Electronic Transactions on Numerical Analysis.
Volume 55, pp. 687–705, 2022.
Copyright © 2022, Kent State University.
ISSN 1068–9613.
DOI: 10.1553/etna_vol55s687

ETNA
Kent State University **and**
Johann Radon Institute (RICAM)

PORTING AN AGGREGATION-BASED ALGEBRAIC MULTIGRID METHOD TO GPUS[*]

ABDESELAM EL HAMAN ABDESELAM[†], ARTEM NAPOV[†], AND YVAN NOTAY[†]

Abstract. We present a hybrid GPU-CPU version of the AGMG software, a popular algebraic multigrid (AMG) solver which implements an aggregation-based AMG method. With the new implementation, the solution stage runs on a GPU, except operations on the coarsest grid, which are executed on a CPU. To maximize the speedup, two novel features are introduced. On the one hand, ℓ_1-Jacobi smoothing is combined with polynomial acceleration (or polynomial smoothing), leading to improved performance compared with standard ℓ_1-Jacobi smoothing, while not requiring to compute eigenvalue estimates as standard polynomial smoothing does. On the other hand, besides the K-cycle used in standard AGMG, we introduce the relaxed W-cycle, which tends to combine the advantages of the K-cycle and the standard W-cycle. Numerical results show that the new implementation inherits the robustness of the original AGMG software, while bringing significant speedups on GPUs. A comparison with AmgX, a reference AMG solver from NVIDIA, suggests that the presented hybrid GPU-CPU version of AGMG is more robust and often significantly faster in the solution stage.

Key words. multigrid, linear systems, iterative methods, AMG, preconditioning, parallel computing, GPU

AMS subject classifications. 65F10, 65N22, 65Y05, 65Y10

1. Introduction.

We present a hybrid GPU-CPU version of the algebraic multigrid (AMG) method from the AGMG [17] software package for the iterative solution of large sparse systems of linear equations

$$A\mathbf{x} = \mathbf{b},$$

where A is an $n \times n$ matrix, $\mathbf{x}$ is the vector of unknowns, and $\mathbf{b}$ is the right-hand side vector, both being of dimension n. AMG methods are the methods of choice for systems arising from the discretization of scalar elliptic PDEs [5, 26]. In particular, they often exhibit optimal convergence properties in that the number of iterations is bounded independently of the mesh size, and, unlike geometric multigrid methods, they are of black-box type.

The use of graphics processing units (GPUs) is increasingly popular in scientific computing [8, 12, 25]. Porting well-established numerical methods to GPUs is therefore an important research topic. In this regard, several works focus on the development of AMG solvers for GPUs, e.g., [9] discusses strategies and experiences for porting to GPUs the solvers from the *hypre* software, including AMG solvers [11, 30], while in [16] the AmgX[1] solver from NVIDIA is presented, a reference solver specifically developed for GPUs. Further, block-asynchronous smoothers are studied in [2].

Here, we consider the portage to GPUs of the AMG method implemented in the AGMG software [17]. As any multigrid method, it is based on the recursive use of a two-grid method, which itself is a combination of a *smoother*, typically a greedy iterative method, and a *coarse-grid correction*, which amounts to solving a related but smaller (or coarser) system. A multigrid method is obtained when the coarser system is in turn solved approximately with one or few iterations of the multigrid method, implying thus its recursive use. The iterative

[*]Received October 3, 2021. Accepted May 9, 2022. Published online on August 23, 2022. Recommended by Maya Neycheva. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 824158. It also benefited from the support of the Fonds de Recherche Scientifique-FNRS (Belgium) under grant No J.0084.16.

[†]Université Libre de Bruxelles, Service de Métrologie Nucléaire (C.P. 165-84), 50 Av. F.D. Roosevelt, B-1050 Brussels, Belgium ({Abdeselam.El.Haman.Abdeselam, Yvan.Notay, Artem.Napov}@ulb.be). Yvan Notay is Research Director of the Fonds de la Recherche Scientifique–FNRS.

[1]https://github.com/NVIDIA/AMGX

scheme used for this approximate solution defines the *multigrid cycle*. The recursion stops for the coarsest (bottom) system, for which a different solver is used.

The resulting set of progressively smaller systems forms the multigrid hierarchy, and in AMG methods such a hierarchy is constructed based solely on the system matrix, implying the black-box nature. The hierarchy construction corresponds to the *setup* phase, whereas the subsequent use of the hierarchy in the solution process is referred to as the *solve* phase.

In this work we focus on the *solve* phase of the considered AMG solver. On the one hand, when several linear systems have to be solved with the same system matrix, the setup stage is performed only once, and its cost is amortized over multiple solves. Hence, accelerating the setup phase is relatively less important, especially when hundreds or thousands of solves are needed as may occur when solving fluid problems with a pressure-correction technique.

On the other hand, a straightforward porting of the AGMG setup phase to GPU may do more harm than good. This is related to the specific design of the AGMG setup stage: the corresponding aggregation scheme [14] aims at building aggregates with guaranteed quality [13], which makes the associated AMG solver particularly robust. However, the resulting design is sequential by nature, and it seems hard to obtain a decent level of GPU parallelism without compromising the quality of the aggregates and, ultimately, robustness. Therefore, in the version presented here, the setup phase is still ran on CPUs with standard AGMG.

We now briefly recall the main ingredients of the AGMG solve phase. Regarding the smoothing iterations, AGMG, like most AMG methods, uses a simple Gauss-Seidel method. The coarse-grid correction is of aggregation type, while the multigrid recursion is based on the K-cycle [24], which amounts to 2 multigrid iterations on the coarse level combined with Krylov subspace acceleration. Krylov subspace acceleration is also used at the fine-grid level, the AMG method being used as a preconditioner for the flexible conjugate gradient (FCG) method [18] in the case where the system matrix is symmetric and positive definite (SPD) and by the GCR method [7] otherwise. Finally, the bottom-level solver is a sparse direct solver.

Each of the above mentioned ingredients induces some difficulties on GPUs, leading us to propose significant changes in the solve phase. Firstly, the Gauss-Seidel iterative method is not adapted to parallel architectures, hence we developed a specific smoother that combines ℓ_1-Jacobi smoothing [4] and polynomial acceleration [1, 10, 30]. To the best of our knowledge, such a combination is considered here for the first time.

Secondly, the K-cycle requires the computation of dot products, which may be suboptimal on GPUs [29]. This leads us to propose a *relaxed W-cycle*, which aims at preserving the robustness of the K-cycle, while avoiding dot product computation.

Finally, sparse direct solvers are not efficient on GPUs, and more generally, the size of the coarsest system is typically too small to make a reasonable usage of GPUs capabilities. This leads us to transfer the coarsest system on the CPU, using thus a CPU bottom-level solver. While a straightforward choice would then consist in using a sparse direct solver, we observe that even better results are obtained using instead a single iteration of the standard (CPU) version of AGMG. Since the setup and the bottom solve are performed on the CPU, while the remaining runs on the GPU, we resort to the “hybrid GPU-CPU” qualification for the presented method.

Numerical results show that with the adaptations mentioned above, the presented approach preserves the robustness of the standard AGMG software while exhibiting considerable speedup in the solve phase. The comparison with AmgX reveals that this latter is less robust and often slower in the solve phase when using coarsening by aggregation, that is, a technology similar to the one used in AGMG. On the other hand, the solve phase with AmgX may be faster when using classical AMG coarsening [26], but this variant appears overall even less robust. These latter results give an additional argument in favor of keeping the quality-based aggregation scheme on the CPU, since this scheme represents the most significant conceptual difference between the aggregation-based version of AmgX and the presented approach.

Eventually, let us point out that, for the sake of clarity, this work is focused on SPD systems. On the one hand, in this case, we can provide theoretical foundations to the original ingredients presented here, i.e., the polynomial acceleration of ℓ_1-Jacobi smoothing and the relaxed W-cycle. On the other hand, for this case, we have a comprehensive test suite inherited from previous works [14, 15, 20], allowing us to challenge the robustness of the solver but also the efficiency of the GPU implementation, as the matrices in this suite exhibit quite diverse connectivity patterns, some associated with low-order discretizations and/or regular meshes and some with higher-order discretizations (up to fourth-order finite elements) and/or unstructured meshes.

In the numerical experiments, we focus on linear systems from scalar elliptic PDEs, which form the main class of applications of the AGMG software. Of course, solving such systems is fast enough on CPUs. Obtaining further speedup thanks to GPUs is nevertheless useful in several contexts. When linear systems are solved repeatedly during a simulation process (e.g., because of time stepping), the total amount of time spent in the linear system solver can be large even though the time spent in solving each individual system is reasonable. On the other hand, numerical simulations are more and more performed interactively by engineers who dynamically adapt parameters based on the obtained results; in such cases, having the answer in, say, 1 second instead of 10 has a significant impact on the user's experience.

The remainder of this paper is organized as follows. In Section 2 we detail the porting of the solve phase to GPUs and discuss the key modifications brought to achieve a proper parallelism. The results of numerical experiments are reported in Section 3 and conclusions are drawn in Section 4.

