A Comparison of Algebraic Multigrid Preconditioners using Graphics Processing Units and Multi-Core Central Processing Units

Markus Wagner[1], **Karl Rupp**[1,2], **Josef Weinbub**[1]
[1] **Institute for Microelectronics, TU Wien**
Gußhausstraße 27-29/E360, 1040 Wien, Austria
[2] **Institute for Analysis and Scientific Computing, TU Wien**
Wiedner Hauptstraße 8-10/E101, 1040 Wien, Austria
`wagner.m01@inode.at`, {`rupp|weinbub`}`@iue.tuwien.ac.at`

Keywords: Algebraic Multigrid, High Performance Computing, OpenCL, ViennaCL

Abstract

The influence of multi-core central processing units and graphics processing units on several algebraic multigrid methods is investigated in this work. Different performance metrics traditionally employed for algebraic multigrid are reconsidered and reevaluated on these novel computing architectures. Our benchmark results show that with the use of graphics processing units for the solver phase, it is crucial to keep algebraic multigrid setup low, even if this leads to a higher number of solver iterations.

1. INTRODUCTION

Many physical processes are described by partial differential equations, for which many popular numerical solution approaches such as the finite element method lead to large sparse systems of linear equations. Ideally, the computational effort for the solution of these systems depends linearly on the number of unknowns. While direct methods or unpreconditioned iterative solvers typically show sub-optimal complexity [1], multigrid methods are able to provide such an optimal complexity for certain classes of problems [2].

Multigrid methods are based on the observation that relaxation schemes show a fast reduction of oscillatory components in the algebraic error. After a few relaxation steps, a smooth error is obtained, for which a good correction can be computed on a coarser grid. This idea is then applied recursively for the solution of the system for the coarse-grid correction. As soon as a reasonably small system is obtained, it is solved exactly by direct methods.

The nested grid hierarchy used within multigrid methods can be set up in two ways. In geometric multigrid methods, a nested grid hierarchy is used. Typically, structured grids are employed for this purpose, because it simplifies the transfer operations between the grid levels. In algebraic multigrid (AMG) methods, the coarse grid is determined in a purely algebraic way from the entries in the system matrix only. Consequently, AMG can be applied in a black-box manner to general systems of linear equations and is often the method of choice whenever unstructured grids on complicated domains are employed [2].

Several algebraic multigrid codes are available, e.g. the C-based library Hypre [3] for a distributed memory environment, PyAMG [4] providing a convenient Python interface, and the commercial SAMG [5] library. With the shift to multi-core central processing units (CPUs) and general purpose computations on graphics processing units (GPUs), parallelization strategies developed for distributed memory systems [6, 7] can be supplemented with parallelization strategies for shared memory systems in each node in order to fully exploit the parallelism provided by modern CPUs and GPUs. The fine-grained parallelism of GPUs is particularly challenging and has been successfully utilized in the context of aggregation-based AMG [8].

For programming GPUs, basically two approaches are in wide-spread use in the scientific community: The Compute Unified Device Architecture (CUDA) from NVIDIA [9], which is a vendor-specific environment for NVIDIA hardware, and the Open Computing Language (OpenCL) [10], which is an open industry standard for program execution across heterogeneous platforms. Even though both the CUDA and the OpenCL language are derived from the C programming language, they require to learn a new language and to get acquainted with the details of the underlying hardware. This is certainly not desirable from an abstraction point of view, for which high-level wrapper libraries such as MAGMA [11] and Cusp [12] using CUDA, and ViennaCL [13] using OpenCL are available.

In this work we consider the implementation of a family of AMG methods to be used both on multi-core CPUs and GPUs. Since many different variants of AMG have been proposed, the focus in this work is on a comparison of performance and on a reevaluation of performance metrics using multi-core CPUs as well as GPUs. First, the basic components of AMG are introduced and a comparison of different AMG approaches is given in Sec. 2. Performance metrics for AMG are defined and discussed in Sec. 3. Selected AMG variants are compared with respect to these metrics in Sec. 4. Finally, a conclusion is drawn in Sec. 5.

