Parallel Computing 92 (2020) 102599

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

AMG based on compatible weighted matching for GPUs[☆]

Massimo Bernaschi[a], Pasqua D'Ambra[b], Dario Pasquini[a,c,*]

[a] Institute for Applied Computing (IAC) – CNR, dei Taurini 19, Rome 00185, Italy
[b] Institute for Applied Computing (IAC) – CNR, Naples branch, Via P. Castellino, 111, 80131 Naples, Italy
[c] Department of Computer Science, "Sapienza" University, Via Salaria, 113, 00198 Rome, Italy

A R T I C L E I N F O

Article history:
Received 10 October 2018
Revised 20 November 2019
Accepted 25 November 2019
Available online 10 December 2019

MSC:
65F10
65N55

Keywords:
AMG
Graph matching
GPU

A B S T R A C T

We describe main issues and design principles of an efficient implementation, tailored to recent generations of Nvidia Graphics Processing Units (GPUs), of an Algebraic MultiGrid (AMG) preconditioner previously proposed by one of the authors and already available in the open-source package *BootCMatch: Bootstrap algebraic multigrid based on Compatible weighted Matching* for standard CPUs. The AMG method relies on a new approach for coarsening sparse symmetric positive definite (s.p.d.) matrices, named *coarsening based on compatible weighted matching*. It exploits maximum weight matching in the adjacency graph of the sparse matrix, driven by the principle of compatible relaxation, providing a suitable aggregation of unknowns which goes beyond the limits of the usual heuristics applied in the current methods. We adopt an approximate solution of the maximum weight matching problem, based on a recently proposed parallel algorithm, referred to as the *Suitor algorithm*, and show that it allows us to obtain good quality coarse matrices for our AMG on GPUs. We exploit inherent parallelism of modern GPUs in all the kernels involving sparse matrix computations both for the setup of the preconditioner and for its application in a Krylov solver, outperforming preconditioners available in the original sequential CPU code as well as the single node Nvidia AmgX library. Results for a large set of linear systems arising from discretization of scalar and vector partial differential equations (PDEs) are discussed.

1. Introduction

We are concerned with the efficient solution, on recent generations of GPU accelerators, of systems of linear equations:

$$A\mathbf{x} = \mathbf{b}, \tag{1}$$

where $A \in \mathbb{R}^{n\times n}$ is a symmetric positive definite (s.p.d.), large and sparse matrix. More specifically, we focus on the main issues and design principles driving a parallel implementation of main functionalities of the package *BootCMatch: Bootstrap algebraic multigrid based on Compatible weighted Matching* [1], for preconditioning and solving system (1) by an Algebraic MultiGrid (AMG) method based on aggregation.

AMG methods are a popular choice for dealing with a system like (1), when it results from the discretization of partial differential equations (PDEs) on complex geometries and unstructured grids or when no information about its origins are available [2–4]. The main distinguishing feature of the above methods, with respect to their geometric counterpart, is the chance of defining an automatic setup of the hierarchy of coarse-level variables and matrices by relying only on the (fine) coefficient matrix. Many variants of AMG methods [5–8] and related parallel software libraries [9–11] have been proposed in the literature. They differ in the way in which coarse-level variables are selected and in the setting of coarse-to-fine transfer operators. Despite of differences, current AMG methods show good algorithmic scalability, meaning that the number of iterations stays almost constant while the system size scales up, when they are applied to classes of sparse matrices corresponding to discretizations of 2nd order scalar elliptic PDEs.

In [12,13], the authors propose a new AMG method, and the corresponding sequential software, which relies on a new setup procedure to generate coarse-level variables aimed at obtaining an AMG preconditioner showing good algorithmic scalability for more general s.p.d. linear systems.

Here we present a parallel version of BootCMatch, which efficiently exploits the fine-grained parallelism and the memory organization of modern GPU accelerators, with the final aim to move

☆ This work has been partially supported by the EC under the Horizon 2020 Project *Energy Oriented Center of Excellence (EoCoE II): Toward Exascale for Energy*, Project ID: 824158 and INdAM-GNCS Project on *Innovative and Parallel Techniques for Large Linear and Nonlinear Systems, Function and Matrix Equations with Applications.*

* Corresponding author at: Institute for Applied Computing (IAC) – CNR, dei Taurini 19, 00185 Rome, Italy.
E-mail address: pasquini@di.uniroma1.it (D. Pasquini).

https://doi.org/10.1016/j.parco.2019.102599
0167-8191/© 2019 Elsevier B.V. All rights reserved.

a step towards AMG for future exascale computations. It is worth noting that in BootCMatch we proposed a bootstrap (adaptive) AMG aimed to obtain a preconditioner with a desired convergence rate, while in the following we do not consider adaptivity and only focus on a single hierarchy AMG based on the new setup procedure described in [12,13].

In the last 10 years a growing number of systems embeds GPU accelerators to exploit their outstanding performance (see [14]). However, these new platforms may require to rethink and redesign algorithms, data structures and software development paradigms for taking full advantage from their usage. In particular, it is not unusual for algorithms that on traditional computing platforms are considered inefficient due to slow convergence, to become very much competitive on GPUs since additional computations are well tolerated and convenient with respect to using complex memory access patterns. For example, it is well known that highly parallel smoothers, such as versions of weighted Jacobi, outperform, in terms of execution times, more effective but intrinsically sequential smoothers as Gauss-Seidel relaxation, for preconditioning and solving sparse linear systems on GPUs (e.g., see [15–17]). One of the main objectives of our work, as described in Section 3, has been to implement highly tuned kernels that access GPU global memory according to best practices of CUDA programming and to use the available computing resources (*i.e.,* CUDA cores) in a cost-effective way by introducing the concept of *miniwarp*, for all the kernels implemented in BootCMatch. The rest of the paper is organized as follows: Section 2 introduces the AMG methods and in particular the variant that relies on the solution of a weighted graph matching problem for the generation of the coarse-level variables. Section 3 describes the issues related to a parallel implementation of the AMG method based on compatible weighted matching. Section 4 provides a brief description of related works. Section 5 presents the results obtained on a large set of test cases. Finally, Section 6 concludes the work also presenting future lines of activity.

