Implementation of the Algebraic Multigrid Solver Designed for Graphics Processing Units Based on the AMGCL Framework

O. I. Ryabkov(✉)

Federal Research Center "Computer Science and Control"
of the Russian Academy of Sciences, Moscow, Russia
roi-techsup@yandex.ru

Abstract. The implementation of the Algebraic Multigrid (AMG) solver designed specifically for Graphics Processing Units (GPUs) is presented. It is based on the well-known and highly efficient AMGCL header-only library designed and implemented by D. Demidov using C++. The original AMGCL approach for GPU speedup relies on the initialization (setup) phase performed purely on the CPU, while the solution (iteration process) is moved to the GPU. This approach works well for the case of transient solvers, when the system matrix does not change much during time-stepping. However, it does not fit for cases of highly nonlinear systems or stationary systems, especially when a linear system is formed in the device memory. For these systems it is better to use GPU-only solvers. To implement the GPU-oriented AMG solver, the design of the original framework had to be changed. The maximal independent set aggregation algorithm and derived smoothed aggregation operations are added to the framework. A number of smoothers on the intermediate levels are implemented with full support for GPUs. The full AMG hierarchy can now be constructed entirely on the GPU with no CPU invokes. The method is tested against the original AMGCL framework on matrices derived from elliptic and parabolic partial differential equations (PDEs). It is shown that the GPU-only approach can speed up the setup phase by up to 5-6 times compared to the original framework.

Keywords: Aggregation · Iterative methods · Algebraic Multigrid methods · Elliptic partial differential equations · General Purpose GPU computations

1 Introduction

There are few modern methods for solving large linear systems that can be considered mainstream. These include multigrid methods and domain decomposition methods. Among the former, Algebraic Multigrid (AMG) methods are one

The study was funded by the Russian Foundation for Basic Research, project No. 20-07-00066.

L. Sokolinsky and M. Zymbler (Eds.): PCT 2022, CCIS 1618, pp. 131–142, 2022.
https://doi.org/10.1007/978-3-031-11623-0_10

of the most popular choices. The methods (generally) require no additional information, however, the matrix itself is relatively memory thrifty and, if correctly constructed, can be as efficient as geometric multigrid (GMG) methods. All this makes AMG a very attractive "black box" solver for particular classes of problems (positive definite, semi-positive definite and M-matrices). Any multigrid solver has two stages: setup and solve. The setup stage performs the necessary computations and prepares operators for the solve phase. The solve phase actually solves a linear system. In AMG methods, the solve phase is a substantial and important part in both the convergence and wall time. We refer the reader to [6,8,11,16] for more detailed information on AMG methods. In this work, we mainly focus on the setup phase.

To construct matrices, smoothing, prolongation and restriction multigrid operators on all levels, one uses entries in the main system matrix, as well as, possibly, some external information. The resulting set of operators is called the AMG hierarchy. The classical AMG approach [11] constructs hierarchies by dividing matrix entries into coarse and fine ones so that the smoothed error slowly varies in the direction of large matrix coefficients. The coarse nodes are used to construct the lower level, and the prolongation operation is defined by interpolation. The restriction operation is usually defined by either the one-to-one restriction operation or the transpose of the prolongation operator. Smoothing operators are built based on the nodes on each levels. This approach often results in relatively good hierarchies that guarantee grid independent convergence. However, to achieve this quality, classical aggregation often leads to large coarse level matrices and requires a substantial amount of memory, see [10] for more information. In addition, the original classical aggregation algorithm is essentially serial. Other variants of the algorithm, such as PMIS, HMIS, CLJS, etc. (see [15]), do not produce hierarchies of such quality.

Another variant considered here is aggregation AMG methods [14]. These methods rely on grouping fine level nodes to form a coarse matrix on the lower level. Smoothing operations are also constructed on top of the formed matrices on each level. The restriction operation usually acts as an averaging of the variables inside each group, and the prolongation operator is a transposed restriction operator. Such an approach is much more economical in terms of memory consumption on the coarse levels, however, this approach is usually not applied since it cannot generally provide grid independent convergence rates [13]. To override this problem, smoothed aggregation was proposed [13,14]. However, smoothed aggregation AMG results in greater memory requirements, and grid independent convergence is still not guarantied for some classes of problems. For a comparison of different approaches see [12,15].