2. AMG COMPONENTS

Consider the linear system $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is the system matrix with entries $a_{i,j}$. The entries x_i of x are in the following associated with a set of points $\Omega = \{1, 2, \dots, n\}$ of an algebraic grid. Central to any multigrid method is the concept of a *smooth error e*, which is only slowly reduced by relaxation schemes and consequently a coarse-grid correction is employed. This is achieved by solving the residual equation $Ae = r$ on a coarser grid, where the residual vector is given by $r = Ax - b$. The computed correction is then transferred back to the fine grid to correct the fine-grid approximation $x \leftarrow x + e$.

In the following a brief discussion of the basic components of an AMG method are given. First, algorithms for smoothing the error (so-called *smoothers*) are discussed, then different coarsening strategies are presented, and finally transfer operators between grids are considered. For additional details about AMG methods we refer to the literature [2, 14].

2.1. Smoother

The efficiency of multigrid methods strongly depends on the smoother. A good smoother strongly reduces the oscillatory error components, such that the remaining smooth error can be well approximated by coarser grids and corrected there. First AMG methods relied on the Gauss-Seidel method due to its good smoothing properties. However, the main drawback of the Gauss-Seidel method is its sequential nature, which is particularly a concern on GPUs. Related to Gauss-Seidel is the Jacobi algorithm, possibly equipped with an additional relaxation factor ω. Good values for the relaxation parameter ω are problem-specific and may even depend on the respective level in the multigrid hierarchy. Nevertheless, a value of $\omega \approx 2/3$ is often used in practice and can even be shown to be optimal for the Poisson equation [2].

For the remainder of this work, the damped Jacobi algorithm will be used. Nevertheless, other parallel smoothers exist, of which the most popular are as follows: Block-based smoothers such as block-Gauss-Seidel can be used in a distributed memory environment [1], yet they show scaling issues [6]. Similarly, multicoloring approaches can be used, but have been reported to be inefficient, since often too many colors are generated particularly on coarser levels. This in turn results in too many idle processors, since not enough work load is available for all processors then [6].

A scalable approach is the use of polynomial smoothers, where motivated by Neumann series a polynomial of the system matrix is used as an approximation of its inverse. For the application in a multigrid context, error reduction properties need to be complementary to the coarse grid correction. For this purpose, certain information about the eigenvalues of the system matrix needs to be available, which can be an expensive task.

In principle, any parallel solver or preconditioner can be used as a smoother. Good smoothing properties for sparse approximate inverses were reported in [15], while incomplete LU factorizations are discussed in [2] and recently accelerated by GPUs in [16].

2.2. Coarsening

While the notion of a coarse grid is inherently figurative for geometric multigrid, different approaches for the construction of a coarse grid Ω^{k+1} from the fine grid Ω_k in the context of algebraic multigrid exist. An overview of different coarse grid generation strategies is given in the following, where the emphasis is on the coarsening methods used for the benchmarks in Sec. 4. Common to all approaches is the notion of coarse points (C-points), which are included in Ω^k as well as Ω^{k+1}, and fine points (F-points), which are in Ω^k, but not in Ω^{k+1}.

2.2.1. Classical Coarsening

The traditional coarsening algorithm is usually called RS (Ruge-Stüben) approach [17]. This approach relies on the concept of *strong influence* of a point j (with unknown x_j) on a point i (with unknown x_i) by means of the system matrix coupling coefficient $a_{i,j}$. A point i is strongly influenced by a point j if

$$-a_{i,j} \geq \theta \max_{k \neq i}(-a_{i,k}) \qquad (1)$$

with a strength threshold parameter $0 < \theta < 1$. Note that this definition was originally motivated by the assumption that A is a symmetric M-matrix [1], yet it can be formally applied to more general matrices. The classical RS-approach attempts to fulfill the following two conditions:

- *(C1)* Each point j influencing an F-point i is either a C-point or strongly depends on a C-point which strongly influences i.
- *(C2)* The C-points should be a maximal independent subset of all points, i.e. no two C-points are connected to each other, and if another C-point is added, independence is lost.

The first condition aims at ensuring interpolation quality, while the second condition aims at restricting the size of the coarse grid. Since it is in general impossible to fulfill both conditions, the second condition is only used as a guideline, while the first condition is enforced in order to ensure certain interpolation properties.