2. Background

2.1. Algebraic multigrid methods

Multigrid methods are linear complexity methods for solving system (1). They are built on a relaxation method (the *smoother*), such as a Richardson-type method, which efficiently damps high-frequency errors, although it is not able to reduce low-frequency errors. However, moving the problem to a coarser grid, what were previously low-frequency errors become high-frequency errors and can be damped by a new application of relaxation. The above procedure, whose setup requires coarser grids and transfer operators for moving among the grids, can be recursively applied obtaining methods with a computational cost which depends only linearly on the problem size [3,18]. While geometric multigrid methods rely on a pre-defined hierarchy of grids and on transfer operators depending on the geometry of the problem, AMG methods use only the information available in the system matrix. In the following, we describe the main components for setup and application of an AMG method; for an exhaustive introduction to AMG we refer the reader to [2].

Let the set of row indices of the s.p.d. matrix A be the fine index space, i.e., $\Omega = \{1, 2, \dots, n\}$. Any AMG generates a hierarchy of nl index spaces and a corresponding hierarchy of matrices,

$$\Omega^1 \equiv \Omega \supset \Omega^2 \supset \dots \supset \Omega^{nl}, \quad A^1 \equiv A, A^2, \dots, A^{nl},$$

by a suitable *coarsening algorithm* using the information contained in A. A vector space $\mathbb{R}^{n_k}$ is associated with Ω^k, where n_k is the size of Ω^k. For all $k < nl$, a prolongation operator is built $P^k \in \mathbb{R}^{n_k \times n_{k+1}}$ and the matrix $A^{k+1} = (P^k)^T A^k P^k$ is computed according to the Galerkin approach. A smoother operator M^k is also defined, representing the iteration matrix of a relaxation method. All the above components are built in the so-called *setup phase*. The components produced in the setup phase may be combined in several ways to obtain different types of *multigrid cycles*; this is done in the *application* or *solve phase*. An example of such a combination, known as symmetric V-cycle, is given in Algorithm 1 . In that case, a single iteration of the same smoother is used before and after the recursive call to the V-cycle (i.e., in the pre-smoothing and post-smoothing phases). However, more robust, although more expensive, choices can be performed, such as W-cycle [18], and recursive Krylov-based cycle (K-cycle) [19]. At the coarsest level, i.e., for $k = nl$, a direct solver is usually employed. Actually, especially in parallel implementations of the algorithm, an iterative solver of the coarsest system is also applied in order to reduce data dependencies among parallel processors.

Algorithm 1: V-cycle

```
V-cycle($k, nl, A^k, \mathbf{b}^k, \mathbf{x}^k$)
if $k \neq nl$ then
    $\mathbf{x}^k = \mathbf{x}^k + (M^k)^{-1}\left(\mathbf{b}^k - A^k\mathbf{x}^k\right)$;
    $\mathbf{b}^{k+1} = (P^{k+1})^T\left(\mathbf{b}^k - A^k\mathbf{x}^k\right)$;
    $\mathbf{x}^{k+1} = $ V-cycle$\left(k+1, A^{k+1}, \mathbf{b}^{k+1}, \mathbf{0}\right)$;
    $\mathbf{x}^k = \mathbf{x}^k + P^{k+1}\mathbf{x}^{k+1}$;
    $\mathbf{x}^k = \mathbf{x}^k + (M^k)^{-T}\left(\mathbf{b}^k - A^k\mathbf{x}^k\right)$;
else
    $\mathbf{x}^k = \left(A^k\right)^{-1}\mathbf{b}^k$;
end
return $x^k$
```

The choice of the coarse index spaces and of the prolongation operators are strictly related to each other and affects the convergence properties of Algorithm 1. Indeed, convergence strongly depends on the ability of the coarse vector spaces to accurately represent the errors unaffected by relaxation (*algebraically smooth vectors*) and the ability of the prolongators to interpolate them back to the fine space well. Recent theoretical developments provide general approaches to the construction of coarse spaces for AMG having optimal convergence, i.e., a convergence independent of the problem size, in the case of general linear systems (see [20] and the references herein). However, despite these theoretical developments, almost all currently available AMG methods and software rely on heuristics to drive the coarsening process among variables; for example the strength of connection heuristics is derived from a characterization of the algebraically smooth vectors that is theoretically well understood only for M-matrices. The above heuristics is generally used both in the *classical coarsening* and in an alternative approach, named *coarsening by aggregation* [2,8,18]. The classical coarsening separates the original index set into either coarse indices (C-indices), which form the coarse level, and fine indices (F-indices), whose unknowns will be interpolated by the C-indices values, while aggregation-based coarsening uses aggregates of fine indices to form the coarse indices. In [12,13] a new coarsening algorithm, which does not require a priori characterization of smooth vectors, has been proposed. It relies on the so-called *compatible relaxation* principle introduced in [21], which indicates a way to measure the quality of a coarse-level space, and exploits a maximum weight matching in the graph defined by the system matrix to find out an automatic aggregation-based coarsening for general s.p.d. matrices. In the following we describe the main features of the above aggregation algorithm and refer the reader to the original papers for details on the rationale and numerical principles at the base of its use for efficient coarsening.

2.2. Aggregation algorithm based on weighted graph matching

Let $G = (V, E, C)$ be the weighted undirected adjacency graph of the matrix A in (1), where the vertex set V consists of the row/column indices of A, the edge set E corresponds to the couples of indices (i, j) of the nonzero entries in A, and $C = (c_{ij})_{(i,j)\in E}$ is a matrix of positive edge weights. A *matching* in G is a subset of edges $\mathcal{M} \subseteq E$ such that no two edges in $\mathcal{M}$ share a vertex. A *maximum weight matching* in the graph G is defined as the $\arg\max_{\mathcal{M}} \sum_{(i,j)\in\mathcal{M}} c_{ij}$.