2. Porting.

For SPD systems, AGMG implements the FCG method with multigrid preconditioning. In the hybrid GPU-CPU version, we use a variant of the FCG algorithm as described in [23], which minimizes the number of synchronization points. Besides operations associated with the preconditioner, the needed operations are sparse matrix-vector products for which we use functions from the cuSPARSE library, and vector operations, for which we use functions from the cuBLAS library.

On the other hand, the multigrid preconditioning amounts to the recursive application of Algorithm 1 given below, which is initially called at the fine-grid level $k = 1$ with A_1 equal to the system matrix A and $\mathbf{r}_1$ equal to the current residual of the linear system. This algorithm involves the parameters $\tau_i^{(k)}$, $i = 1, 2, 3$. As discussed in Section 2.2, computing them via some dot products gives the K-cycle, selecting uniformly $\tau_i^{(k)} = 1$ gives the standard W-cycle, whereas the relaxed W-cycle is obtained using uniformly $\tau_i^{(k)} = \tau$ for some $1 < \tau < 2$. How the algorithm navigates between the levels is further illustrated in Figure 2.1.

Pre- and post-smoothing iterations (steps 1 and 7) are discussed in Section 2.1. In between, the residual is transferred to the coarser level $k + 1$ (step 3) by multiplying by the transpose of the prolongation matrix P_k computed during the setup phase. At the coarse level, an approximate solution of the coarse system (with the matrix A_{k+1} also defined during setup) is computed (step 4) using in general two iterations with the same multigrid preconditioner at that level, which is therefore called recursively. The recursion is stopped when one reaches level L, where a bottom-level solver is called; its choice is further discussed in Section 2.3. The correction computed on the coarse level $k + 1$ is afterward prolongated at level k (step 5) by multiplying by the prolongation matrix P_k.

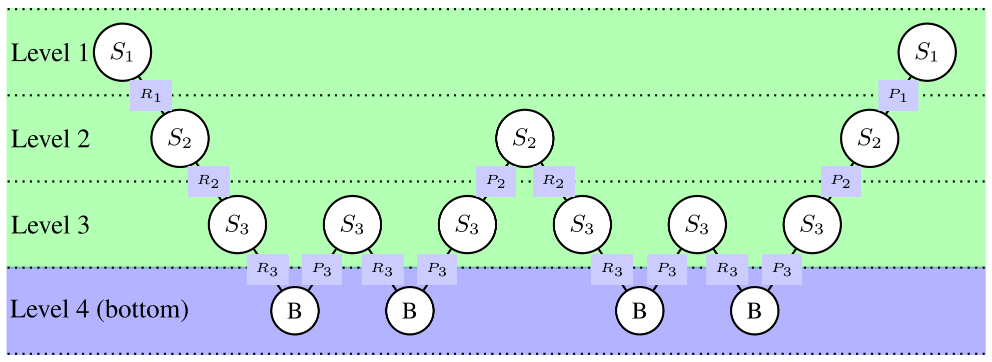
Algorithm 1 Multigrid as a preconditioner at level k $(k \geq 1)$; K- or (relaxed) W-cycle.

```
Calling sequence: $\mathbf{z}_k = \text{MG}_{\text{prec}}(\mathbf{r}_k,\, k)$.
Data: matrix $A_k$, smoother $M_k$, prolongation $P_k$, matrix $A_{k+1}$
Algorithm:
1: Pre-smoothing:
     $\mathbf{x}_k^{(0)} = 0$
     $\mathbf{x}_k^{(\mu+1)} = \mathbf{x}_k^{(\mu)} + \omega_k^{(\mu)}\, M_k(\mathbf{r}_k - A\mathbf{x}_k^{(\mu)}), \quad \mu = 1, \dots, \nu_k$
     $\mathbf{z}_k^{(1)} = \mathbf{x}_k^{(\nu_k)}$
2: Compute new residual: $\widetilde{\mathbf{r}}_k = \mathbf{r}_k - A_k\, \mathbf{z}_k^{(1)}$.
3: Restrict residual: $\mathbf{r}_{k+1} = P_k^T \widetilde{\mathbf{r}}_k$.
4: Compute an approximate solution $\widetilde{\mathbf{x}}_{k+1}$ to $A_{k+1}\, \mathbf{x}_{k+1} = \mathbf{r}_{k+1}$:
     if $k+1 = L$:
        Bottom-level solver: $\widetilde{\mathbf{x}}_{k+1} = \text{BLS}\,(A_{k+1},\, \mathbf{r}_{k+1})$
     else:
        $\mathbf{c}_{k+1} = \text{MG}_{\text{prec}}(\mathbf{r}_{k+1},\, k+1)$
        $\mathbf{v}_{k+1} = A_{k+1}\, \mathbf{c}_{k+1}$
        $\widetilde{\mathbf{r}}_{k+1} = \mathbf{r}_{k+1} - \tau_1^{(k+1)}\, \mathbf{v}_{k+1}$
        $\mathbf{d}_{k+1} = \text{MG}_{\text{prec}}(\widetilde{\mathbf{r}}_{k+1},\, k+1)$
        $\widetilde{\mathbf{x}}_{k+1} = \tau_2^{(k+1)}\, \mathbf{c}_{k+1} + \tau_3^{(k+1)}\, \mathbf{d}_{k+1}$
5: Prolongate coarse-grid correction: $\mathbf{z}_k^{(2)} = P_k \widetilde{\mathbf{x}}_{k+1}$.
6: Compute new residual: $\bar{\mathbf{r}}_k = \widetilde{\mathbf{r}}_k - A_k\, \mathbf{z}_k^{(2)}$.
7: Post-smoothing:
     $\widetilde{\mathbf{x}}_k^{(0)} = 0$
     $\widetilde{\mathbf{x}}_k^{(\mu+1)} = \widetilde{\mathbf{x}}_k^{(\mu)} + \omega_k^{(\mu)}\, M_k(\bar{\mathbf{r}}_k - A\widetilde{\mathbf{x}}_k^{(\mu)}), \quad \mu = 1, \dots, \nu_k$
     $\mathbf{z}_k^{(3)} = \widetilde{\mathbf{x}}_k^{(\nu_k)}$
8: $\mathbf{z}_k = \mathbf{z}_k^{(1)} + \mathbf{z}_k^{(2)} + \mathbf{z}_k^{(3)}$
```

Regarding the GPU implementation of the operations in Algorithm 1, first note that, as discussed in Section 2.1, the matrices M_k are in fact diagonal. Hence, except for restriction (step 3) and prolongation (step 5), all operations are straightforwardly ported using cuBLAS for vector operations and cuSPARSE for matrix-vector products with the matrices A_k.

The restriction and prolongation operations (steps 3 and 5) were implemented with custom kernels. Let us first recall that with aggregation-based AMG, P_k are Boolean matrices with at most one nonzero entry per row, whereas, with algorithms using successive pairwise aggregations [14, 20], there exits a known and tight upper bound for the number of nonzero entries per column according to the number of performed pairwise passes. That said, both restriction and prolongation were implemented in a similar manner assigning a thread per variable of the coarser grid, i.e., per column in the matrix P_k ; for each such variable j on the coarser grid, one just needs to store the list of associated fine-grid unknowns, i.e., the list of row indexes i for which $(P_k)_{ij} = 1$.

In the following sections we discuss the differences introduced in Algorithm 1 compared with what is used in standard AGMG. Namely, the smoothing is discussed in Section 2.1, the multigrid cycles are considered in Section 2.2, and the bottom-level solver is addressed in



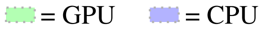


FIG. 2.1. *Levels traversal of Algorithm 1 in the case of $L = 4$ levels. S_k stands for smoothing and related operations at level k (steps 1–2, 6–8), R_k stands for the restriction operation (step 3), P_k stands for the prolongation operation (step 5), and B stands for the bottom-level solve; as discussed in Section 2.3, this latter is performed on the CPU.*

Section 2.3.

2.1. Smoother. Gauss-Seidel iterations are the smoothers of choice for sequential AMG methods [5]. They require no additional parameters while often exhibiting attractive smoothing properties. AGMG uses as default one forward Gauss-Seidel sweep for pre-smoothing and one backward sweep for post-smoothing.

Nevertheless, Gauss-Seidel iterations are not convenient for a parallel implementation. Richardson and damped Jacobi iterations are much more suitable to run on GPUs. However, they require a damping parameter which depends on the largest eigenvalue of the system matrix (Richardson) or on the largest eigenvalue of the system matrix preconditioned by its diagonal (Jacobi). For the set of test problems considered in Section 3, these largest eigenvalues with diagonal preconditioning range from 2 to 3.4, so that it seems difficult to fix a uniform value for the damping parameter that could be both robust and near optimal in all cases.