RS coarsening is achieved in a two-pass algorithm and works very well for many applications, yet the main drawback of the algorithm is its sequential nature of the second pass. A parallel variant of RS thus is to carry out only the first pass. By replacing *(C1)* with

- *(C1′)* Each F-point i needs to strongly depend on at least one C-point,

the computational effort can be reduced, since less C-points are generated, while typically a smaller convergence rate is obtained. Allowing for even less coarse points leads to aggressive coarsening, for which we refer to [2] for details.

A parallel variant of RS coarsening initially designed for distributed memory architectures is RS0-coarsening, which relies on a block decomposition of the system matrix A such that each processor works on one diagonal block only. Since RS0 ignores off-diagonal entries completely, worse convergence is to be expected. This drawback is mitigated by RS3-coarsening, which adds a third pass to RS0 in order to add additional coarse points from the set of points at processor boundaries to improve convergence.

Other strategies for distributed memory environments are CLJP coarsening, PMIS and HMIS, which are all based on parallel independent set algorithms [6]. These strategies are not considered in our comparison, since they are mostly designed to resolve problems of RS0 and RS3 coarsening in distributed memory settings, while the focus of this work is on a comparison on a single shared memory system.

2.2.2. Aggregation Coarsening

A different approach to coarsening is aggregation-based coarsening, which uses a different concept of influence strength. Here, matrix entries are only considered if they fulfill

$$|a_{i,j}| > \theta\sqrt{|a_{i,i}a_{j,j}|} \ . \qquad (2)$$

A single aggregate consists of a root point i and all points j for which $a_{i,j}$ fulfills (2). Similar to RS-type coarsening, this approach consists of two phases: In a first pass, new root points are picked if they are not adjacent to any existing aggregate. Remaining unaggregated points are either integrated into already existing aggregates or used to form new aggregates.

Coarsening by aggregation is typically faster than RS-type coarsening. Overall computational effort, however, also depends on the interpolation used, which will be described next.

2.3. Interpolation

Given a fine grid Ω^k and a coarse grid Ω^{k+1}, transfer operators between the two grids need to be defined. The transfer from the fine grid to the coarse grid is commonly called *restriction*, while the transfer from the coarse grid to the fine grid is referred to as *interpolation* or *prolongation*. In order to preserve an eventual symmetry of the initial system matrix also on coarser levels, the restriction matrix is commonly chosen to be the transpose of the interpolation matrix. For this reason, it is sufficient to deal with interpolation in the following.

An interpolation of the F-point i is given by

$$e_i = \sum_{j\in C_i} w_{i,j}e_j \ , \qquad (3)$$

where C_i denotes the set of coarse points influencing i. Since classical AMG assumes an algebraically smooth error, $Ae = r = Au - f \approx 0$ is required. Thus, the i-th equation becomes

$$a_{i,i}e_i + \sum_{j\in N_i} a_{i,j}e_j = 0 \ , \qquad (4)$$

where N_i denotes the neighborhood of the point i, i.e. the column indices of all nonzero entries in row i. In the following, different choices for the weights $w_{i,j}$ are discussed, leading to different interpolation operators. Since proper interpolation operators also depend on the underlying coarse grid, interpolation strategies for RS-type coarsenings and aggregation-based coarsenings are discussed separately.

In order to keep computational effort low, interpolation truncation can be employed. All interpolation weights below a relative threshold of ε are replaced by zero, and other nonzero weights are modified such that consistency of the interpolation is preserved. This results in a sparser matrix at the coarser level, reducing the subsequent computational effort.

2.3.1. RS-type Interpolation

The interpolation described in [17], termed *classical interpolation* in the following, is based on dividing the neighborhood N_i of a point i into the set of coarse points C_i, strongly influencing neighbors F_i^{s} and weakly influencing neighbors F_i^{w}. With condition (C1) one obtains for the interpolation weights

$$w_{i,j} = -\frac{1}{a_{i,i} + \sum_{k\in F_i^{\mathrm{w}}} a_{i,k}} \left(a_{i,j} + \sum_{k\in F_i^{\mathrm{s}}} \frac{a_{i,k}a_{k,j}}{\sum_{m\in C_i} a_{k,m}} \right) \ . \qquad (5)$$

Note that this formula fails whenever condition (C1) is violated.