Having described one problem of selecting an appropriate setup procedure, one faces another problem, namely, how to speed up the setup phase. To speed up an aggregation-based algorithm, it is necessary to apply a parallel aggregation algorithm with all variants, including smoothed aggregation (since none of them can be universal). Modern high-performance computing architectures de facto must have Graphics Processing Units (GPUs). Such an architecture is efficient,

if properly programmed, and more environmentally friendly. In addition, modern desktops can fit up to 4 (or 8) powerful GPUs capable of solving relatively large-scale problems. It would be unwise to deprive the users of these desktops from solving middle-sized problems of academic or engineering orientation. Hence, the implementation of the fully GPU-accelerated AMG solver is an important task.

We tried to utilize the AMGX solver for our problems on GPUs, but failed, see [5]. To verify our implementations of GPU-accelerated setup procedures, we used the AMGCL library [3] by D. Demidov. It is a C++ header-only library that heavily relies on template metaprogramming and has GPU support via CUDA and OpenCL. It is efficient and has been tested in many applications. However, the library is explicitly designed in such a way that the setup process is performed on CPUs only, accelerated by either OpenMP or MPI. GPU support is localized only in the solve stage. Besides, if the system matrix is formed on GPUs, then the library performs CPU↔GPU memcpy. The author's idea is that the setup is executed only a limited number of times, and the matrix of the linear system can be reused (by calling the rebuild process also performed on the CPU), if applied many times, say in Newton's method, see [4] for details. However, if the problem being solved is complicated, and the stationary point is not easily found (see, for example, [2]), then this strategy may lead to unsatisfactory results, e.g. substantially decrease the CFL number in implicit methods. In this paper, we would like to overcome this flaw. We apply the Parallel Maximal Independent Set K (MIS(K)) on the GPU, as described in [1,7], with modifications that form aggregates closer to the serial version. The method is implemented in CUDA C++ using templates.

The paper is laid out as follows. First, the aggregation method, the modified MIS(K) method and its application in AMG during the setup are described. Next, the modifications introduced in the AMGCL library to implement this method in the GPU-only approach are outlined. Numerical experiments on several available and generated sparse matrices are also presented, and the performance and convergence of the modified and original AMGCL library are measured. The paper is finalized by a conclusion.

2 Aggregation AMG on the GPU

The initial approach adopted for the AMG hierarchy build process in AMGCL is based on constructing aggregates as noted in the introduction. Aggregates are unions of nodes (variables) on the fine level. After aggregation, each aggregate corresponds to one and only one node on the coarse level. Let n be the number of nodes on the current (fine) level. We can mathematically describe the aggregate structure by the array of numbers a_i, where $i \in \{0, ..., n-1\}$ and $a_i \in \{0, ..., n_c - 1\}$, and n_c is, in turn, the number of nodes on the next (coarse) level.

Transfer operators (prolongation and restriction) are fully determined by the aggregate structure. Regular (non-smoothed) aggregation builds the restriction operator (matrix) R in the following way:

$$R_{j,i} = \begin{cases} \frac{1}{|\{k|a_k=j\}|}, & \text{if } a_i = j \\ 0, & \text{otherwise} \end{cases}$$

The prolongation matrix is defined as a transposition of the restriction matrix: $P = R^T$. The coarse operator matrix is defined according to the Galerkin projection: $A_c = RAP$, where A is the fine level matrix. For smoothed aggregation, the restriction matrix is defined as a product of the restriction matrix defined above and the smoothing matrix $I + \omega A^F$. ω is the relaxation parameter, and A^F is a specially filtered version of the matrix A. Further levels are constructed in a recursive way.

One can see that in this formalism the overall aggregation algorithm is fully determined by the method of constructing aggregates. Regardless of the choice of a particular algorithm, a strong connections graph first needs to be constructed. There are several variations of strong connections criteria. The one used in AMGCL is described, for example, in [14]. We denote the strong connections graph incidence matrix by C, $C_{i,j} = 1$ means that the node with the number i is strongly connected to the node with the number j, while $C_{i,j} = 0$ means the absence of connection. Note that the matrix C is supposed to be symmetric in the algorithm mentioned below.