Therefore, in this work, we opt for a ℓ_1-Jacobi smoothing [4], which amounts to using, for the matrix $A_k = (a_{ij}^{(k)})$,

$$M_k = \operatorname{diag}\left(1 \Big/ \sum_{j=1}^{n} |a_{ij}^{(k)}|\right) . \tag{2.1}$$

Standard ℓ_1-Jacobi smoothing uses no relaxation, i.e., steps 1 and 7 are implemented with $\omega_k^{(\mu)} = 1$. Observe that

$$\|M_k A_k\|_\infty = \max_i \sum_{j=1}^{n} \frac{|a_{ij}^{(k)}|}{\sum_{j=1}^{n} |a_{ij}^{(k)}|} = 1 .$$

Hence, if A_k is SPD, the eigenvalues of $M_k A_k$ are in the interval $(0, 1]$, implying that these standard ℓ_1-Jacobi smoothing iterations are always convergent. Note also that if the fine-grid matrix A is SPD, then the used coarsening scheme ensures that the matrices A_k are SPD at every level.

In comparison, standard Jacobi smoothing uses $M_k^{(\mathrm{Jac})} = \omega_{\mathrm{Jac}} \operatorname{diag}\left(1/a_{ij}^{(k)}\right)$, where ω_{Jac} is a damping parameter. For scalar elliptic PDEs, low-order finite difference or finite element discretizations often lead to matrices satisfying $\sum_{j=1}^n |a_{ij}| \approx 2\, a_{ii}$; this is because off-diagonal entries are nonnegative while the row-sum is zero everywhere except near boundaries. Then, there is no significant difference between ℓ_1-Jacobi and standard Jacobi smoothing with $\omega_{\mathrm{Jac}} = 1/2$. Moreover, if ℓ_1-Jacobi smoothing is combined with relaxation or polynomial acceleration (see below), then again similar results can be obtained using standard Jacobi smoothing, adapting the relaxation parameters in a straightforward way. However, for higher-order discretizations, there is a significant difference between ℓ_1-Jacobi and standard Jacobi smoothing, with the main consequence that the largest eigenvalue of $M_k^{(\mathrm{Jac})} A_k$ changes from problem to problem. Therefore, just finding a value ω_{Jac} that would be robust and close to optimal in all cases seems out of reach. A fortiori, using polynomial acceleration with fixed parameter sets as explained below is not possible with standard Jacobi smoothing.

Indeed, here we further exploit the fact that the eigenvalues of $M_k\, A_k$ are known to be in the interval $(0, 1]$ to combine ℓ_1-Jacobi smoothing with polynomial acceleration. That is, we apply the Chebyshev iterative method (see for instance [3]) to the system $A_k\, \mathbf{x}_k = \mathbf{r}_k$ preconditioned with the matrix M_k. This may also be seen as combining ℓ_1-Jacobi smoothing and polynomial smoothers [4], a combination considered here seemingly for the first time. Note, however, that optimizing the weights for the standard Jacobi smoother via Chebyshev polynomials is considered in some early works under the name of "multi-stage Jacobi smoothing" [6, 27].

In Algorithm 1, the polynomial acceleration is reflected by using relaxation factors $\omega_k^{(\mu)} \neq 1$ at steps 1 and 7. Consider, e.g., the first of these steps. The error $\mathbf{e}_k^{(\mu)} = A_k^{-1}\mathbf{r}_k - \mathbf{x}_k^{(\mu)}$ satisfies, after ν_k smoothing iterations,

$$\mathbf{e}_k^{(\nu_k)} \;=\; \Pi_{\mu=1}^{\nu_k} (I - \omega_k^{(\mu)} M_k A_k)\, \mathbf{e}_k^{(0)} \,. \tag{2.2}$$

It means that an eigenmode of $M_k\, A_k$ associated with the eigenvalue λ_i is damped by a factor of

$$\Pi_{\mu=1}^{\nu_k} \left(I - \omega_k^{(\mu)} \lambda_i\right) \;=\; p_{\nu_k}(\lambda_i) \,, \tag{2.3}$$

where p_{ν_k} is a polynomial of degree ν_k such that $p_{\nu_k}(0) = 1$. The smoothing scheme is most efficient when this damping is maximized for all high-frequency modes. A rule of thumb defines these as those associated with eigenvalues in the interval $[a\lambda_{\max},\, \lambda_{\max}]$, where $\lambda_{\max}$ is the largest eigenvalue of $M_k A_k$ and a a fixed fraction, e.g., $a = 0.3$ or $a = 0.25$ [4]. Then, given the number of smoothing steps ν_k, one selects the polynomial that provides optimal damping for the eigenvalues in the corresponding interval, and this fixes the $\omega_k^{(\mu)}$, which are the inverses of the roots of this polynomial. Such optimal polynomials are known to be Chebyshev polynomial [3], whose roots are given by

$$\left(\omega_k^{(\mu)}\right)^{-1} = \frac{\lambda_{\max}}{2} \left((1-a) \cos\left(\tfrac{(2\mu-1)\Pi}{2\nu_k}\right) + 1 + a \right) \,, \qquad \mu = 1, \dots, \nu_k \,. \tag{2.4}$$

A further nice property of these polynomials is that the iteration remains convergent for positive eigenvalues below the lower end $a\lambda_{\max}$ of the selected interval. Hence, the smoothing scheme is relevant as long as $\lambda_{\max}$ is not underestimated.

With classical polynomial smoothing as in [4], this estimation of $\lambda_{\max}$ is crucial and requires additional preprocessing. However, here we combine this approach with ℓ_1-Jacobi smoothing for which one knows that the largest eigenvalue is at most 1 and practically equal to this value. Hence, using $\lambda_{\max} = 1$ is both safe and practically optimal. It remains thus only to

ETNA
Kent State University and
Johann Radon Institute (RICAM)

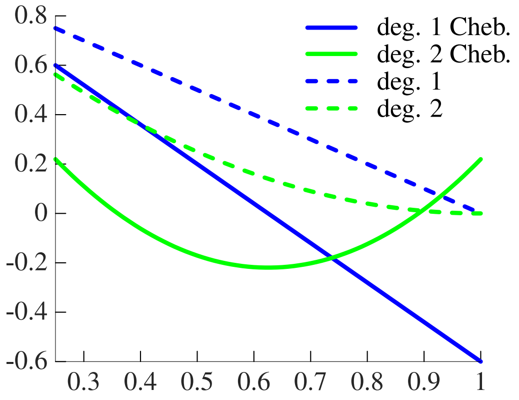


FIG. 2.2. *Factor* (2.3) *for* $\lambda_i \in [a, 1]$ *with* $a = 0.25$*, for degree 1 and 2 polynomials with Chebyshev acceleration* ($\omega_k^{(\mu)}$ *given by* (2.4)*), and without it* ($\omega_k^{(\mu)} = 1$)*.*

select the fraction a. We found that the results were not much sensitive to its value and overall best with $a = 0.25$, which we therefore used uniformly in all experiments reported below.

Regarding the number of smoothing steps, standard AGMG uses a single step for both pre- and post-smoothing. However, Gauss-Seidel is a better smoother than ℓ_1-Jacobi, and using more smoothing steps might therefore be a sensible option. Regarding the inner levels ($k > 1$: all levels but the fine-grid one), it turns out that using more than $\nu_1 = 1$ smoothing steps hardly improves the number of iterations needed by the solver. Note that even with a single step, and thus a polynomial of degree 1, there is a significant difference between the standard ℓ_1-Jacobi iteration and its polynomially accelerated version, in which $\omega_k^{(1)} = \frac{2}{1.25} = \frac{8}{5}$; see also Figure 2.2 for a sketch of the polynomial.

At fine-grid levels, however, we found that the results are overall better using $\nu_1 = 2$ steps for both pre- and post-smoothing (and thus degree-2 polynomials; see Figure 2.2). The main reason is that smoothing iterations at fine-grid levels are among those operations for which GPUs provide the highest speedup. Hence, it is worth performing more such operations if this helps to save on operations on the coarser grids, for which the speedup is lower. Also, it has been observed that $\nu_1 = 2$ at the fine-grid level has more impact on the number of solver iterations than using $\nu_k = 2$ in coarser levels (i.e, for $k = 2, \dots, L-1$).

In Table 2.1 we illustrate the benefit of the smoothing strategy proposed here compared with standard approaches by reporting the number of iterations with several alternative strategies for a sample of our tests problems presented in Section 3. In all cases we ran the same code except for the smoothing; see Section 3 for the details of the experimental setting and the problem description. The number of smoothing iterations was the same in all cases, that is, as written above, 2 pre- and 2 post-smoothing steps at the fine-grid level, and 1 pre- and 1 post-smoothing step at all other levels.

In the last two columns we compare ℓ_1-Jacobi smoothing with and without polynomial acceleration; that is, the proposed method that uses $\omega_k^{(\mu)}$ defined by (2.4) with $\lambda_{\max} = 1$ is compared with the standard method obtained by setting $\omega_k^{(\mu)} = 1$ for all k. One sees that the polynomial acceleration brings a significant benefit in most cases, although the standard method is already robust.