A simpler expression for the weights $w_{i,j}$ is *direct interpolation*, which can also be used if (C1) is violated:

$$w_{i,j} = -\left(\frac{\sum_{k\in N_i} a_{i,k}}{\sum_{l\in C_i} a_{i,l}} \right) \frac{a_{i,j}}{a_{i,i}} \ . \qquad (6)$$

In general, direct interpolation leads to worse convergence rates than classical interpolation.

More information than for classical and direct interpolation is used for *standard interpolation*, which uses an extended neighborhood, but is otherwise similar to direct interpolation. Multipass interpolation is used for coarsening procedures such as aggressive coarsening, where condition (C1) is not fulfilled, thus leading to F-points without neighboring C-point. It uses direct interpolation for all F-points with a neighboring C-point, otherwise standard interpolation is employed.

2.3.2. Interpolation for Aggregation

In an aggregation-based AMG method, all points in an aggregate are identified with the same value. Consequently, $w_{i,j}$ is equal to one if j is the root node of the aggregate, and zero otherwise. The advantage of this interpolation is that it is simple to implement and very fast. However, as only one C-point is used for the interpolation, the approximation is often worse compared to other approaches, leading to smaller convergence rates.

Smoothed aggregation employs an additional smoothing step such that a broader interpolation base consisting of values also from other aggregates is used. This leads to additional computational effort than for standard aggregation-based AMG, but typically results in better convergence rates. For further details we refer to [18].

3. AMG METRICS

The execution time of an AMG solver depends to a certain extent on the implementation and the underlying hardware. In order to compare different AMG methods not only with respect to execution time, but also independent of the implementation and hardware, several metrics employed in the AMG literature are introduced in this section. However, before introducing the individual metrics, the computational effort of the AMG setup phase is analyzed. This effort is typically dominant when compared to the application of restriction and interpolation operators during the actual solution process.

After the selection of a coarsening procedure, for which the computational effort was already indicated in Sec. 2., the restriction and prolongation operators R^k and P^k need to be set up. On the finest level this effort is typically comparable to the time spent on setting up the system matrix A and mostly depends on the number of coarse points considered for the interpolation of each point. This is particularly true if A stems from a low-order finite element or finite volume discretization.

Given R^k and P^k for the transfer between Ω^k and Ω^{k+1}, the next step is to compute the Galerkin operator A^{k+1} on Ω^{k+1}, which is obtained from the operator A^k on Ω^k by

$$A^{k+1} = R^k A^k P^k \ . \tag{7}$$

Extra care needs to be taken during the implementation of the matrix products, since naive implementations may lead to an accidental higher complexity [8].

We now turn to a discussion of AMG metrics used for the benchmarks in Sec. 4.

- **Number of Coarse Levels.** With every additional coarse level, additional transfer operators R^k and P^k as well as the coarse system matrix A^k needs to be set up. Consequently, it is desirable to keep the number of coarse levels small and thus reduce the overall computational effort.

- **Number of Solver Iterations.** The required number of iterations is a common figure of merit for AMG, either used as preconditioner or as standalone solver. As often reported in the literature and also confirmed by the benchmark results in Sec. 4., only a low number of iterations are commonly required for AMG. Thus, a single additional iteration leads to an increased execution time of about 10 percent for the solver phase.

- **Operator Complexity.** Operator complexity is defined as the ratio of the sum of all entries on all levels k compared to the number of entries of the initial system matrix A on the finest level. In addition to the stencil size, this measure also takes the number of points at each level into account.

- **Maximum Stencil Size.** While geometric multigrid preserves the average stencil size of the system matrix on all coarse levels, this is not necessarily the case for AMG. Therefore, matrices A^k on coarser levels become denser, leading to larger sets of neighboring nodes on average and thus increasing computational effort. Therefore, a smaller maximum stencil size leads to less computational effort in each solver cycle.