In [12,13] a maximum weight matching has been exploited to form aggregates of index pairs for good-quality coarsening in AMG methods. The main element driving the aggregation scheme is the definition of a suitable matrix $C = C(A, \mathbf{w})$ of edge weights for the adjacency graph of the original system matrix, which is function of A and of a vector $\mathbf{w} \in \mathbb{R}^n$. More specifically, C consists of positive values arising from a computation, which has linear complexity, involving the entries of A and of a good sample $\mathbf{w}$ of the algebraically smooth vectors for the system at hand. For the sake of completeness, details on the way we define the matrix C and exploit maximum weight matching in the aggregation algorithm are summarized in Appendix A. Here we observe that accurate solutions for the computation of maximum weight matching in a graph are based on the Hungarian algorithm to search optimal augmenting paths in the matrix between unmatched vertices [22]. That algorithm has a super-linear worst-case complexity and it is intrinsically sequential, therefore it represents the main issue in the search for an efficient parallel computation of a maximum weight matching. However, approximate solutions featuring near-linear complexity have been shown to represent a viable approach to obtain a more efficient matching for good-quality coarsening and some of them have been included in the BootCMatch software framework. For a detailed discussion on the impact of approximate matching algorithms on the preconditioner performance we remind to [13].

3. Parallel algorithms for GPUs

In the following we describe the design principles applied in our implementation, specifically tailored for recent generations of Nvidia GPUs using the CUDA framework, of the main kernels involved in setup and application of the compatible weighted matching AMG procedure as preconditioner in a preconditioned Conjugate Gradient (CG) method. As in the original BootCMatch code and in the Nvidia AmgX library, we chose to employ a CSR (Compressed Sparse Row) storage format for the sparse matrices.

3.1. Setup of the preconditioner

The two main issues in the development of a GPU version of the setup phase are: the computation of maximum weight matching in weighted graphs and the computation of coarse matrices.

The original CPU version offers the choice among a set of optimal and approximate maximum weight matching algorithms that, however, are either inherently sequential or unsuitable to a good GPU implementation. Therefore, we employed a different algorithm, named *Suitor*, which is a fast half-approximate matching algorithm, i.e., a matching whose total weight is at least half the optimal weight and whose cardinality is at least half the maximum cardinality. The computational complexity of the sequential Suitor is $\mathcal{O}(|E|\Delta)$, where $|E|$ is the number of edges and Δ is the maximum vertex degree in the graph G. The algorithm follows an approach based on *speculative matches* to reduce the number of candidate mates for a vertex. A vertex i proposes (tentatively matches) to its heaviest neighbor j that does not already have a proposal of heavier weight. This reduces the number of neighbors a vertex considers as candidate mates. If i has a proposal of lower weight, then i matches itself to j and unmatches the previous mate of j, i.e., k, then the algorithm has to find a new mate for k. The main feature of the Suitor algorithm, with respect to similar algorithms using a local dominant strategy, is to avoid any central queue for storing the dominant edges, therefore it is more easy to parallelize. A detailed description of the parallel Suitor algorithm is out of the scope of the present work since for our aims we relied on the parallel version for GPU proposed in [23] and made available to us from the authors in source form. For more details we refer the interested readers to [23,24].

We note that the CUDA kernels in Suitor original implementation make use of the *shuffle* instructions, a feature available starting from the *Kepler* architecture that offers a way to directly share data among threads belonging to the same warp (a group of 32 threads). The original version of warp-level primitives depended on implicit warp-synchronous behaviour that, however, is not longer guaranteed starting on CUDA 9.0. Therefore, we adapted the Suitor algorithm to the new thread-scheduling policy supported by CUDA 9.0, although CUDA supports, by using suitable compiler options, a behaviour of the warp compatible with the legacy environment. We tested both options (i) updating Suitor so that it uses the new shuffle primitives (with explicit synchronization) and (ii) using compiler options to use the legacy warp (synchronous) behaviour, but we did not find significative differences in the execution time of our code. We observe that the Suitor algorithm, as already shown for other approximate matchings available in the original CPU code [13], makes possible to obtain good quality coarsening in our AMG (see Section 5).

After optimizing parallel execution of the matching-based aggregation algorithm, as also remarked in related work (see Section 4), the most time-consuming computation remains the triple-matrix product involved in the Galerkin approach for computation of coarse matrices. In the beginning, we resorted to the standard kernel available in the *cusparse* [25] library (*cusparseDcsrmm*). However, we found that its performance was far from being optimal and we changed our code to use *Nsparse*, a recent implementation of sparse matrix-matrix product available in open source format [26]. Nsparse, as the implementation of Suitor, relies on the legacy shuffle primitives, nevertheless it provides a clear advantage with respect to the general-purpose primitives available in *cusparse*.

All other matrix operations of the setup phase, that are the computation of the transpose of the prolongator and the restriction of the current-level smooth vector, are implemented by using a load balancing technique that we refer as *miniwarps* in analogy with the group of 32 consecutive threads named *warp* in the CUDA jargon. This approach considers sets (having the same number of elements) of contiguous threads with a possible cardinality of 2, 4, 8, 16 or 32. Those sets of threads are then concurrently mapped to different data like in classic warp-centric kernels that use a warp to manage a block of contiguous data. The threads contained in a miniwarp are able to perform cooperative computation by exploiting the efficient and fine-grained intra-warp communication allowed by recent CUDA versions. During the execution, each row of the sparse matrix is assigned to a single miniwarp; that is, multiple rows are concurrently executed in the same full warp of 32 threads. The size of a miniwarp is dynamically dependent on the average number of nonzero entries per row of the sparse matrix. The main advantage of this technique is that, for matrices with few nonzero entries per row, the number of idle threads decreases. With the full warp, if a row has, on average, only $k < 32$ nonzero entries, there are, always on average, $32 - k$ threads that