The initial algorithm in AMGCL (called plain aggregation) uses a substantially serial approach that exploits a given order of nodes for their grouping. On the other hand, our task was to implement the fully GPU workflow for the setup phase, thus the parallelizable algorithm had to be utilized. A common choice for constructing parallel aggregates is the Maximal Independent Set algorithm, see, for example, [1]. A parallel version of this algorithm uses random seeds to construct MIS(K). MIS(K) is the subset of fine level nodes, and the shortest path length between any two MIS(K) nodes in the graph C is larger than K. "Maximal" means that adding any other node to MIS(K) breaks this property. Usually $K = 2$ is used in the context of AMG.

Our version of the MIS(K) algorithm for constructing aggregates is presented here as Algorithm 1. Note that there are two parts that differ from the original version of MIS(K), highlighted in colour in the Algorithm. The first one is in the node weights (the second element of the tuples T_i). While originally only random numbers v_i were used for the weights, we added an extra term $n_i W_{nb}$. W_{nb} is the global algorithm parameter. $W_{nb} = 0$ falls back to the initial version, while $W_{nb} = 1$ or $W_{nb} = -1$ can be used to adjust the behavior of constructing aggregates. We noted that $W_{nb} = 0$ usually resulted in the lower aggregates number compared to the original AMGCL plain aggregation. This leads to a lower convergence rate, thus usually slowing down the solve phase. The $W_{nb} = -1$ choice enlarges the aggregates number, thereby partially fixing the convergence problem. However, for some matrices (not considered in the current paper), $W_{nb} = 1$ may be the best option, since the reduced number of variables on the coarse levels speeds up the computational wall time.

Algorithm 1. MIS(K) parallel, with modification outlined by colour.

```
1: function MISK_AGGREGATION(C, K, W_nb)
2:     I = {0, ..., n − 1};
3:     a ← −1; s ← 0; v ← random;                    ▷ init states and random vector
4:     while {i ∈ I : s_i = 0} ≠ ∅ do
5:         for i ∈ I do                               ▷ for each node in parallel
6:             n_i ← #{j : C_{i,j} ≠ 0, s_j == 0};    ▷ number of neighbors
7:             T_i ← (s_i, v_i+n_i W_nb, i);          ▷ set tuple (state,value,index)
8:         for r = 1, ..., K do                       ▷ propagate distance K
9:             for i ∈ I do                           ▷ for each node in parallel
10:                t ← T_i;
11:                for j : C_{i,j} ≠ 0 do
12:                    t ← max(t, T_j);               ▷ maximal tuple among neighbors
13:                T̂ ← t;
14:            T = T̂;
15:        for i ∈ I do                               ▷ for each node in parallel
16:            (s_max, v_max, i_max) ← T_i;
17:            if s_i == 0 then                       ▷ if unmarked...
18:                if i_max == i then                 ▷ if current is maximal...
19:                    s_i ← 1;                       ▷ mark as new aggregate
20:                    a_i ← i;
21:                else if s_max == 1 then
22:                    s_i ← −1;                      ▷ add to existing aggregate
23:                    a_i ← i_max;
24:    for i ∈ I do                                   ▷ for each node in parallel
25:        if s_i == −1 then                          ▷ not center of aggregate
26:            for j : C_{i,j} ≠ 0 do
27:                if s_j == 1 then                   ▷ neighbor is aggregate center...
28:                    a_i ← j;                       ▷ reconnect to neighbor
29:    return (a);                                    ▷ return list of MIS(K) aggregates
```

The second difference is in the post-processing part. One can consider it as a reconnection procedure. While the original plain aggregates implementation manually connects all closest strong neighbors to newly created aggregate centers, the MIS(K) algorithm can produce highly skewed aggregates. To overcome this problem, additional regrouping was added after the main iterations cycle. The point is to reconnect the nodes initially connected to the "far" aggregates center to the close ones, if any. Together, these two improvements reduce the convergence rate drop to an acceptable level of approximately 10% in the worst case among the systems under consideration. Moreover, the initial plain aggregates algorithm can produce different results depending on matrix ordering, while the randomized algorithm demonstrates robustness to this factor.

All other parts of the AMG hierarchy construction (transfer operators, Galerkin projections) are naturally parallelizable for both CPUs and GPUs.

The most time-consuming part is the sparse matrix-matrix product, and it will be addressed further in more detail.

3 Implementation

Algorithm 1 can be readily implemented on GPUs. Loops with the comment "for each node in parallel" turn into CUDA kernel calls, since there are no data dependencies inside them. There are some technical issues. First, it was not initially clear which data layout was the best for the tuples T. Experiments showed that the "structure of arrays" was still preferable, despite the fact that the size of one tuple is 16 bytes in our implementation. Second, tuples initialization and maximum tuple iterations loops were merged together using the CUB reduce_by_key algorithm, which resulted in better performance.