4. BENCHMARK RESULTS

The following AMG methods are compared in the following: Classical interpolation used with RS coarsening (*rs_classic*), direct interpolation used with RS coarsening (*rs_direct*), RS one-pass coarsening (*rsop_direct*), RS0 coarsening (*rs0_direct*) and RS3 coarsening (*rs3_direct*). Aggregation-based coarsening is employed with basic interpolation (*ag*) and with smoothed aggregation interpolation (*sa*). While many more variations of AMG are available, our selection is such that the main flavors of AMG methods are included. Coarse levels are constructed until a maximum of 50 points are obtained on the coarsest level. Three pre- and postsmooth Jacobi iterations with a relaxation factor $\omega = 0.67$ are employed. The threshold parameters are $\theta = 0.25$ for RS-based coarsening and $\theta = 0.08/2^{k-1}$ at the k-th level of aggregation-based coarsening. An interpolation truncation weight of $\varepsilon = 0.2$ is used for direct and classical truncation. The weight for smoothed aggregation interpolation is chosen to $\omega = 0.67$. All AMG methods are used as a preconditioner for a conjugate gradient (CG) solver in the case of symmetric matrices, and for a stabilized bi-conjugate gradient (BiCGStab) solver for the nonsymmetric case [1].

Benchmarks are carried out on a workstation equipped with an INTEL Core i7-960, 12 GB of main memory and a NVIDIA Geforce GTX 470 with driver version 270.41.19.

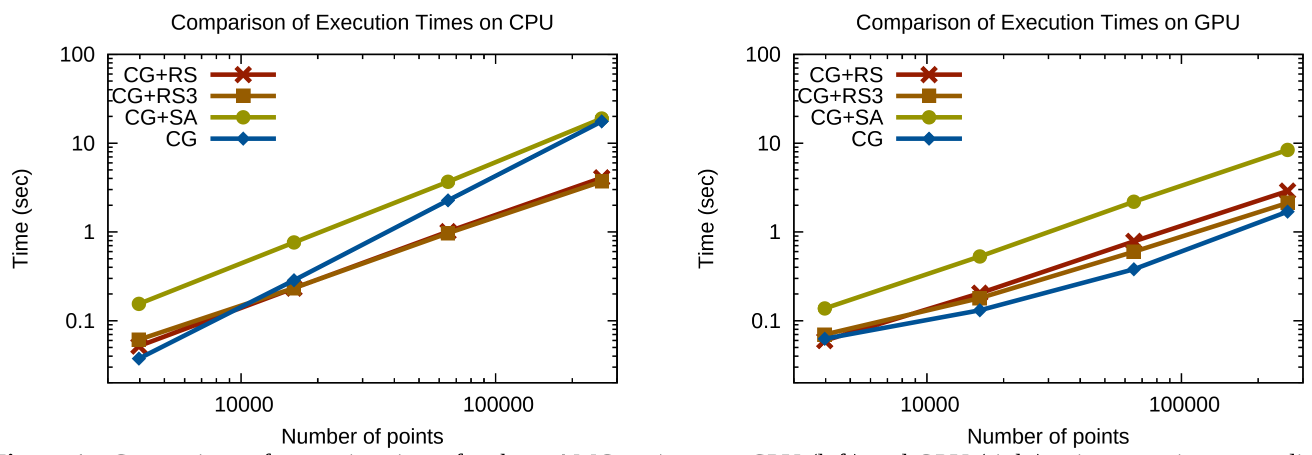

Figure 1. Comparison of execution times for three AMG variants on CPU (left) and GPU (right) using a conjugate gradient (CG) solver. For comparison, execution times for an unpreconditioned CG solver are given. Except for very small problems, RS3 leads to shortest execution times on the CPU, while the unpreconditioned CG solver still provides best results for the considered problem sizes on the GPU.