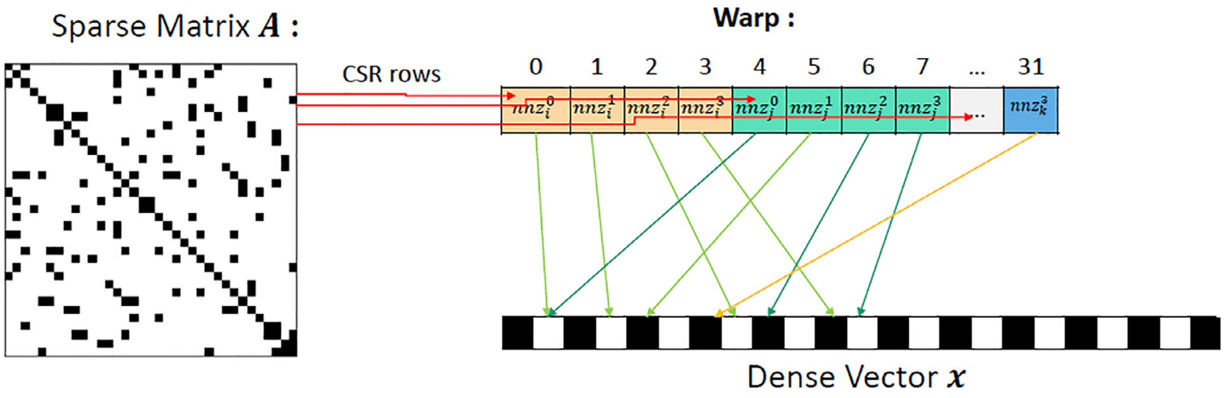

Sparse Matrix A :
CSR rows
Warp :
0 1 2 3 4 5 6 7 ... 31
nnz_i^0 nnz_i^1 nnz_i^2 nnz_i^3 nnz_j^0 nnz_j^1 nnz_j^2 nnz_j^3 ... nnz_k^3
Dense Vector x

Fig. 1. Each *miniwarp* is in charge of a row of the matrix stored in CSR format.

remain idle. Miniwarps reduce the difference significantly by using a size that is much closer to the average number of nonzero entries per row. Fig. 1 shows how the miniwarp works when applied to a matrix in CSR format in the case of a sparse-matrix dense-vector product.

3.2. Application of the preconditioner

In the preconditioner application, within the solve phase, we focused on two main kernels: the application of the smoothing in the multigrid cycle and the implementation of an optimized version of the CG method. We note that in our code we implemented a flexible version of the CG method [27], needed in the case of variable preconditioners, such as Krylov-based AMG cycles (K-cycle) [19].

While in the CPU version of the code we rely on the Gauss-Seidel relaxation method as smoother and on exact solution of the coarsest systems, in BootCMatchG we decided to avoid the intrinsically sequential computation of triangular system solution. Therefore, both for smoothing and for the solution of the linear systems at the coarsest level, we chose a version of Jacobi relaxation already used in [17] for AMG in a GPU setting. It is the so-called ℓ^1-Jacobi smoother, which is a paramater-free version of the highly parallel Jacobi method always convergent for s.p.d. matrices and having good smoothing properties for strictly diagonally dominant matrices. Our implementation of this kernel relies again on miniwarps, each miniwarp is in charge of a row and the selection of the miniwarp size follows the same criterion above defined. The most expensive computation both for ℓ^1-Jacobi smoother and for the preconditioned CG is the product between a sparse-matrix and a dense-vector that we indicate with *SpMV*, therefore we focused on the tuning of that kernel for our aims. The sparse matrix involved in a *SpMV* can be:

- a coarse matrix; in this case, no assumption can be done on the number of nonzero entries per row;
- a prolongator; in our aggregation scheme, also known as *plain or unsmoothed aggregation*, the matrix has a single nonzero entry per row;
- a transposed prolongator; the matrix has a number of nonzero entries per row that is, at most, equal to the size of the aggregates.

For the first case, depending on the sparsity degree of the matrix, the product is implemented by using either a custom kernel that relies on the concept of miniwarp or by the general-purpose *cusparse* primitive for the *SpMV*. More precisely, the *cusparse* kernel is used when the number of non-zero entries per row is, on average, at least equal to the number of threads in a full warp (i.e., 32 threads). Indeed, we observed that, when the input matrix tends to be more dense, the *cusparse* primitive performs better. The same technique is used for the transposed prolongator matrix, but in that case, the miniwarp product perfectly fits with the prolongator sparsity pattern.

For the prolongator matrix, it is possible to execute the *SpMV* more efficiently taking into account that the matrix has a single nonzero entry per row. In that case we used, for any row, a single thread. The miniwarp approach employed in the *SpMV*, compared to the first *cuSPARSE* based implementation, provides, on average, a 1.4 × speedup of the total solving time. That speedup increases up to 2.5 × when the setup phase produces a hierarchy composed by matrices with a low variance of *nnz* per row.

For an efficient implementation of the preconditioned CG method, besides optimization of the SpMV computations, we also focused on reducing the number of GPU global memory access operations by employing a version of CG, originally proposed for a distributed implementation in [28]. This version of the method is described in Algorithm 2, where the application of the preconditioner is represented by the mapping $\mathcal{B}(\cdot)$ from $\mathbb{R}^n$ to $\mathbb{R}^n$. For

Algorithm 2: Preconditioned Flexible Conjugate Gradient

```
1:  Given $u_0$ and set $r_0 = b - Au_0$
2:  $w_0 = d_0 = \mathcal{B}(r_0)$
3:  $v_0 = q_0 = Aw_0$
4:  $\alpha_0 = w_0^T r_0$
5:  $\beta_0 = \rho_0 = w_0^T v_0$
6:
7:  $u_1 = u_0 + \alpha_0/\rho_0 d_0$
8:  $r_1 = r_0 - \alpha_0/\rho_0 q_0$
9:
10: for $i = 1, \ldots$ do
11:     $w_i = \mathcal{B}(r_i)$
12:     $v_i = Aw_i$
13:
14:     $\alpha_i = w_i^T r_i$
15:     $\beta_i = w_i^T v_i$
16:     $\gamma_i = w_i^T q_{i-1}$
17:
18:     $\rho_i = \beta_i - \gamma_i^2/\rho_{i-1}$
19:
20:     $d_i = w_i - \gamma_i/\rho_{i-1} d_{i-1}$
21:     $u_{i+1} = u_i + \alpha_i/\rho_i d_i$
22:
23:     $q_i = v_i - \gamma_i/\rho_{i-1} q_{i-1}$
24:     $r_{i+1} = r_i - \alpha_i/\rho_i q_i$
25:
26: end for
```

its efficient implementation on GPU we computed the sequence of the three scalar products (see instructions from 14 till 16 in Algorithm 2) within the main loop, by using a single kernel. This approach allows us to reduce of a factor three the number of memory access operations, indeed the values of the vector **w** that is involved in all the three products can be maintained in the registers. The reordering proposed in [28] requires an additional *AXPY* computation (the update of a vector Y as $Y = \alpha X + Y$) with respect to the original version of the method, however, we grouped the *AXPY* computations in two pairs that are executed in a single kernel (see the pair of instructions 20 and 21, and the pair of instructions 23 and 24, in Algorithm 2). The results of the first *AXPY* are maintained in the GPU registers and used for the second *AXPY* operation so reducing, again, the number of GPU global memory access operations.