From the performance point of view, sparse matrix-matrix multiplication was found to be the bottleneck. We first used the legacy cusparseXcsrgemm2 operation from the cuSparse library of the CUDA toolkit, however, that already deprecated version performed badly. The new version introduced in the latest CUDA toolkit exposed a huge speedup of this operation, but failed in terms of extra memory consumption. Finally, we tried the SpECK library [9], which showed the best results in both memory consumption and performance. The only disadvantage of this library is the absence of support for Compute Capabilities lower than 6.1. The Legacy cuSparse implementation was left for the case of older hardware.

Another important aspect of the fully GPU AMG stack is a smoothers ("relaxation" in terms of AMGCL) implementation without any CPU invokes. We ported setups for three of them: spai0 (sparse approximate inversion), ilu0 (incomplete LU factorization) and damped Jacobi. Although in an algorithmic sense there are no problems with their implementation on the GPU, some problems arise with the AMGCL architecture. Initially it was not designed for such GPU-only usage, therefore, we needed to introduce a new setup constructor conveyor from the top make_solver class down to coarsening and smoothers initialization methods.

4 Numerical Experiments

All experiments are conducted on symmetric matrices. The following hardware is used in all experiments: CPU – 2×Intel Xeon Gold 6248R, totally having 48 cores (96 threads) with 512 GB of ECC host memory, GPU – Nvidia Tesla V100 with 32GB of ECC device memory. Double precision is used in all calculations. The AMGCL setup for all experiments is the same: the conjugate gradient method is used as the main solver, preconditioned by a single AMG V-cycle. The ilu0 smoother is used on each level of the multigrid, the exact solver is used in the lowest level. All problems are convergent, the target relative residual is set to $1.0 \cdot 10^{-14}$. All results are presented by three figures – wall time for the setup, solve phases and speedup. In addition, for each matrix, a table that contains

the minimum wall time for all runs and for all implementations is presented. It should be noted once again that the original AMGCL implementation uses a multi-threaded CPU setup phase in both CPU and GPU implementations. Thus, the obtained speedup for the setup phase depends on the number of OpenMP threads for both implementations.

The first experiment is conducted with the matrix available in the AMGCL examples folder, i.e. a small matrix from the discretization of a Poisson equation called POISSON3DB. Its size is $8.56E4$, and the number of nonzero elements is $2.37E6$. The results are presented in Fig. 1 and Table 1.

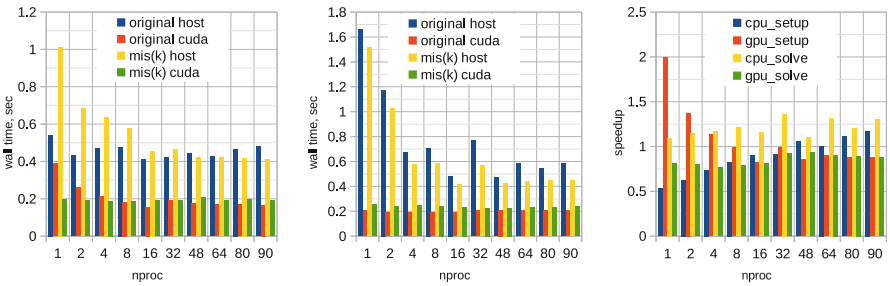

Fig. 1. Results for the POISSON3DB matrix depending on the number of OpenMP threads: setup phase - left, solve phase - center, speedup - right.

It is observed that for such matrix sizes, the speedup is negligible or even reversed. The setup of the AMG hierarchy using the MIS(K) algorithm on the CPU is slower than the original algorithm. The GPU setup phase algorithm is faster than the single-threaded execution of the original algorithm. However, it is slower for 48, 64 and 80 threads. It is not recommended to use GPUs for small matrices.

The next experiment is taken from the sparse matrix market, the matrix is called PARABOLIC_FEM. Its size is $5.26E5$, and it has $3.67E6$ nonzero elements. The results are presented in Fig. 2 and Table 2.