Method \ Unknowns	**3 969**	**16 129**	**65 025**	**261 121**
rs_classic (CPU)	0.052 (0.031;6)	**0.233** (0.141;6)	1.011 (0.604;6)	4.069 (2.219;6)
rs_classic (GPU)	**0.060** (0.037;6)	0.204 (0.171;6)	0.785 (0.716;6)	2.890 (2.704;6)
rs3_direct (CPU)	0.061 (0.031;8)	0.234 (0.124;7)	**0.969** (0.459;8)	**3.705** (1.756;8)
rs3_direct (GPU)	0.070 (0.032;8)	0.180 (0.130;7)	0.600 (0.494;8)	2.119 (1.870;8)
sa (CPU)	0.155 (0.092;8)	0.761 (0.355;9)	3.675 (1.432;12)	18.975 (5.677;16)
sa (GPU)	0.138 (0.084;8)	0.531 (0.419;9)	2.192 (1.795;12)	8.388 (6.481;16)
cg (CPU)	**0.037** (-;192)	0.286 (-;355)	2.265 (-;672)	17.564 (-;1 262)
cg (GPU)	0.063 (-;192)	**0.131** (-;355)	**0.380** (-;672)	**1.695** (-;1 262)

Table 1. Total execution time for a two-dimensional finite element discretization of the Poisson equation. Total execution times in seconds are given. Setup times and the number of solver iterations are given in parentheses. For the CPU-based execution, the unpreconditioned CG is fastest for the smallest matrix, while classic RS coarsening and RS3 coarsening are fastest for larger matrices. The unpreconditioned CG solver is essentially the fastest if GPU acceleration is employed.

The operating system is a 64-bit Funtoo Linux with kernel version 2.6.38. The AMG setup phase is accelerated on the CPU using OpenMP, while the GPU is employed in the solver cycle phase. Due to the need for frequent dynamic memory allocations, no GPU support is used during the setup phase. In addition, data transfer and compute kernel launch overhead soon becomes too large for achieving significant performance gains especially at coarser levels. Even in the highly optimized setting of an aggregation-based AMG method [8], the setup phase could only be accelerated by a GPU by a factor of two compared to a multi-core CPU, while an acceleration by a factor of six was obtained for the solver cycling phase.

Total execution times for a two-dimensional finite element discretization of the Poisson equation using different grids are depicted in Fig. 1. While the case of a purely CPU-based execution shows the optimal linear scaling of AMG with the problem size, a higher asymptotic complexity of $O(N^{1.5})$ with the number of unknowns N is obtained for an unpreconditioned CG solver. Even though asymptotics on the GPU are the same, interesting differences to the CPU case are observed for small to medium sized problems. OpenCL kernel launch overhead dominates the CG solver for small problems below about 10^4 unknowns. Between 10^4 and 10^5 unknowns, OpenCL kernel launch overheads become less dominant and thus improve performance, while at the same time a higher number of iterations are required. This leads to a nearly linear dependence of the execution times on the problem size in this regime. Above 10^5 unknowns, OpenCL kernel launch overheads do not play a role any longer and a $O(N^{1.5})$ dependence of the unpreconditioned CG solver is retained. Nevertheless, due to the high performance of GPUs, the unpreconditioned CG solver is fastest up to problem sizes of about 10^6, while the use of AMG methods already starts to pay off in a purely CPU-based environment at problem sizes of 10^4.

The benchmark results in Tab. 1 further show that the use of GPU acceleration reduces the solver cycle time by up to

Method	**poisson2d**	**poisson3d**	**navier-stokes**
rs_direct (CPU)	1.064 (0.609;6)	5.161 (1.777;10)	5.476 (1.501;13)
rs_direct (CPU)	0.794 (0.724;6)	1.940 (2.587;10)	1.640 (2.393;13)
rsop_direct (CPU)	1.123 (0.632;6)	**1.947** (0.806;12)	**1.631** (0.480;21)
rsop_direct (GPU)	0.772 (0.702;6)	**1.090** (0.854;12)	**0.789** (0.508;21)
rs0_direct (CPU)	1.192 (0.434;13)	4.460 (1.540;10)	4.777 (1.154;17)
rs0_direct (GPU)	**0.586** (0.452;13)	2.354 (1.730;10)	2.166 (1.319;17)
rs3_direct (CPU)	**0.988** (0.470;8)	6.215 (2.132;10)	6.395 (1.555;16)
rs3_direct (GPU)	0.591 (0.486;8)	3.298 (2.427;10)	2.922 (1.801;16)

Table 2. Total execution time for three different linear systems. Total execution times in seconds are given. Setup times and the number of solver iterations are given in parentheses. Due to the short setup times, one-pass RS coarsening is the fastest method using CPU and GPU for the three-dimensional problems despite the additional solver iterations. For the two-dimensional Poisson equation, RS3 and RS0 are fastest on the CPU and GPU respectively.