4. Related work

Preconditioning and solving ever more large and sparse linear systems is a key kernel in computational science and the need to exploit the potential of GPUs for parallel preconditioners in iterative linear solvers is widely recognized [29]. Due to their flexibility and potential scalability, many efforts were in particular devoted to parallel versions of AMG preconditioners specifically tailored to use single and multiple GPUs.

Some works [30–34] focused on benchmarks of well-known AMG algorithms, such as AMG based on classical C/F coarsening [5,7] and aggregation-based AMG [6,8], by using GPUs to only accelerate the application of the preconditioner at each iteration of Krylov methods. They rely on efficient implementations of the SpMV kernel and emphasize that the setup of an AMG is a bottleneck in parallel AMG methods due to the sequential nature of the coarsening processes. On the other hand, focusing on accelerating application phase of AMG is justified by the need to repeat the above application iteratively in a Krylov process. Furthermore, it is frequent the need of solving many linear systems with the same matrix but different right-hand sides, e.g., in time-dependent or in Newton-type methods, therefore the setup cost can be amortized by multiple application phases.

Early work devoted to obtain a GPU implementation of both AMG setup and application phase on a single GPU is presented in [35]. The authors describe main issues and their choices in implementing a version of the smoothed aggregation-based AMG proposed in [6]. They rely on a fine-grained parallel implementation of a generalized maximal independent set algorithm for producing aggregates with similar properties and focus on efficient kernels for the Galerkin triple-matrix product which represents the main roadblock on the way to obtain efficient AMG setup.

In [17] the authors present a GPU implementation of an unsmoothed aggregation-based AMG, where the focus is both to implement an efficient parallel algorithm for computation of maximal independent set of variables, specifically tuned for standard isotropic graph Laplacian arising in 2nd order elliptic PDEs, and to simplify the Galerkin triple-matrix product. Indeed, when standard unsmoothed aggregation is employed, the prolongation operator is a binary matrix and the Galerkin product is reduced to summations of entries in the matrix at the finer level, which can be efficiently implemented in CUDA. They also emphasize that more sophisticated cycles than the standard V-cycle, such as K-cycle, should be employed in the case of unsmoothed-type aggregation schemes in order to preserve optimal convergence of the multilevel AMG method. Furthermore, they propose to use the ℓ^1-Jacobi method both as smoother and as coarsest solver.

An efficient implementation for GPU of an unsmoothed aggregation-based AMG is also discussed in [16] and it is at the base of the GAMPACK commercial code [36] running both on single and multiple GPUs. Also in that work, the authors rely on a parallel algorithm for maximal independent set of coarse variables, representing aggregates of strictly connected fine variables, and propose to use a hybrid cycle to accelerate convergence of the application phase. They use K-cycle at the first 2 levels of the AMG hierarchy, whereas V-cycle is employed at the successive levels, in order to obtain a tradeoff between parallel efficiency and optimal convergence.

A description of the algorithms included in the publicly available Nvidia AmgX library [37], running on single and multiple-GPUs, is in [38]. AmgX implements both classical and unsmoothed aggregation-based AMG methods, with different choices for coarsening and prolongation operators. The parallel implementation of classical AMG is largely based on the methods implemented in the last available version of the hypre library [9]. The aggregation algorithm of AmgX is based on a pairwise scheme similar to that proposed in [8], coupling strongly-connected variables, which relies on a parallel graph matching techniques for efficient coarsening on single GPU. The library makes available a variety of cycles, such as V and W, and smoothers and coarsest solvers, including weighted-Jacobi, ℓ^1-Jacobi, block-Jacobi, Gauss–Seidel, and an incomplete-LU (ILU) factorization. AmgX is the state of the art of AMG preconditioners for GPUs and in this paper we consider its single-node version for our performance comparisons.

5. Numerical experiments

Hereafter, we discuss results obtained by using our GPU version of BootCMatch, named *BootCMatchG (BCMG)*, for the solution of linear systems arising from scalar and vector PDE problems, as explained in the following.

ANI These test cases derive from the following anisotropic 2D PDE on the unit square, with homogeneous Dirichlet boundary conditions:

$$-\operatorname{div}(K\,\nabla u) = f,$$

where K is the constant coefficient matrix

$$K = \begin{bmatrix} a & c \\ c & b \end{bmatrix}, \quad \text{with} \quad \begin{cases} a = \epsilon + \cos^2(\theta) \\ b = \epsilon + \sin^2(\theta) \\ c = \cos(\theta)\sin(\theta) \end{cases}$$

The parameter $0 < \epsilon \le 1$ defines the strength of anisotropy in the problem, whereas the parameter θ specifies the direction of anisotropy. In the following we discuss results related to test cases with $\epsilon = 0.001$ and $\theta = 0, \pi/8$, which we refer to as ANI1 and ANI2, respectively. The problem was discretized using the Matlab PDE toolbox, with linear finite elements on (unstructured) triangular meshes of three different sizes, obtained by uniform refinement. The resulting three linear systems have s.p.d. matrices with the size $168,577$, $673,025$, and $2689,537$, respectively.