Table 1. Minimum wall times, mean iterations and attained residuals for the POISSON3DB matrix.

name	time_setup	time_solve	total_time	iterations	residual
original host	0.409184	0.471179	0.892288	21	3.62E-16
original cuda	0.155554	0.189926	0.346058	21	5.55E-15
mis(k) host	0.409326	0.417123	0.847573	22	3.12E-15
mis(k) cuda	0.18529	0.225905	0.416847	23	9.47E-16

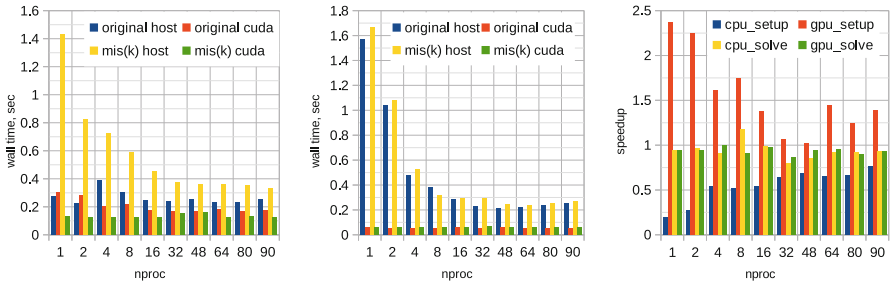

Fig. 2. Results for the PARABOLIC_FEM matrix: setup phase - left, solve phase - center, speedup - right.

Table 2. Minimum wall times, mean iterations and attained residuals for the PARABOLIC_FEM matrix.

name	time_setup	time_solve	total_time	iterations	residual
original host	0.221891	0.212528	0.453675	14	3.22E-15
original cuda	0.164093	0.055595	0.220113	14	3.22E-15
mis(k) host	0.334386	0.240353	0.597847	12	5.67E-15
mis(k) cuda	0.123283	0.05631	0.182331	11	1.17E-15

The results again indicate that the host variant of MIK(K) is slower than the original variant in both the setup and solve phases. The GPU MIS(K) setup phase is 2.4 times faster than the single-threaded original AMGCL implementation and about as fast as the 48-threaded version. The solve phase is slightly slower for the GPU implementation (0.96 times in average). The convergence is slower on 1–2 iterations. The results indicate that the CPU implementation can be used instead of the GPU implementation with a slight penalty on the wall time. These two matrices are small to be efficiently used on GPUs.

The next sparse market matrix is called THERMAL2, its size is $1.23E6$ with $8.58E6$ nonzero elements. The obtained speedup, presented in Fig. 3 and Table 3, shows that the CPU MIS(K) version is slightly faster in the solve phase for 48 threads or more. However, it is almost twice slower for the setup phase.

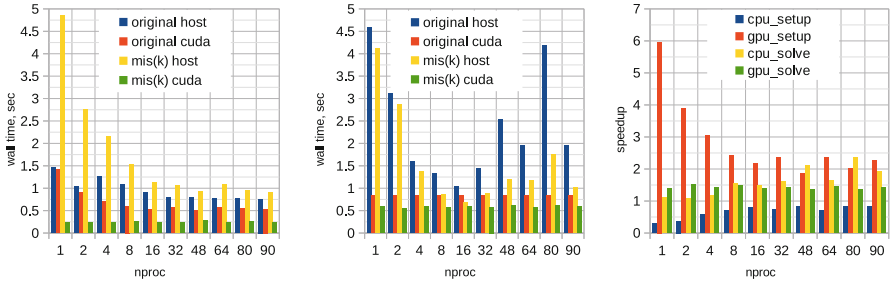

Fig. 3. Results for the THERMAL2 matrix: setup phase - left, solve phase - center, speedup - right.

Table 3. Minimum wall times, mean iterations and attained residuals for the THERMAL2 matrix.

name	time_setup	time_solve	total_time	iterations	residual
original host	0.759762	1.030344	1.925627	13	6.70E-15
original cuda	0.511443	0.830287	1.342058	13	6.70E-15
mis(k) host	0.914104	0.691784	1.824731	11	4.82E-15
mis(k) cuda	0.233334	0.547645	0.780979	13	1.18E-15

The GPU MIS(K) variant is 6 times faster than the single-threaded setup phase of the original implementation. The minimum speedup of the GPU MIS(K) implementation compared to the best multi-threaded original GPU implementation is 1.85 times for the setup phase. The solve phase for the GPU MIS(K) implementation is about 1.5 times faster due to the difference in the obtained AMG hierarchy.