Method	poisson2d	poisson3d	navier-stokes
rs_direct	(6; 2.2; 10)	(8; 7.2; 327)	(8; 7.2; 297)
rsop_direct	(6; 2.2; 9)	(5; 2.4; 64)	(4; 1.7; 47)
rs0_direct	(6; 2.2; 14)	(8; 7.3; 300)	(8; 6.3; 260)
rs3_direct	(7; 2.3; 25)	(9; 10.2; 377)	(9; 8.7; 326)

Table 3. Number of coarse levels, operator complexity and maximum stencil size for the three different linear systems considered for the benchmark in Tab. 2.

one order of magnitude. Consequently, total execution time of AMG methods is mostly given by the setup time when GPUs are used. In a pure CPU setting, solver cycle times still contribute more than half of the total execution time on average. Therefore, we conclude that GPUs lead to a shift of emphasis towards the AMG setup phase, hence additional solver iterations are much less a concern than in a pure CPU setting.

With the AMG setup time identified as the main bottleneck in a GPU environment, we now turn to a more detailed comparison of setup times for different coarsening strategies. Three different matrices are compared: The two-dimensional finite element discretization with 65 025 unknowns from the previous benchmark (*poisson2d*), a three-dimensional finite element discretization of the Poisson equation with 31 713 unknowns (*poisson3d*), and a system of 24 202 unknowns obtained from the three-dimensional discretization of the time-dependent Navier-Stokes equations using a Backward Euler scheme (*navier-stokes*). The results in Tab. 2 again confirm that shortest setup times finally yield smallest total execution times. For the two-dimensional Poisson equation, RS0 leads to shortest setup times and shortest execution times on GPUs, because the additional solver iterations required compared to other AMG methods do not significantly contribute to the overall time budget. On CPUs, however, the additional effort spent on the introduction of additional coarse points at block boundaries pays off due to the smaller number of solver iterations required. One-pass RS coarsening leads to shortest

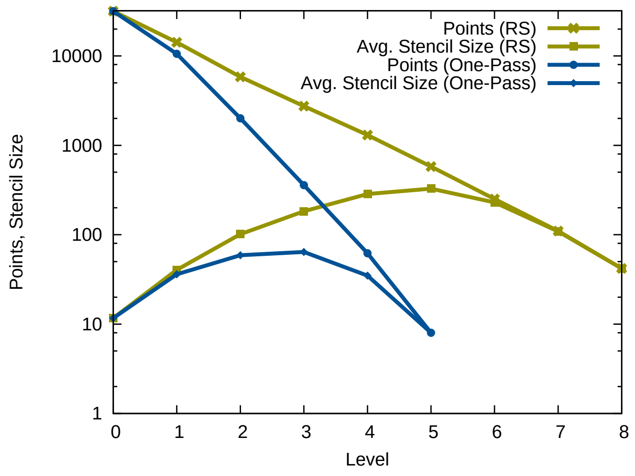

Figure 2. Comparison of RS and one-pass coarsening for the three-dimensional Poisson equation benchmark problem. The number of points and the average stencil size are plotted for each coarse level. It can be seen that less coarse levels considerably reduce the maximum stencil size.

execution for the solution of the three-dimensional Poisson equation and the Navier-Stokes equations both for CPUs and GPUs, even though the highest number of solver iterations are required. Thus, for all three systems shortest setup times lead to shortest overall execution times if GPU acceleration is used. Moreover, our benchmark results show that a large difference in total execution times is obtained for the various AMG methods, hence we expect that it is insufficient to tune only a single AMG method for GPUs when aiming at good performance for a large class of problems.