On the other hand, in the first three columns we report the results obtained with standard polynomial smoothing as presented in [4]. This approach may also be described by (2.2) in which one uses $M_k = I$ and $\omega_k^{(\mu)}$ defined in (2.4) with $\lambda_{\max}$ set to the largest eigenvalue of

TABLE 2.1
Number of iterations for AGMG with the K-cycle using standard polynomial smoothing [4] (with different values of the a parameter) or ℓ_1-Jacobi smoothing with (poly. acc.) or without (standard) polynomial acceleration.

		std. polynomial smoother			ℓ_1-Jacobi smoother	
Problem	size	$a = 0.16$	$a = 0.25$	$a = 0.30$	standard	poly. acc.
MOD2D	S1	14	14	14	17	13
	S2	14	15	15	17	14
	S3	14	15	15	18	14
JUMP2D	S1	283	284	271	28	24
	S2	n.c.	n.c.	n.c.	33	28
	S3	n.c.	n.c.	n.c.	35	28
MOD3D	S1	12	15	12	13	10
	S2	21	19	17	14	10
	S3	15	15	15	14	11
SPHERE$_1$	S1	107	109	110	13	12
	S2	n.c.	n.c.	n.c.	11	10
	S3	n.c.	n.c.	n.c.	11	10

A_k. Thus, as additional preprocessing, the method requires the numerical assessment of the largest eigenvalue of the matrix at each level. We present the results for several values of the parameter a according to the discussion in [4] (0.16 is used as an approximation of $1/6$).

The results are relatively similar to those obtained with our approach for the model problems MOD2D and MOD3D. This is to be expected because for such problems the M_k defined by (2.1) do not differ much from a constant diagonal, hence both approaches are close to each other; the main advantage of the ℓ_1-Jacobi smoother being that $\lambda_{\max}$ does not need to be computed. However, standard polynomial smoothing appears significantly less robust as it essentially fails on problems JUMP2D and SPHERE$_1$, which are from PDEs with jumping coefficients (see Section 3). Of course, convergence can likely be restored by using more smoothing steps, especially at the coarse levels, the word "polynomial" suggesting to use more than one step. We did not pursue in this direction since polynomially accelerated ℓ_1-Jacobi smoothing seems anyway definitely more cost effective in the context of AGMG.

2.2. Multigrid cycles. Standard AGMG uses the K-cycle [24], which is known to be robust [14, 20]. It is obtained using Algorithm 1 with $\tau_i^{(k+1)}$, $i = 1, 2, 3$, computed as a function of dot products involving the iteration vectors $\mathbf{c}_{k+1}$, $\mathbf{v}_{k+1}$, etc.; see [20] for algorithmic details. Five dot products are needed to compute all three parameters, hence, if $L \geq 3$, then each application of the multigrid preconditioner at the fine-grid level requires globally $5\,(2^{L-2} - 1)$ dot products. Such operations are not very efficient on GPUs [29], which may suggest to use the standard W-cycle instead. With this, one uses $\tau_i^{(k)} = 1$ throughout, hence no dot product is computed besides those needed at the fine-grid level by the FCG algorithm.

However, the standard W-cycle appears significantly less robust, and using it instead of the K-cycle leads in most cases to an unacceptable increase of the number of iterations [20]. Therefore, motivated by the analysis presented below, we consider the *relaxed* W-cycle, in which one sets uniformly $\tau_i^{(k)} = \tau$ for some fixed $1 < \tau < 2$; since τ is fixed, here again no dot product computation is needed.

To see why this can be useful, consider thus $\tau_i^{(k)} = \tau$, $i = 1, 2, 3$, for some fixed τ, let G_k be the matrix such that $\mathrm{MG}_{\text{prec}}(\mathbf{r}_k,\, k) = G_k\mathbf{r}_k$, and let B_k be the matrix such that step 4 of Algorithm 1 amounts to $\tilde{\mathbf{x}}_{k+1} = B_{k+1}\mathbf{r}_{k+1}$. One sees that, if $k < L$, then $B_k = 2\tau G_k - \tau^2 G_k A_k G_k$. Since B_k is symmetric and A_k is SPD, it follows that $B_k A_k$ is similar to a symmetric matrix. Hence, its eigenvalues are real, and one may verify that its largest and smallest eigenvalues satisfy

$$\sigma(G_k A_k) \subset [\lambda_k,\, 1] \;\Rightarrow\; \begin{cases} \lambda_{\max}(B_k A_k) & \leq\; 1 \\ \lambda_{\min}(B_k A_k) & \geq\; \min\left(2\,\tau\,\lambda_k - \tau^2\lambda_k^2,\; 2\,\tau\, - \tau^2\right)\;. \end{cases} \tag{2.5}$$

Let now G_k^{TG} be the matrix corresponding to the two-grid version of the preconditioner at level k. That is, the matrix such that $\mathrm{MG}_{\text{prec}}^{\text{TG}}(\mathbf{r}_k,\, k) = G_k^{\text{TG}}\mathbf{r}_k$, where $\mathrm{MG}_{\text{prec}}^{\text{TG}}()$ stands for a non-recursive version of Algorithm 1 in which step 4 is exchanged for $\tilde{\mathbf{x}}_{k+1} = A_{k+1}^{-1}\mathbf{r}_k$. Because we use the same scheme for pre- and post-smoothing with an SPD smoother, it is known that (see, e.g., [22, 28])

$$\lambda_{\max}(G_k^{\text{TG}} A_k) \;=\; 1 \qquad \text{and} \qquad \lambda_{\min}(G_k^{\text{TG}} A_k) \;=\; \lambda_k^{TG} \;>\; 0\,. \tag{2.6}$$

On the other hand, Theorem 2.2 in [19] tells us that

$$\begin{aligned} \lambda_{\max}(G_{k-1}A_{k-1}) &\leq \lambda_{\max}(G_{k-1}^{\text{TG}}A_{k-1}) \cdot \max\left(\lambda_{\max}(B_k A_k),\, 1\right), \\ \lambda_{\min}(G_{k-1}A_{k-1}) &\geq \lambda_{\min}(G_{k-1}^{\text{TG}}A_{k-1}) \cdot \min\left(\lambda_{\min}(B_k A_k),\, 1\right). \end{aligned}$$

Combining this with (2.5) and (2.6), one deduces that, for $k < L$,

$$\sigma(G_k A_k) \;\subset [\lambda_k,\, 1] \quad \Rightarrow \quad \sigma(G_{k-1}A_{k-1}) \;\subset [\lambda_{k-1},\, 1]$$

with

$$\lambda_{k-1} \;=\; \lambda_{k-1}^{TG}\; \min\left(2\,\tau\,\lambda_k - \tau^2\lambda_k^2,\; 2\,\tau\, - \tau^2\right)\;. \tag{2.7}$$

This allows us to estimate the eigenvalues of G_1A_1 by following a recursion bottom to top, and the larger the final value λ_1, the better the preconditioner. Note that $\lambda_{L-1} \;=\; \lambda_{L-1}^{TG}$ if the bottom-level solver is a direct solver; otherwise, λ_{L-1} may be estimated using again [19, Theorem 2.2].

Consider now the effect of τ in (2.7). Clearly, one needs $\tau \;<\; 2$, otherwise $2\,\tau\; - \tau^2$ is negative. On the other hand, choosing $\tau \;<\; 1$ yields a less favorable estimate than for $\tau = 1$. However, selecting $\tau = 1$ as the standard W-cycle does not seem very wise as well since this minimizes $2\,\tau\,\lambda_k \;-\; \tau^2\lambda_k^2$ over $1 \;\leq\; \tau \;<\; 2$. One sees that increasing τ yields a smaller value for λ_{k-1} as long as $2\,\tau\; - \tau^2$ does not become too small. In fact, the best would be to balance the two terms in the minimum of the right-hand side of (2.7). This is what the AMLI-cycle implements [28], but this requires to compute τ level by level based on the eigenvalue estimates.

With the relaxed W-cycle, we propose a simpler approach that uses a fixed value of τ. The following theorem details the conditions under which the recursion (2.7) yields a bound on λ_1 independent of the number of levels L.

THEOREM 2.1. *Assume* $1 \leq \tau < 2$, *and let* λ^{TG} *be such that* $0 < \lambda^{TG} \leq \lambda_k^{TG} < 1$ *for* $k = 1, \ldots, L-2$.

(1) If

$$\lambda^{TG} \geq \frac{1}{\tau^2} \qquad \textit{and} \qquad \lambda^{TG}\,(2\tau - \tau^2) \leq \lambda_{\min}(G_{L-1}A_{L-1}) < 1\,,$$

then the recursion (2.7) *yields*

$$1 \;>\; \lambda_k \;\geq\; \lambda^{TG}\,(2\tau-\tau^2) \qquad \textit{for all } k = L-2,\ldots,1.$$

(2) If

$$\frac{1}{2\,\tau} < \lambda^{TG} < \frac{1}{\tau^2} \qquad \textit{and} \qquad \lambda^{TG} - \frac{(\lambda^{TG}-\frac{1}{\tau})^2}{\lambda^{TG}} \leq \lambda_{\min}(G_{L-1}A_{L-1}) < 1\,,$$

then the recursion (2.7) *yields*

$$1 \;>\; \lambda_k \;\geq\; \frac{1}{\tau}\Big(2-\frac{1}{\tau\lambda^{TG}}\Big) \qquad \textit{for all } k = L-2,\ldots,1.$$

Proof. We proceed by induction. We also use the fact that $a \,\geq\, x \,\geq\, b$ implies the inequality $2x-x^2 \geq \min(2a-a^2,\, 2b-b^2)$.