LE2D A second set of test cases comes from the discretization of the following Lamé equations for linear elasticity:

$$\mu\Delta\mathbf{u} + (\lambda + \mu)\nabla(\operatorname{div}\mathbf{u}) = \mathbf{f} \qquad \mathbf{x} \in \Omega$$

where $\mathbf{u} = \mathbf{u}(\mathbf{x})$ is the displacement vector, Ω is the spatial domain, and λ and μ are the Lamé constants. A mix of Dirichlet boundary conditions and traction conditions are applied to have a unique solution. Discretization of the vector equation leads to systems of equations whose coefficient matrix is s.p.d. and, since each scalar component of the displacement vector is considered separately, has a block form where each diagonal block corresponds to the matrix coming from the discretization of Laplace equation for each unknown component. We considered Lamé equations on a 2D beam characterized by $\mu = 0.42$ and $\lambda = 1.7$. The problem, which we refer to as LE2D, was discretized using linear finite elements on triangular meshes of three different sizes, obtained by uniform refinement using the software package MFEM [39]. The resulting three linear systems have the size $66,690,\ 264,450,$ and $1053,186,$ respectively.

Parflow A third set of s.p.d. linear systems comes from a groundwater model, aimed at the numerical simulation of the filtration of 3D incompressible single-phase flows through anisotropic porous media. The linear systems arise from the discretization of the Darcy's equation, with no-flow boundary conditions, performed by a cell-centered finite volume scheme on a (structured) Cartesian grid. They were generated by using a Matlab code implementing the fundamentals of reservoir simulations [40] and can be regarded as simplified samples of systems arising in ParFlow, a parallel computational model developed at the Jülich Supercomputing Centre (JSC). We considered three different systems with size 10^6, corresponding to anisotropic permeability tensors randomly generated from a lognormal distribution having mean 1 and 5 different values of standard deviation, i.e., from 1 till 5, corresponding to increasing anisotropy levels, which we refer to as Mi, for $i = 1, \ldots, 5,$ respectively.

In all cases we solved the linear systems with right-hand sides set equal to the unit vector. The runs have been carried out on an Nvidia Titan V (Nvidia Volta with 12 GB and 5120 CUDA cores running CUDA 9.1), operated by IAC-CNR in Rome. For the sake of comparison between the CPU and GPU versions of the methods we also ran the original BootCMatch code on 1 core of an Intel Xeon Platinum 8176 CPU. The CPU executable has been generated using the GNU C compiler 5.5 under the control of the 3.1 Linux kernel. Comparisons with single-node version of Nvidia AmgX 2.0.0.130-open source library have been finally carried out.

We always used the AMG preconditioner coupled with our flexible version of the preconditioned CG solver (see Section 3). The CG procedure stopped when the euclidean norm of the relative residual reached the tolerance $rtol = 10^{-6}$ or the number of iterations reached a predefined threshold $itmax = 5000$. We always considered AMG hierarchies with maximum size of the coarsest matrix fixed to $maxcset \cdot n^{1/3}$, where $maxcset$ is a positive arbitrary value and n is the original matrix dimension, so that the cost of possible direct solution of the coarsest system is no larger than the cost of a matrix-vector product involving the original matrix. This is a usual way to fix the size of the coarsest system aimed to obtain a good tradeoff between the number of levels of the final hierarchy and the efficiency of the preconditioner for increasing problem size. In our tests we put $maxcset = 40,$ which is the default in BootCMatch. The same rule for the maximum size of the coarsest matrix is also used in [11]. A maximum number of levels was also fixed to 40. To maintain a maximum coarsening ratio equal to 4, we composed couples of prolongator operators computed by matching-based aggregation, resulting in double pairwise aggregates, in line with the approach implemented in [11] and also supported by AmgX [37]. Note that, for all the experiments discussed in the following, we chose the unit vector as original smooth vector involved in the aggregation algorithm described in Appendix A. In all cases one sweep of ℓ^1–Jacobi method was applied as both pre- and post-smoother whereas 20 sweeps of the same method were applied at the coarsest level.

5.1. Performance results

We first compare results obtained by applying the AMG preconditioner built by *BCMG* both as V-cycle and W-cycle. Table 1 summarizes performance results of the preconditioned CG, when V-cycle and W-cycle are applied, for the test cases ANI1 and ANI2 while system size increases. We report the execution time in milliseconds (ms), needed for the setup (*tsetup*) of the preconditioner, the number of iterations (*it*) and the time for the solution of the systems (*tsolve*). Note that the total time for the preconditioner setup and the system solution is $tsolve + tsetup$. As expected, W-cycle requires a smaller number of iterations than V-cycle, also showing a better algorithmic scalability. Indeed number of iterations increases more slowly for increasing problem size for all the ANI test cases. On the other hand, W-cycle generally has a larger computational cost per iteration, therefore the best execution times are generally obtained by applying V-cycle, but for the largest size, where the large reduction of the number of iterations results also in a lower solving time for W-cycle. Comparison between V-cycle and W-cycle are also reported for the Parflow test cases in Table 2. We observe that, also in this case, W-cycle requires a smaller number of iterations than V-cycle; however, in all cases V-cycle outperforms W-cycle in terms of execution times. Similar behaviour are observed also for the LE2D test cases, therefore, we conclude that for the available choice of parallel smoother and coarsest solver, best execution times of *BCMG* are generally obtained when V-cycle is applied.

Table 1
BCMG: ANI test cases.

Size	*tsetup*	*V-cycle*		*W-cycle*	
		it	*tsolve*	*it*	*tsolve*
ANI1					
168,577	34.15	192	115.65	123	216.30
673,025	64.60	302	464.69	161	664.84
2689,537	164.80	466	2476.52	205	2410.39
ANI2					
168,577	30.34	194	116.66	123	213.90
673,025	64.50	307	471.39	163	650.14
2689,537	167.59	481	2556.50	209	2437.63

Table 2
BCMG: Parflow test cases.

