	1.85 GHz
memory bus	136.8 GB/s	169.2 GB/s



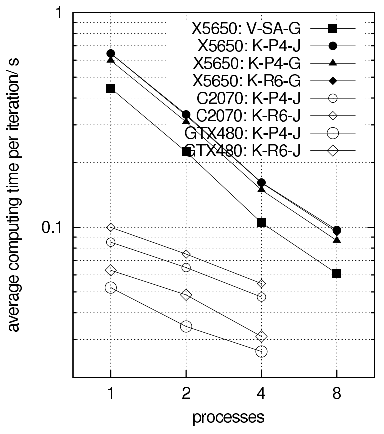
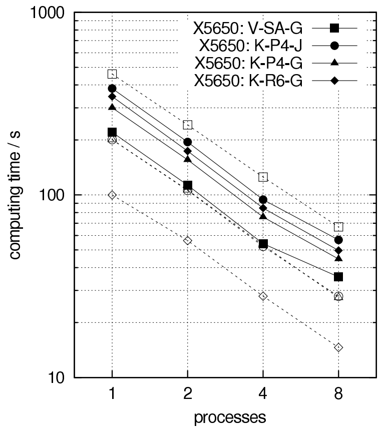


Fig. 2. Problem 1: average number of iterations (left) and average time per iteration on CPU and different GPUs (right)

Problem 1: different types of GPUs The average number of iterations in the left diagram of figure 5 shows that the number of iterations is increased by about 10 % if the Gauß-Seidel smoother is replaced by the Jacobi smoother, which is common practice in AMG on GPU. The substitution of the double-pairwise aggregation (K-P4-J) by the plain aggregation with six nodes per aggregate (K-R6-J) increases the accumulated number of iterations by about a similar amount.

The comparison of the average computing time per iteration shows that computer 2 with the GeForce GTX480 graphics boards is around 40% faster than computer 1 with the Tesla C2070. This is essentially due to the faster memory bus of the GeForce GTX480. The execution of the same solution algorithm on the faster GPU is up to 13 times faster than on the CPU. The comparison of the computing times for setup phase and solution phase in figure 2 shows that for the Smoothed Aggregation AMG (V-SA-G) the setup is the dominant part. Since on the GPU we essentially reduce only the solution phase, this algorithm appears not to be favourable, although the time per iteration is short due to the v-cycle. The setup of the algorithm with the plain aggregation scheme (K-R6-J) is significantly faster than that of the double-pairwise aggregation (K-P4-J). Since the setup becomes dominant on the GPU, this leads to a significant reduction of the total computing time, too, see figure 3. Thus, while on the CPU, the fastest algorithm employs the double-pairwise aggregation, the fastest algorithm on th GPU employs the plain aggregation scheme. The total computing times with computer 2 using the GeForce GTX480 graphics boards is up to four times faster than the fastest calculation on the CPU, see again figure 3. Finally, the parallel efficiency

$$E_p := \frac{t_1}{p \cdot t_p} \tag{6}$$



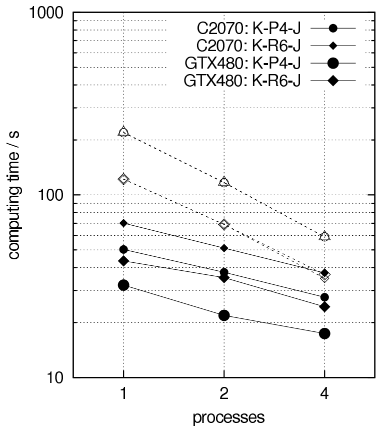
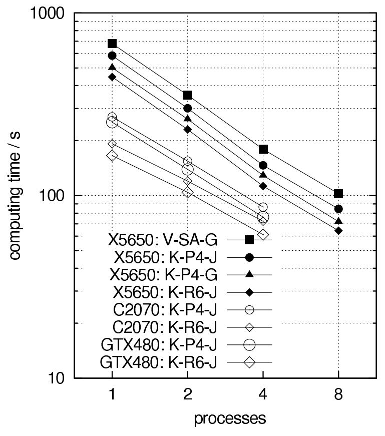


Fig. 3. Computing times for problem 1: CPU calculations (left), GPU calculations (right), filled symbols: AMG solution, empty symbols: AMG setup



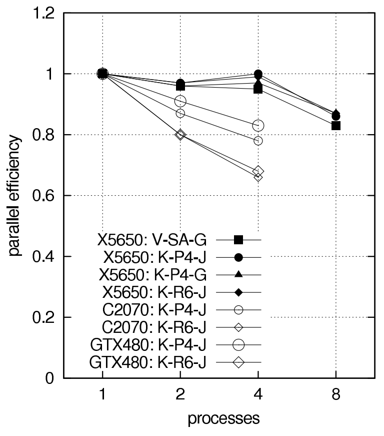
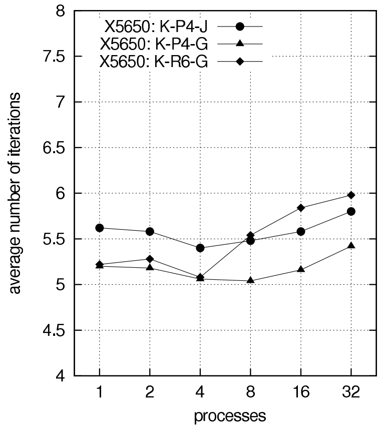


Fig. 4. Total computing times for problem 1 (left), Parallel efficiency for problem 1 (right)

where t_p is the run-time with p parallel processes, is presented in the right diagram of figure 3. Although the parallel efficiency of the calculations with GPU acceleration is inferior to that of the CPU calculations, it is within an acceptable range for practical applications.