(1): $1 > \lambda_k \geq \lambda^{TG}\,(2\tau-\tau^2)$ holds for $k \,<\, L-1$ by induction, whereas it hold for $k = L-1$ by assumption. With $1 > \lambda^{TG} \geq \frac{1}{\tau^2}$, it implies $\tau > \tau\lambda_k \geq 2-\tau$; therefore, $2\,\tau\,\lambda_k - (\tau\lambda_k)^2 \geq 2\,\tau - \tau^2$, from which one sees that (2.7) yields $1 > \lambda_{k-1} \geq \lambda^{TG}\,(2\tau-\tau^2)$.

(2): $\tau \,>\, \tau\,\lambda_k \,\geq\, 2 - \frac{1}{\lambda^{TG}\,\tau}$ holds for $k \,<\, L-1$ by induction, whereas it holds for $k = L-1$ because $\tau\big(\lambda^{TG} - \frac{(\lambda^{TG}-\frac{1}{\tau})^2}{\lambda^{TG}}\big) = 2 - \frac{1}{\lambda^{TG}\,\tau}$. It implies that

$$2(\tau\,\lambda_k) - (\tau\,\lambda_k)^2 \;\geq\; \min\left(\tfrac{1}{\tau\,\lambda^{TG}}\Big(2-\tfrac{1}{\tau\,\lambda^{TG}}\Big), 2\,\tau\, - \tau^2\right) \;=\; \tfrac{1}{\tau\,\lambda^{TG}}\Big(2-\tfrac{1}{\tau\,\lambda^{TG}}\Big)\,,$$

where the last equality follows from $1 \,\leq\, \tau \,<\, \frac{1}{\tau\,\lambda^{TG}} \,<\, 2\,$; one sees thus that (2.7) yields $1 > \lambda_{k-1} \geq \frac{1}{\tau}\big(2-\frac{1}{\tau\,\lambda^{TG}}\big)$ as claimed. □

If $\tau \,=\, 1$, then the condition $\lambda^{TG} \,>\, \frac{1}{2}$ corresponds to the one obtained in [19] for the optimality of the W-cycle. Using $\tau > 1$ leads to significantly weaker (i.e., more favorable) conditions. In particular, for any $\lambda^{TG} \,>\, \frac{1}{4}$ one may find a $\tau \,>\, 1$ such that the resulting multigrid cycle is optimal (at least, if $\lambda_{\min}(G_{L-1}A_{L-1}) \geq \lambda^{TG}$); this is the same condition as for the AMLI-cycle [28]. Moreover, it is not necessary to have formally a bound independent of the number of levels since in practice there are not that many levels, especially with the GPU implementation. On the one hand, we use the aggregation algorithm from [14] with three passes of pairwise aggregation, which is relatively aggressive. On the other hand, as discussed in the next section, we stop the recursion when there is still a relatively large number of unknowns at the coarsest level.

Overall, the numerical results show that the best results were obtained setting the parameter $\tau = \tau_{1,2,3}^{(k+1)} = 1.75$, and this value has been used uniformly in all experiments reported below.

In Table 2.2 we compare the so-defined relaxed W-cycle with the standard W-cycle; see Section 3 for the details of the experimental setting and the problem description. One sees that the relaxation brings a significant improvement at virtually no cost, and it helps to maintain a nearly constant number of iteration as the problem size increases. The relaxed W-cycle is compared with the K-cycle in Section 3.

2.3. Bottom-level solver. Standard AGMG uses a sparse direct solver as a bottom-level solver. However, to the best of our knowledge, the current implementation of sparse direct solvers is not efficient enough on GPUs. Moreover, the coarsest level has relatively few unknowns, hence the potential of GPU acceleration is significantly more limited than at the fine-grid level. Therefore, here the bottom-level solver is implemented on the CPU rather than on the GPU, as illustrated in Figure 2.1: the residual restricted at the coarsest level L is transferred from GPU to CPU, then the bottom-level solve is applied on the CPU, and the approximate solution is transferred back to GPU memory.

TABLE 2.2
Number of iterations for the standard and relaxed W-cycle.

		W-cycle	
Problem	size	standard	relaxed
MOD2D	S1	23	15
	S2	28	15
	S3	31	15
JUMP2D	S1	41	25
	S2	58	30
	S3	74	32
MOD3D	S1	15	11
	S2	17	12
	S3	20	12
SPHERE$_1$	S1	12	12
	S2	13	11
	S3	15	10

For this CPU bottom-level solver we could use a sparse direct solver as does standard AGMG, but, inspired by [23], we obtained even better results using the standard (CPU version) AGMG, that is, the bottom-level solver consists in one application of the sequential AGMG preconditioner.

This turns out to be cost effective for three reasons. Firstly, despite one application of sequential AGMG only provides a relatively rough approximation of the exact solution, the numerical experiments show that the number of FCG iterations is practically not affected.

Secondly, even though the number of unknowns at the coarsest level is relatively small, one such application remains significantly cheaper than one solve with the triangular factors of a pre-computed sparse LU factorization, and the influence on the overall cost is not negligible because the use of the K- or of the relaxed W-cycle entails that one needs 2^{L-2} bottom-level solves per call to the preconditioner at the fine-grid level.

Finally, using AGMG implies that the time is proportional to the number of involved unknowns, whereas the time needed by direct solvers grows more than linearly. This allows us to increase up to $n_c = 5000$, which is the threshold for the number of the coarse-grid unknowns below which the bottom level is reached, while with a sparse direct solver a smaller value would be needed. In this way, the total number of levels is often smaller, implying less calls to the bottom-level solver, whereas one avoids to deal on the GPU with intermediate levels having too few unknowns for an efficient GPU acceleration.

3. Numerical results.

Numerical experiments were performed with linear systems arising from finite difference (FD) or finite element (FE) discretizations of the second-order scalar elliptic boundary value problem

$$\begin{cases} -\overline{\nabla}\left(D\,\overline{\nabla}u\right) &= f \qquad \text{in } \Omega \subset \mathbb{R}^d, \\ u &= g_0 \qquad \text{on } \Gamma_D, \\ \dfrac{\partial u}{\partial n} &= g_1 \qquad \text{on } \Gamma_N = \partial\Omega \backslash \Gamma_D, \end{cases}$$

TABLE 3.1
Problem sizes and sparsity of the associated system matrices.

Problem	*S1*		*S2*		*S3*	
	$\frac{n}{10^6}$	$\frac{nnz(A)}{n}$	$\frac{n}{10^6}$	$\frac{nnz(A)}{n}$	$\frac{n}{10^6}$	$\frac{nnz(A)}{n}$
ANI2D, BFE, JUMP2D, MOD2D	0.4	5.0	3.2	5.1	25	5.0
LSHAPE$_1$	0.3	7.0	1.3	7.0	5.1	7.0
LSHAPE$_2$	0.3	11.4	1.3	11.5	5.1	11.5
LSHAPE$_3$	0.7	16.9	2.9	17.0	11.5	17.0
LSHAPE$_4$	1.3	23.4	5.1	23.5	–	–
ANI3D$_{a.b}$, JUMP3D, MOD3D	0.5	6.9	4.02	7.0	33	7.0
SPHERE$_1$	0.008	12.5	0.06	15.0	0.5	15.2
SPHERE$_2$	0.06	28.3	0.5	28.8	–	–
SPHERE$_3$	0.2	48.3	1.7	47.4	–	–
SPHERE$_4$	0.5	74.8	–	–	–	–

in two ($d = 2$) or three ($d = 3$) dimensions. We briefly describe the test problems below, treating separately two- and three-dimensional problems (2D and 3D, respectively); more details can be found in [20] for problems discretized on Cartesian grids, whereas the other problems are similar to those considered in [14, 15]. Note that all discretizations using Cartesian grids correspond to a unit square (2D) or a unit cube (3D).

The 2D problems include

- five-point FD discretization of constant coefficient problems with coefficient $D = (1, \varepsilon_y)$ on a Cartesian grid; we consider $\varepsilon_y = 1$ (MOD2D) and $\varepsilon_y = 10^{-2}$ (ANI2D);
- five-point FD discretization of a piecewise-constant coefficient problem on a Cartesian grid (JUMP2D) ;
- bilinear FE discretization of a constant coefficient problem with anisotropic coefficient $D = (1, 10^{-2})$ on a Cartesian grid (BFE);
- FE discretization using P_k Lagrangian finite elements on an unstructured L-shaped mesh with simplex size progressively decreased near the reentering corner in such a way that the mesh size in its neighborhood is about 10^3 times smaller; the orders $k = 1, \dots, 4$ correspond to LSHAPE$_1$, ..., LSHAPE$_4$.