Matrix	*tsetup*	*V-cycle*		*W-cycle*	
		it	*tsolve*	*it*	*tsolve*
M1	94.46	91	206.80	26	227.54
M2	96.06	82	189.98	42	390.43
M3	96.70	125	288.63	107	995.52
M4	96.10	87	198.36	27	236.69
M5	97.83	604	1388.30	584	5569.89

5.2. Comparison with BootCMatch

In the following, we compare the performance of *BCMG* with that of the original CPU version. To the purpose of making the comparison as fair as possibile, we show results obtained by using a sequential version of the Suitor algorithm in the setup phase. In the application phase, two different configurations of the V-cycle are considered: in the first configuration, which we refer to as *BCM1*, we applied the same choices for the pre/post-smoother and the coarsest solver available in *BCMG*, i.e., 1 sweep of ℓ^1–Jacobi as both pre- and post-smoother and 20 sweeps of the same method at the coarsest level; a second configuration, referred to as *BCM2* employs 1 sweep of the more robust Gauss-Seidel smoother at the intermediate levels and exact solution, obtained by the sparse direct solver SuperLU [41] available through BootCMatch, for the coarsest system. This second choice generally gives better results in terms of number of iterations and solution time at the limited cost of an LU factorization of the coarsest matrices, as we discuss in the following. Finally, we also report comparison with the

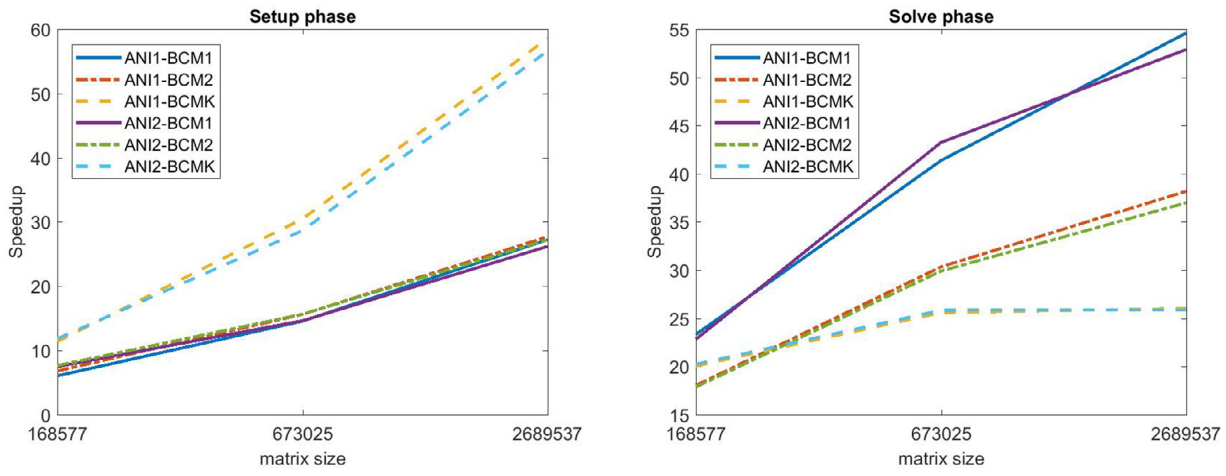


Fig. 2. Speedup of *BCMG* vs BootCMatch: ANI test cases.

original CPU version of the code, using the default parameters for both the setup and the application, i.e., the Auction-type algorithm included in BootCMatch for near-optimal maximum weight matching in the setup phase and K-cycle with the same choices of smoother and coarsest solver as in *BCM2* in the application phase; we refer to this method as *BCMK*. This default configuration of the CPU code often results in the lowest number of iterations and then of the solution time, especially for increasing problem size, although it entails a larger cost for the setup.

In Fig. 2 we show speedup of the GPU version of the preconditioner versus the two CPU versions, for the test cases ANI1 and ANI2. The speedup obtained in the setup phase (see Fig. 2 on the left) has a similar behaviour both for *BCM2* and *BCM1* in all the ANI test cases, with very small variations probably due to fluctuations in the system load. We remark that, as expected, the cost of the LU factorization of the coarsest matrices has a negligible impact on the setup cost of *BCM2* for these test cases, whereas for both *BMC2* and *BCM1* most of the setup time is spent in the triple-matrix Galerkin products and in the aggregation algorithm based on maximum weight matching. The speedup always increases with the matrix size and, when the largest system of the ANI1 test cases is considered, it reaches a maximum value of 27.2 and of 27.7 for *BCM1* and *BCM2*, respectively. Similar results are obtained also for ANI2. A good speedup is obtained also in the solve phase (see Fig. 2 on the right), where maximum values of 54.6 and 38.2 are obtained for the test cases ANI1, when *BCMG* is compared with *BCM1* and *BCM2*, respectively. Note that the speedup obtained in the solve phase by *BCMG* is better when it is compared with *BCM1* due to the reduction in the number of iterations, and then in the solution time, of the *BCM2* version of the CPU preconditioner. Indeed, the number of iterations needed to *BCM2* for the largest system size is 321 for the test cases ANI1 and 333 for the test cases ANI2, whereas *BCMG* requires 466 and 481 iterations (see Table 1) for ANI1 and ANI2, respectively, and the corresponding CPU version *BCM1* requires 461 and 471 iterations for ANI1 and ANI2, respectively. Looking at the speedup of *BCMG* versus *BCMK*, we observe maximum values of speedup in the setup phase of 58.6 and 56.6 for ANI1 and ANI2, respectively, due to the larger cost of the Auction-based matching w.r.t. the Suitor algorithm, and a maximum speedup of about 26 in the solve phase for all the ANI test cases. The number of iterations needed to *BCMK* for the largest system size is 134 for ANI1 and 136 for ANI2.

Fig. 3 shows speedup results for the LE2D test cases. In these test cases the speedup of the setup phase is slightly better when *BCMG* is compared with *BCM1*, and it reaches a value of 43.4 for the largest problem size, whereas a value of 36.5 is obtained for *BCM2*. Actually, we believe that the above difference is due to fluctuations in the system load. As a matter of fact, the time difference between the setup time of *BCM1* and *BCM2* for the largest size is only ~ 0.2 s. It is worth noting that the improvement in the speedup of the LE2D setup, when the problem size increases, with respect to the previous ANI test cases is due to the larger impact of the matching computations on the overall setup cost and shows the effectiveness of the GPU implementation of the above computation. Also in the LE2D test cases, the best speedup of *BCMG* in the setup is obtained when it is compared with the default choice of the C-code *BCMK*; in this case a speedup of 53.3 is observed for the largest problem size. The speedup of the solve phase reaches a maximum value of 46.9 when *BCMG* is compared to the corresponding CPU version of the same preconditioner (*BCM1*). An expected degradation of the performance is observed in the solve phase when BCMG is compared with *BCM2*, where a maximum speedup of 18.9 is obtained. This behaviour is due to the dramatic reduction in the number of iterations obtained when a more robust smoother is employed at the intermediate levels and the direct solver is applied on the coarsest system for these, more challenging, test cases. As a matter of fact, the number of iterations needed by *BCM2* for the largest system size is equal to 1141, whereas 2702 and 2751 are the number of iterations needed by *BCM1* and *BCMG*, respectively. The speedup of the solve phase when *BCMG* is compared to *BCMK* is increasing going from the smallest to the medium size of the problem, whereas it has a significant decrease when the largest size problem is solved, showing a value of 15.5. This behaviour is related to the large reduction of number of iterations required by *BCMK* for increasing problem size, that in this case shows the best value of 510.