A comparison of the metrics defined in Sec. 3. for the three linear systems is given in Tab. 3. The number of solver iterations given in Tab. 1 and Tab. 2 is clearly not suitable as a metric for performance of AMG methods when GPU acceleration is employed, because fastest total execution times are obtained for methods with highest iteration count. The three-dimensional benchmarks *poisson3d* and *navier-stokes* suggest that the number of coarse levels, operator complexity, and average stencil size are related metrics. Particularly, a low operator complexity for fast setup times requires a small number of coarse levels, which in turn keeps the average stencil size small.

The number of points as well as the average stencil size at each level are plotted in Fig. 2. While the average stencil size initially increases at the same rate, the faster coarsening of one-pass RS coarsening leads to an earlier saturation of the average stencil size, which in turn reduces the computational effort considerably. In contrast, RS coarsening leads to essentially dense matrices at the coarsest levels, thus increasing the overall computational effort substantially even though a much smaller number of unknowns than for the initial system is encountered.

5. OUTLOOK AND CONCLUSION

With the broad availability of GPUs with high computational power on average workstations, a reevaluation of algebraic multigrid methods is required. While execution times for setup and solver phase need to be somewhat balanced in a purely CPU-based setting, execution times for the solver phase become much less pronounced if GPU acceleration is used. Even though GPU acceleration could also be employed for the setup phase, results by Bell *et al.* [8] suggest that only mild performance improvements can be obtained there. Consequently, research efforts need to be focused on the AMG setup phase in order to reduce setup times, possibly at the cost of higher solver iterations. Clearly, linear scaling of execution times with respect to the number of unknowns should still be preserved.

Our benchmark results show that smallest overall execution times with GPU acceleration is obtained for AMG methods with smallest setup time. Moreover, for small to medium sized problems, unpreconditioned iterative solvers on GPUs are considerably more attractive than in a purely CPU-based environment despite of a high number of solver iterations required.

REFERENCES

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems, Second Edition*, SIAM (2003).

[2] U. Trottenberg *et. al.*, *Multigrid*. Academic Press (2001).

[3] Hypre.
`http://acts.nersc.gov/hypre/`

[4] Algebraic Multigrid Solvers in Python (PyAMG).
`http://www.pyamg.org/`

[5] Algebraic Multigrid Methods for Systems (SAMG).
`http://www.scai.fraunhofer.de/samg/`

[6] U. M. Yang, Parallel Algebraic Multigrid Methods - High Performance Preconditioners. In: *Numerical Solution of Partial Differential Equations on Parallel Computers*, p. 209-236, Springer (2006).

[7] A. J. Cleary *et al.*, Robustness and Scalability of Algebraic Multigrid. *SIAM Journal on Scientific Computing*, vol. 21, p. 1886–1908 (2000).

[8] N. Bell *et al.*, Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods. NVIDIA Technical Report NVR-2011-002 (2011).

[9] NVIDIA CUDA.
`http://www.nvidia.com/`.

[10] OpenCL.
`http://www.khronos.org/opencl/`.

[11] MAGMA Library.
`http://icl.cs.utk.edu/magma/`

[12] Cusp Library.
`http://code.google.com/p/cusp-library/`

[13] Vienna Computing Library (ViennaCL).
`http://viennacl.sourceforge.net/`.

[14] P. S. Vassilevski, *Multilevel Block Factorization Preconditioners*, Springer (2008).

[15] M. J. Grote and T. Huckle, Parallel Preconditioning with Sparse Approximate Inverses. *SIAM J. Sci. Comp.*, vol. 18, no. 3, p. 838–853 (1997).

[16] V. Heuveline *et al.*, Parallel Smoothers for Matrix-based Multigrid Methods on Unstructured Meshes Using Multicore CPUs and GPUs. *EMCL Preprint Series*, no. 9 (2011)

[17] J. W. Ruge and K. Stüben, Algebraic Multigrid (AMG). In S. F. McCormick (editor): *Multigrid Methods*, SIAM, p. 73-130 (1987).

[18] P. Vanek *et al.*, Algebraic Multigrid By Smoothed Aggregation For Second And Fourth Order Elliptic Problems. *Computing*, vol. 56, no. 3, p. 179-196 (1996).