Regarding the 3D problems, they include

- seven-point FD discretization of constant coefficient problems in 3D with coefficient $D = (1, \varepsilon_y, \varepsilon_z)$; we consider $\varepsilon_y, \varepsilon_z = 1$ (MOD3D), $\varepsilon_y = 1$, $\varepsilon_z = 0.07$ (ANI3D$_a$), and $\varepsilon_x = \varepsilon_y = 0.07$ (ANI3D$_b$);
- seven-point FD discretization of a piecewise-constant coefficient problem on a Cartesian grid (JUMP3D);
- FE discretization using P_k Lagrangian finite elements on the unit cube with $D = 1$ everywhere except within a small sphere of radius $2/5$ at the center of the domain, where $D = 10^3$; the grid is unstructured and quasi-uniform; the orders $k = 1, \dots, 4$ correspond to SPHERE$_1$, ..., SPHERE$_4$.

Each problem comes in (at most) 3 different sizes that are given in Table 3.1.

Two variants of the hybrid GPU-CPU version of AGMG are considered: one using a K-cycle and the other one using a relaxed W-cycle with relaxation parameter $\tau = 1.75$; see Section 2.2 for details. Besides this, we use the method as described in the previous section, thus with a Chebyshev-accelerated ℓ_1-Jacobi smoother (Section 2.1) and a standard AGMG on

TABLE 3.2
Solve time and number of iterations for the sequential CPU version of AGMG (CPU_{seq}), multithreaded CPU version (6 threads) of AGMG (CPU_{mth}), and the presented hybrid GPU-CPU version (GPU), as well as the speedup ($\frac{CPU_{mth}}{GPU}$) of the GPU version over the multithreaded one. All variants use the K-cycle.

		Solve (ms)			Speedup	Number of iterations		
Problem	size	CPU_{seq}	CPU_{mth}	GPU	$\frac{CPU_{mth}}{GPU}$	CPU_{seq}	CPU_{mth}	GPU
JUMP2D	S1	536.	229.	75.	3.05	23	41	24
	S2	6090.	3230.	458.	7.05	26	49	28
	S3	52100.	30300.	2620.	11.6	27	56	28
MOD3D	S1	375.	114.	29.	3.93	10	11	10
	S2	3350.	1080.	143.	7.55	10	11	10
	S3	30600.	9560.	1230.	7.77	11	11	11
LSHAPE_3	S1	1620.	430.	99.	4.34	14	16	13
	S2	6650.	1960.	287.	6.83	14	17	14
	S3	30800.	8680.	1010.	8.59	16	18	14
SPHERE_1	S1	10.	8.	7.	1.14	8	10	12
	S2	116.	31.	22.	1.41	11	10	10
	S3	1060.	293.	76.	3.86	10	10	10

the CPU as bottom-level solver (Section 2.3). The setup is performed with standard AGMG, thus using quality-based multiple pairwise aggregation [14], with in this case $n_{\text{pass}} = 3$ passes (i.e., aggregates of size at most 8). The coarsest level is reached if the number of coarse unknowns is below 5000.

Both variants are used as preconditioners for the FCG method, with the zero vector as initial approximation; the iterations are stopped when the relative residual error is below 10^{-6}. The results were obtained on a single-processor unit (Intel i7-7800X at 4GHz, with 16GB RAM) with a RTX 2080 Ti GPU (11GB RAM). All computation were done in double precision.

First, we report in Table 3.2 the results for the hybrid GPU-CPU version of AGMG based on the K-cycle along with those of the latest (sequential and multithreaded) versions of AGMG [21] running on the CPU; the multithreaded versions is run with 6 threads (optimal for a 6-core processor). For the sake of brevity, only a subset of the results is presented with two problems of each kind (2D and 3D; structured and unstructured); the results for other test problems are similar. Regarding the number of iterations, we note that it is essentially the same for the hybrid GPU-CPU version and the sequential CPU version and even sometimes slightly larger for this latter version. This illustrates the effectiveness of the Chebyshev-accelerated ℓ_1-Jacobi smoother as proposed in Section 2.1, since the major difference between both solvers from the algorithmic viewpoint is precisely in the smoothing scheme. On the other hand, the number of iterations of the multithreaded version is higher for some problems. This is because the parallelisation of this latter is different from that of the hybrid GPU-CPU version. Regarding the reported speedups of the GPU version over the multithreaded one, they mainly indicate for the considered machine a typical performance improvement from transferring the computation to GPU.

We now report on the comparison of the considered method with AmgX, a GPU only solver by NVIDIA [16]. Two configurations of AmgX are considered, referred to as *aggre-*

ETNA
Kent State University and
Johann Radon Institute (RICAM)

TABLE 3.3
Setup time for AGMG (sequential, on CPU), solve time, and number of iterations for the hybrid GPU-CPU version of AGMG (AGMG-GPU) and AmgX applied to the 2D problems; "Agg" refers to the aggregation configuration and "Cls" to the classical one; n.c. means that the solver did not converge in 300 iterations.

		Setup	Solve				Number of iterations			
			AGMG-GPU		AmgX		AGMG-GPU		AmgX	
Problem			K-cycle	W-cycle	Agg	Cls	K-cycle	W-cycle	Agg	Cls
ANI2D	S1	187.	68.	57.	172.	43.	23	24	159	39
	S2	2120.	363.	373.	1127.	200.	25	30	199	41
	S3	20600.	2220.	2650.	8959.	1445.	26	32	216	44
BFE	S1	299.	74.	74.	105.	–	23	28	98	n.c.
	S2	3070.	455.	524.	887.	–	27	33	165	n.c.
	S3	26200.	3200.	3740.	7412.	–	30	36	193	n.c.
JUMP2D	S1	212.	75.	65.	103.	195.	24	25	107	42
	S2	2350.	458.	469.	788.	172.	28	30	176	45
	S3	22300.	2620.	2790.	6818.	1428.	28	32	216	48
MOD2D	S1	213.	47.	44.	85.	43.	13	15	89	39
	S2	2300.	213.	222.	644.	172.	14	15	140	41
	S3	21900.	1280.	1280.	4996.	1297.	14	15	159	45
LSHAPE$_1$	S1	211.	17.	19.	17.	25.	5	6	17	25
	S2	832.	52.	57.	61.	73.	6	7	23	30
	S3	3200.	183.	199.	329.	276.	7	8	39	38
LSHAPE$_2$	S1	322.	47.	51.	59.	–	13	15	55	n.c.
	S2	1290.	131.	138.	216.	–	13	15	76	n.c.
	S3	4990.	401.	389.	796.	–	14	15	81	n.c.
LSHAPE$_3$	S1	998.	99.	107.	139.	–	13	15	57	n.c.
	S2	4020.	287.	289.	564.	–	14	15	80	n.c.
	S3	16100.	1010.	1100.	2214.	–	14	16	82	n.c.
LSHAPE$_4$	S1	2330.	252.	268.	–	–	15	17	n.c.	n.c.
	S2	9480.	819.	824.	–	–	16	17	n.c.	n.c.

gation and *classical*. For the *aggregation* configuration, we select the option that provides aggregates of size 4. For the *classical* configuration we use classical AMG coarsening, where the vertices of the coarser grid are connected if they are directly or indirectly connected in the finer grid ("distance-2" graph or "D2 interpolation"). Further, for the latter configuration, aggressive coarsening is used at the 2 finest levels to avoid systematically running out of memory for the largest problems. Both configurations use AMG as a preconditioner for the FCG method and the block Jacobi smoother with $\omega = \frac{2}{3}$. These configurations are similar to the "standard" configuration provided with AmgX, and these options have been chosen as they give overall better results (lower running times) than alternatives.

The results are given in Table 3.3 for the 2D problems and in Table 3.4 for the 3D problems, and they are also illustrated with bar diagrams in Figures 3.1, 3.2, 3.3, and 3.4 (one figure per problems kind: 2D and 3D, structured and unstructured).