A set of parameterized matrices was generated to perform analysis in terms of matrix sizes. A finite difference 7-point 3D Laplace operator was generated and used for the Poisson equation with Neumann and a single Dirichlet boundary condition. The cubic domain was discretized with 50, 100, 150, 200 and 250 grid points in each direction, respectively. The results of the solution of this problem are presented in Fig. 4 and Table 4.

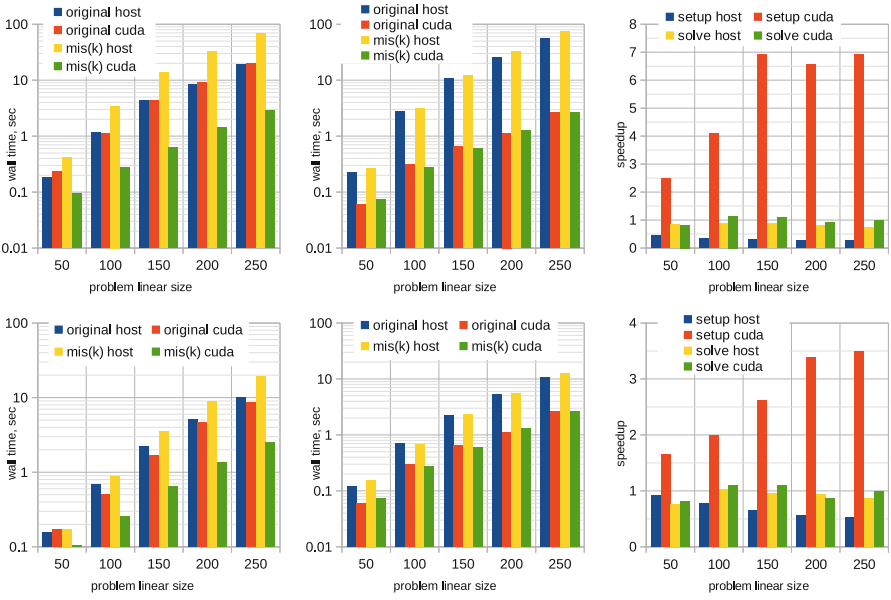

Fig. 4. Results for the generated finite difference Laplace operator: setup phase - left, solve phase - center, speedup - right, upper row - one thread, lower row - best times of all OpenMP threads.

First, we analyze the behavior for the case when a single thread is used in the original AMGCL implementation. The CPU variant of our implementation is clearly inferior compared to the original AMGCL implementation. The GPU variant, on the other hand, is efficient. For this case, we obtain a substantial speedup starting from a linear matrix size of 150. The maximum speedup in the setup phase of about 7 times is achieved for the largest matrix.

Next, we analyze the behavior for the best variant of the AMGCL multi-threaded GPU implementation. In this case, a speedup of about 3.51 times is achieved for the largest matrix in the setup phase. The solve phase is approximately the same for both implementations. The solve phase fluctuates around one, see Table 4.

Table 4. Minimum wall times for all generated Poisson problem matrices and all considered OpenMP threads.

setup				
lin.size	original host	original cuda	mis(k) host	mis(k) cuda
50	0.160	0.167	0.146	0.094
100	0.694	0.508	0.854	0.250
150	2.239	1.554	3.401	0.607
200	4.943	4.449	8.844	1.375
250	9.754	8.253	16.345	2.346
solve				
50	0.085	0.060	0.082	0.069
100	0.580	0.301	0.545	0.259
150	2.271	0.656	2.039	0.594
200	4.317	1.131	5.189	1.262
250	10.145	2.597	10.073	2.621

5 Conclusion

In this research, we presented the implementation of the AMG framework that targets GPUs. The AMGCL header-only library, designed and implemented by D. Demidov using C++, was used as a base framework, which was subject to deep modifications. The whole process (both setup and solve phases) was implemented and tested on multiple symmetric matrices, generated and real-world-alike. It is concluded that for small matrices, e.g. POISSON3DB, the usage of our implementation is not recommended. The GPU load is insufficient to deliver any speedup against modern CPUs. We also do not recommend using the CPU variant of MIS(K) aggregates since it is clearly inferior in all numerical experiments. On the other hand, we obtained a speedup of the setup phase for intermediate and large matrices (matrix size starting from $\sim 1E6$) by about 7 times against the AMGCL single-threaded GPU implementation and by about 3.5 times for the best

multi-threaded variant. The speedup of the solve phase depends on the problem (since aggregates are generated differently for AMGCL and our implementation using MIS(K)) and fluctuates between 0.95 and 1.15. The suggested GPU-only implementation can be recommended if matrices are generated on GPUs for stationary problems with a time-consuming setup phase, as well as for hard transient problems, when a matrix rebuild is required on each time step.