Problem 2: multiple GPUs on different nodes With regard to the increase of the number of iterations due to the use of algorithms better adapted to the requirements of the GPU, i.e. the replacement of the Gauß-Seidel smoother by the Jacobi smoother and the replacement of the double-pairwise aggregation by the plain aggregation, we make for problem 2 the same observations as for problem

1, see figure 5. The right diagram in this figure shows that in this case, too, the cost per iteration of the double-pairwise aggregation and the plain aggregation scheme are almost identical. It is, however, more important to observe that the usage of additional nodes with GPUs still accelerates the calculation in a reasonable manner: Remember that we have four GPUs per node, i.e. the calculation with 8 and 16 parallel processes runs on two and four nodes, respectively.



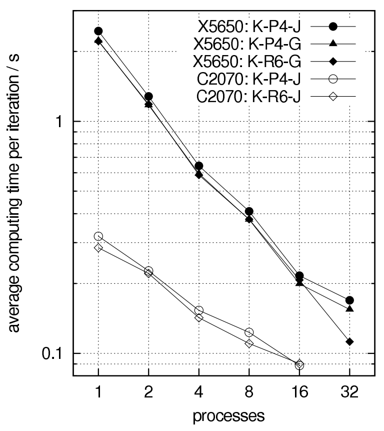
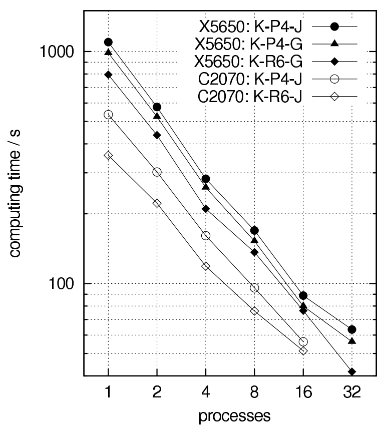


Fig. 5. Cumulative iteration count of various aggregation AMG algorithms (left) and Average time per iteration of various AMG algorithms on CPU and GPU (right)



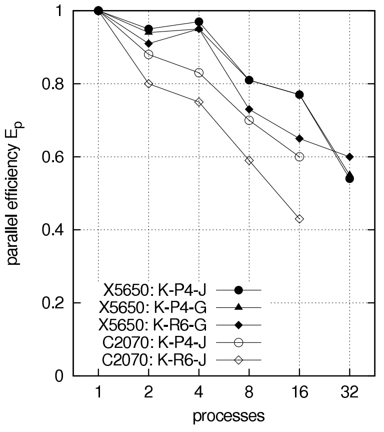


Fig. 6. Cumulative computing times: dashed line, empty symbols: AMG setup phase

The left diagram in figure 6 shows that the plain aggregation scheme leads to a significantly faster AMG method than the double-pairwise aggregation. For the GPU calculations the portion of the computing time spent in the setup is larger than for the CPU calculations. The reduction of the total computing time by substituting the double-pairwise aggregation by the plain aggregation is therefore for the GPU computations relatively large (30 %) whereas for the CPU computations it is only around 10 %. In total, i.e. including the setup on the CPU, the GPU implementation on the Tesla C2050 is around twice as fast as the fastest conventional implementation. Computations for problem 2 on computer 2 could not be carried out since the memory of the GeForce GTX480 graphics boards of computer 2 was not sufficient.

2 Conclusions

We have presented a parallel k-cycle AMG for GPUs. The conventional double-pairwise aggregation, implemented on the CPU, contributes significantly to the total computing time of the k-cycle AMG on GPU-accelerated hardware. It has been shown that it can be replaced by a more efficient plain aggregation algorithm. We have tested our implementation on a GPU cluster with four nodes each of which was equipped with four Nvidia Tesla C2050 GPUs. On a computer with four of the faster GeForce GTX480 graphics boards we could show that the entire AMG algorithm runs up to four times faster than the fastest AMG variant on the CPU. Although the parallel efficiency is already acceptable, future effort should be directed to an improved parallel performance.

References

1. Flynn, M.: Some computer organizations and their effectiveness. IEEE Transactons on Computers **C-21** (1972) 948–960
2. Notay, Y.: An aggregation-based algebraic multigrid method. Electronic Transactions on Numerical Analysis **37** (2010) 123–146
3. Vaněk, P., Mandel, J., Brezina, M.: Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Computing 56 (1996) 179–196
4. Emans, M.: Efficient parallel amg methods for approximate solutions of linear systems in CFD applications. SIAM Journal on Scientific Computing **32** (2010) 2235–2254
5. Falgout, R., Jones, J., Yang, U.: Conceptual interfaces in hypre. Future Generation Computer Systems **22** (2006) 239–251
6. Emans, M.: Performance of parallel AMG-preconditioners in CFD-codes for weakly compressible flows. Parallel Computing **36** (2010) 326–338
7. Haase, G., Liebmann, M., Douglas, C., Plank, G.: A parallel algebraic multigrid solver on graphical processing unit. In Zhang, W., Chen, Z., Douglas, C., Tong, W., eds.: High Performance Computing and Applications 2010. Volume 5938 of Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg (2010) 38–47
8. Emans, M.: AMG for linear systems in engine flow simulations. In Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J., eds.: PPAM2009, Part II. Volume

6068 of Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg (2010) 350–359

9. Hanjalic, K., Popovac, M., Hadziabdic, M.: A robust near-wall elliptic-relaxation eddy-viscosity turbulence model for cfd. International Journal of Heat and Fluid Flow **25** (2004) 1047–1051
10. Patankar, S., Spalding, D.: A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. International Journal Heat Mass Transfer **15** (1972) 1787–1806