Comparing first the results for the K-cycle and the relaxed W-cycle variants, we note that their performance is quite similar. If the number of iterations is the same or close, as typically

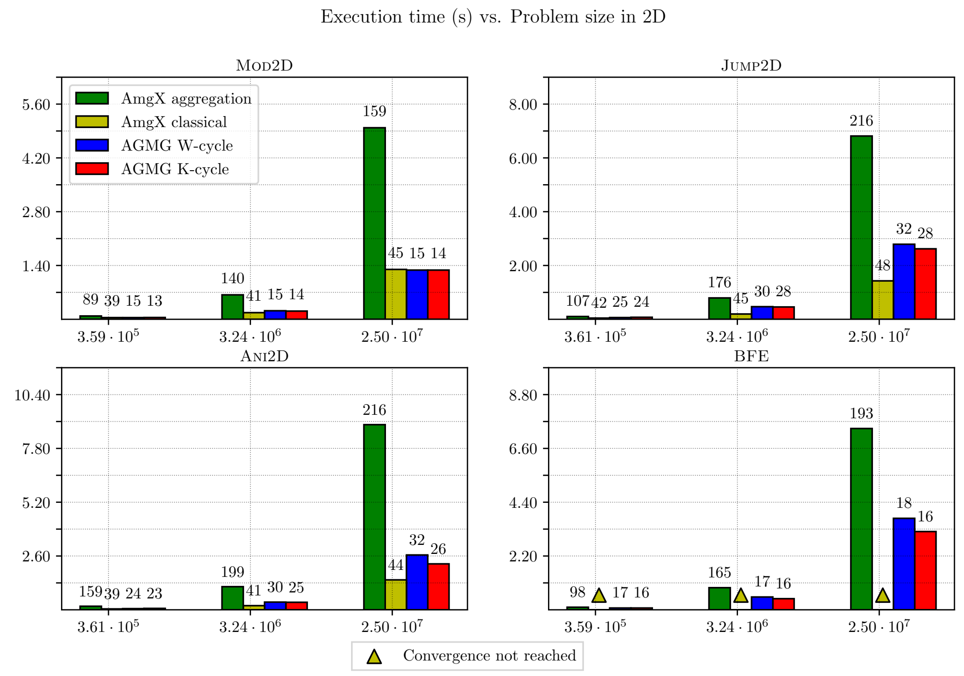


FIG. 3.1. *Results for structured 2D problems in GPU timing. The number on top of each bar is the number of iterations.*

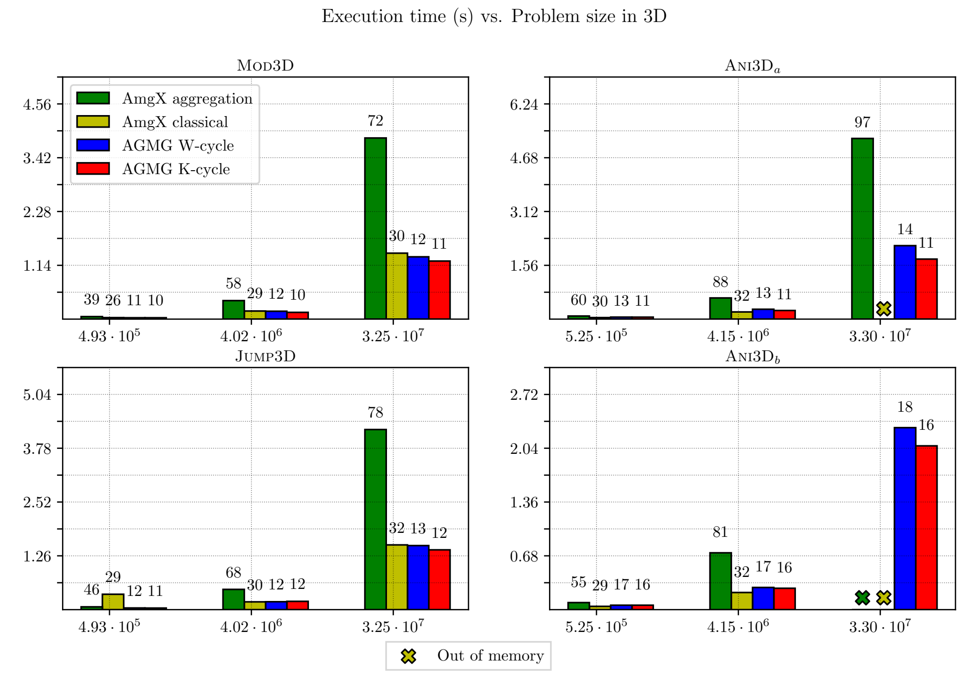


FIG. 3.2. *Results for structured 3D problems in GPU timing. The number on top of each bar is the number of iterations.*

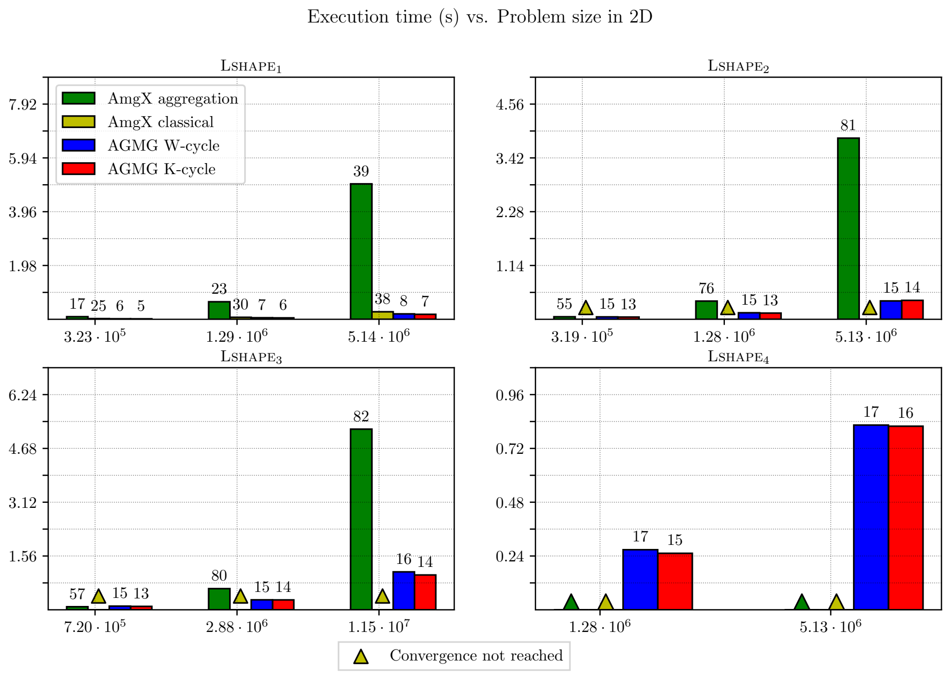


FIG. 3.3. *Results for unstructured 2D problems in GPU timing. The number on top of each bar is the number of iterations.*

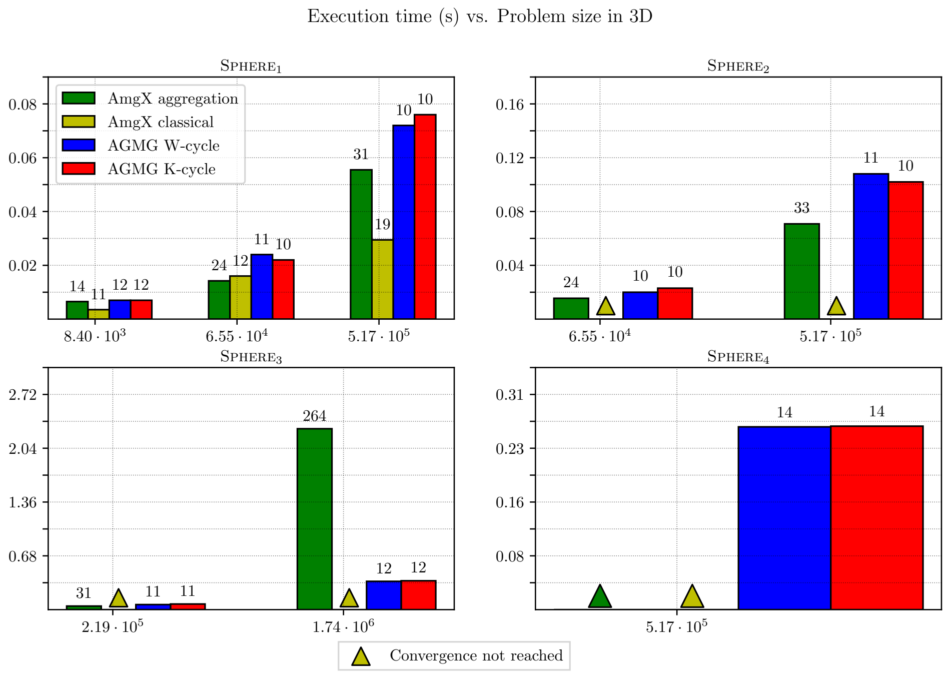


FIG. 3.4. *Results for unstructured 3D problems in GPU timing. The number on top of each bar is the number of iterations.*

TABLE 3.4
Setup time for AGMG-GPU (sequential, on CPU), solve time, and number of iterations for the hybrid GPU-CPU version of AGMG (AGMG-GPU) and AmgX applied to the 3D problems; "Agg" refers to the aggregation configuration and "Cls" to the classical one; n.c. means that the solver did not converge in 300 iterations; o.o.m. means that the solver run out of memory. Both setup and solve time are represented in milliseconds.