Variations of classical AMG aggregation algorithms, as well as support for multiple GPUs, are to be implemented for GPUs.

References

1. Bell, N., Dalton, S., Olson, L.N.: Exposing fine-grained parallelism in algebraic multigrid methods. SIAM J. Sci. Comput. **34**(4), C123–C152 (2012). https://doi.org/10.1137/110838844
2. Bocharov, A., Evstigneev, N., Petrovskiy, V., Ryabkov, O., Teplyakov, I.: Implicit method for the solution of supersonic and hypersonic 3D flow problems with lower-upper symmetric-gauss-seidel preconditioner on multiple graphics processing units. J. Comput. Phys. **406**, 109189 (2020). https://doi.org/10.1016/j.jcp.2019.109189
3. Demidov, D.: AMGCL - A C++ library for efficient solution of large sparse linear systems. Softw. Impacts **6**, 100037 (2020). https://doi.org/10.1016/j.simpa.2020.100037
4. Demidov, D.E.: Partial reuse AMG setup cost amortization strategy for the solution of non-steady state problems. Lobachevskii J. Math. **42**(11), 2530–2536 (2021). https://doi.org/10.1134/s1995080221110093
5. Evstigneev, N.M., Ryabkov, O.I.: Application of the AmgX library to the discontinuous Galerkin methods for elliptic problems. In: Sokolinsky, L., Zymbler, M. (eds.) PCT 2021. CCIS, vol. 1437, pp. 178–193. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-81691-9_13
6. Falgout, R.: An introduction to algebraic multigrid. Comput. Sci. Eng. **8**(6), 24–33 (2006). https://doi.org/10.1109/mcse.2006.105
7. Gandham, R., Esler, K., Zhang, Y.: A GPU accelerated aggregation algebraic multigrid method. Comput. Math. Appl. **68**(10), 1151–1160 (2014). https://doi.org/10.1016/j.camwa.2014.08.022
8. McCormick, S.F.: Multigrid Methods. Society for Industrial and Applied Mathematics, Philadelphia (1987)
9. Parger, M., Winter, M., Mlakar, D., Steinberger, M.: Speck: accelerating GPU sparse matrix-matrix multiplication through lightweight analysis. In: Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM (2020). https://doi.org/10.1145/3332466.3374521
10. Stüben, K.: Algebraic multigrid (AMG). An introduction with applications. Technical report (1999). http://publica.fraunhofer.de/documents/B-73234.html
11. Stüben, K.: A review of algebraic multigrid. In: Partial Differential Equations, pp. 281–309. Elsevier (2001). https://doi.org/10.1016/b978-0-444-50616-0.50012-9
12. Thomas, S.J., Ananthan, S., Yellapantula, S., Hu, J.J., Lawson, M., Sprague, M.A.: A comparison of classical and aggregation-based algebraic multigrid preconditioners for high-fidelity simulation of wind turbine incompressible flows. SIAM J. Sci. Comput. **41**(5), S196–S219 (2019). https://doi.org/10.1137/18m1179018

13. Vaněk, P., Brezina, M., Mandel, J.: Convergence of algebraic multigrid based on smoothed aggregation. Numerische Mathematik **88**(3), 559–579 (2001). https://doi.org/10.1007/s211-001-8015-y
14. Vaněk, P., Mandel, J., Brezina, M.: Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Computing **56**(3), 179–196 (1996). https://doi.org/10.1007/bf02238511
15. Yang, U.M.: Parallel algebraic multigrid methods – high performance preconditioners. In: Bruaset, A.M., Tveito, A. (eds.) Numerical Solution of Partial Differential Equations on Parallel Computers. LNCS, pp. 209–236. Springer, Heidelberg (2006). https://doi.org/10.1007/3-540-31619-1_6
16. Yavneh, I.: Why multigrid methods are so efficient. Comput. Sci. Eng. **8**(6), 12–22 (2006). https://doi.org/10.1109/mcse.2006.125