		Setup	Solve				Number of iterations			
			AGMG-GPU		AmgX		AGMG-GPU		AmgX	
Problem			K-cycle	W-cycle	Agg	Cls	K-cycle	W-cycle	Agg	Cls
ANI3D$_a$	S1	418.	57.	58.	88.	44.	11	13	60	30
	S2	4460.	250.	286.	613.	208.	11	13	88	32
	S3	49900.	1740.	2130.	5239.	–	11	14	97	o.o.m.
ANI3D$_b$	S1	435.	56.	56.	89.	41.	16	17	55	29
	S2	4390.	270.	280.	718.	216.	16	17	81	32
	S3	47400.	2070.	2300.	–	–	16	18	o.o.m.	o.o.m.
JUMP3D	S1	331.	37.	39.	65.	36.	11	12	46	29
	S2	3420.	194.	185.	474.	182.	11	12	68	30
	S3	39800.	1400.	1500.	4218.	1516.	12	13	78	32
MOD3D	S1	332.	29.	29.	53.	31.	10	11	39	26
	S2	3360.	143.	167.	392.	171.	10	11	58	29
	S3	39000.	1230.	1320.	3838.	1397.	11	12	72	30
SPHERE$_1$	S1	11.	7.	7.	6.	3.	12	12	14	11
	S2	107.	22.	24.	14.	16.	10	11	24	12
	S3	932.	76.	72.	55.	29.	10	10	31	19
SPHERE$_2$	S1	163.	23.	20.	15.	–	10	10	24	n.c.
	S2	1500.	102.	108.	70.	–	10	11	33	n.c.
SPHERE$_3$	S1	815.	71.	64.	45.	–	11	11	31	n.c.
	S2	6830.	365.	358.	2286.	–	12	12	264	n.c.
SPHERE$_4$	S1	2670.	266.	265.	–	–	14	14	n.c.	n.c.

is the case for unstructured problems, the relaxed W-cycle variant is slightly faster as it does not perform any dot product computation. However, for most problems, the K-cycle variant requires slightly less iterations, which is often enough to compensate for a slightly slower multigrid cycle. Further, both variants are robust in that the number of iterations remains under 40 for the relaxed W-cycle variant, while it does not exceed 30 for the K-cycle one. The number of iterations is also hardly affected by the problem size.

This robustness is even more striking when comparing with the considered AmgX configurations. Regarding the aggregation configuration, we note that it is typically slower, which is due to a large number of iterations required in order to reach convergence (whereas the time per iteration is actually typically smaller than that of the hybrid GPU-CPU version of AGMG). This number of iterations often increases dramatically with the problem size. Regarding the classical configuration of AmgX, its performance is comparable to the hybrid GPU-CPU version of AGMG, and it is even significantly faster for some problems. However, despite two levels of aggressive coarsening, it runs out of memory for some problems, while exhibiting slow convergence or lack of it for some others, as it is the case, for instance, for 2D and 3D FE discretizations of order 2 and higher.

4. Conclusions.

We have presented an aggregation-based AMG method for hybrid GPU-CPU architectures. The distinctive features of the method are the Chebyshev-accelerated

ℓ_1-Jacobi smoother, an optional use of the relaxed W-cycle as an alternative to the standard K-cycle that avoids dot products, and an AMG-based bottom-level solver that runs on the CPU. Other features include the quality-based aggregation coarsening, which is recursively used during the setup stage on the CPU to build the multigrid hierarchy.

Numerical experiments with an AGMG-based implementation have revealed that the resulting solver inherits the robustness of the standard sequential AGMG, while its solution stage runs up to 12 times faster than its multithreaded counterpart. A comparison with AmgX, a GPU-based AMG code from NVIDIA, further indicates that the presented implementation is significantly more robust. This superior robustness leads to a faster solution stage for most of the problems for which one of the AmgX configurations does converge.

Incidentally, our results show that Chebyshev-accelerated ℓ_1-Jacobi smoothing can outperform both standard polynomial smoothing and standard ℓ_1-Jacobi smoothing and also that the relaxed W-cycles can be significantly faster than the standard W-cycle.

Acknowledgments. We thank Siham Boukhris for extensive and fruitful discussions. We also thank Frank Hülsemann and two anonymous referees for their useful comments.

REFERENCES

[1] M. Adams, M. Brezina, J. Hu, and R. Tuminaro, *Parallel multigrid smoothing: polynomial versus Gauss-Seidel*, J. Comput. Phys., 188 (2003), pp. 593–610.
[2] H. Anzt, S. Tomov, M. Gates, J. Dongarra, and V. Heuveline, *Block-asynchronous multigrid smoothers for GPU-accelerated systems*, Procedia Comput. Sci., 9 (2012), pp. 7–16.
[3] O. Axelsson, *Iterative Solution Methods*, Cambridge University Press, Cambridge, 1994.
[4] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, *Multigrid smoothers for ultraparallel computing*, SIAM J. Sci. Comput., 33 (2011), pp. 2864–2887.
[5] W. L. Briggs, V. E. Henson, and S. McCormick, *A Multigrid Tutorial*, 2nd ed., SIAM, Philadelphia, 2000.
[6] E. Dick and K. Riemslagh, *Multi-staging of Jacobi relaxation to improved smoothing properties of multigrid methods for steady Euler equations*, J. Comput. Appl. Math., 50 (1994), pp. 241–254.
[7] S. C. Eisenstat, H. C. Elman, and M. H. Schultz, *Variational iterative methods for nonsymmetric systems of linear equations*, SIAM J. Numer. Anal., 20 (1983), pp. 345–357.
[8] E. Alerstam, T. Svensson, and S. Andersson-Engels, *Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration*, J. Biomed. Optics, 13 (2008), Art. 060504, 3 pages.
[9] R. D. Falgout, R. Li, B. Sjögreen, L. Wang, and U. M. Yang, *Porting hypre to heterogeneous computer architectures: strategies and experiences*, Parallel Comput., 108 (2021), Art. 102840, 12 pages.
[10] P. Ghysels, P. Kłosiewicz, and W. Vanroose, *Improving the arithmetic intensity of multigrid with the help of polynomial smoothers*, Numer. Linear Algebra Appl., 19 (2012), pp. 253–267.
[11] V. E. Henson and U. M. Yang, *BoomerAMG: A parallel algebraic multigrid solver and preconditioner*, Appl. Num. Math., 41 (2002), pp. 155–177.
[12] D. C. Jespersen, *Acceleration of a CFD code with a GPU*, Sci. Programm., 18 (2010), pp. 193–201.
[13] A. Napov and Y. Notay, *Algebraic analysis of aggregation-based multigrid*, Numer. Linear Algebra Appl., 18 (2011), pp. 539–564.
[14] ———, *An algebraic multigrid method with guaranteed convergence rate*, SIAM J. Sci. Comput., 34 (2012), pp. A1079–A1109.
[15] ———, *Algebraic multigrid for moderate order finite elements*, SIAM J. Sci. Comput., 36 (2014), pp. A1678–A1707.
[16] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, *AmgX: a library for GPU accelerated algebraic multigrid and preconditioned iterative methods*, SIAM J. Sci. Comput., 37 (2015), pp. S602–S626.
[17] Y. Notay, *AGMG software and documentation*. http://agmg.eu
[18] Y. Notay, *Flexible conjugate gradients*, SIAM J. Sci. Comput., 22 (2000), pp. 1444–1460.
[19] ———, *Convergence analysis of perturbed two-grid and multigrid methods*, SIAM J. Numer. Anal., 45 (2007), pp. 1035–1044.

[20] ———, *An aggregation-based algebraic multigrid method*, Electron. Trans. Numer. Anal., 37 (2010), pp. 123–146.
http://etna.ricam.oeaw.ac.at/vol.37.2010/pp123-146.dir/pp123-146.pdf

[21] ———, *User's guide for AGMG*, 2010. http://agmg.eu

[22] ———, *Algebraic theory of two-grid methods*, Numer. Math. Theory Methods Appl., 8 (2015), pp. 168–198.

[23] Y. NOTAY AND A. NAPOV, *A massively parallel solver for discrete Poisson-like problems*, J. Comput. Phys., 281 (2015), pp. 237–250.

[24] Y. NOTAY AND P. S. VASSILEVSKI, *Recursive Krylov-based multigrid cycles*, Numer. Linear Algebra Appl., 15 (2008), pp. 473–487.

[25] J. D. OWENS, M. HOUSTON, D. LUEBKE, S. GREEN, J. E. STONE, AND J. C. PHILLIPS, *GPU computing*, Proc. IEEE, 96 (2008), pp. 879–899.

[26] K. STÜBEN, *An introduction to algebraic multigrid*, in Multigrid, U. Trottenberg, C. W. Oosterlee, and A. Scüller, eds., Academic Press, London, 2001, pp. 413–532.

[27] K. STÜBEN AND U. TROTTENBERG, *Multigrid methods: fundamental algorithms, model problem analysis and applications*, in Multigrid Methods (Cologne, 1981), W. Hackbusch and U. Trottenberg, eds., vol. 960 of Lecture Notes in Math., Springer, Berlin, 1982, pp. 1–176.

[28] P. S. VASSILEVSKI, *Multilevel Block Factorization Preconditioners*, Springer, New York, 2008.

[29] H. WANG AND Y. YANG, *Descent methods for elastic body simulation on the GPU*, ACM Trans. Graph., 35 (2016), Art. 212, 10 pages.

[30] U. M. YANG, *Parallel algebraic multigrid methods—high performance preconditioners*, in Numerical Solution of Partial Differential Equations on Parallel Computers, A. M. Bruaset and A. Tveito, eds., vol. 51 of Lecture Notes in Comput. Sci Eng., Springer, Berlin, 2006, pp. 209–236.