Results for the Parflow test cases are shown in Fig. 4. We see that in the setup phase a speedup of about 23 is obtained by *BCMG* versus *BCM1*, whereas a speedup ranging from 23 and 24 is obtained when *BCMG* is compared to *BCM2*. Also in these test cases, the cost of the LU factorization of the coarsest matrix is negligible, therefore the behaviour of the speedup for both *BCM1* and *BCM2* is very similar. In the solve phase, we obtain a speedup ranging from 46.6 till 48 when *BCMG* is compared to *BCM1*, whereas we observe a minimum speedup of about 25.8 for the M1 test case and a maximum speedup of about 37.3 for the M5 test case when *BCMG* is compared to *BCM2*. This large variability is due to the different number of iterations needed by *BCM2* for the different systems, which is still smaller than that needed when *BCMG* and *BCM1* are applied, e.g., in the best case of M5, *BCM2* requires 432 iterations against a number of iterations equal to 604 both for *BCM1* and *BCMG*. If we look at the speedup results of *BCMG* versus

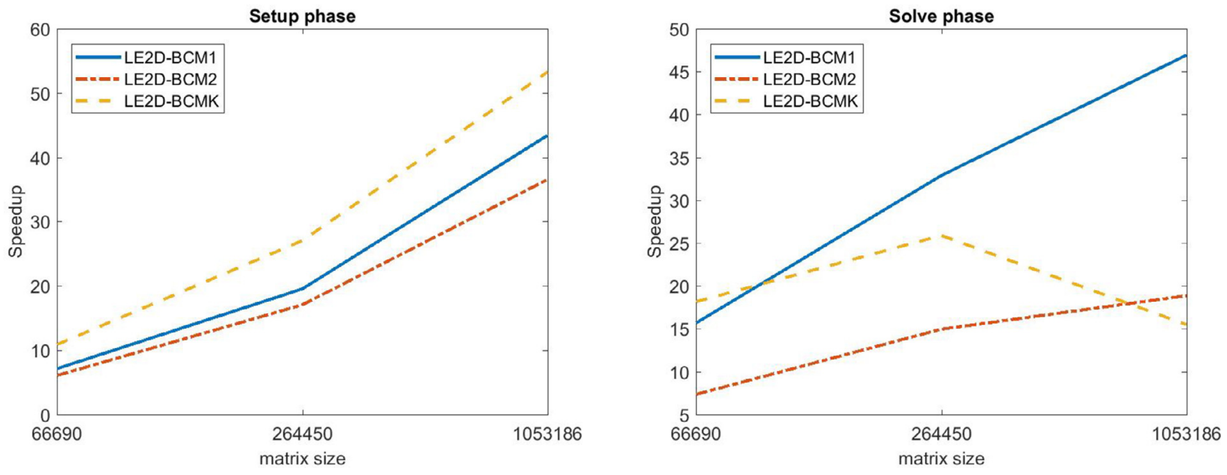


Fig. 3. Speedup of *BCMG* vs BootCMatch: LE2D test cases.

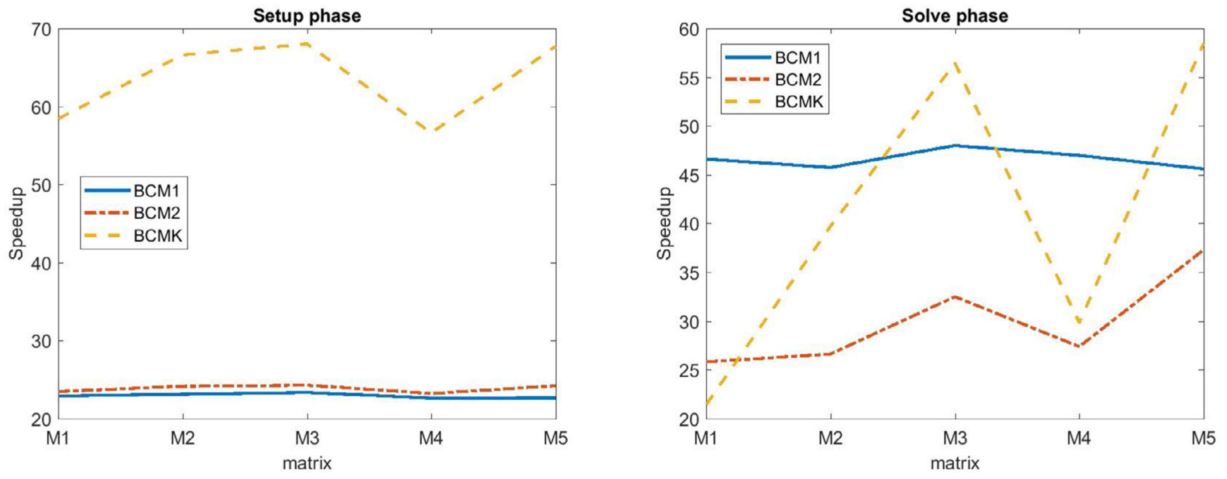


Fig. 4. Speedup of *BCMG* vs BootCMatch: Parflow test cases.

BCMK, we see that in the setup phase we obtain a speedup ranging from 56.6 and 68, whereas in the solve phase we obtain a maximum speedup of 58.5 for the M5 test case, where 600 iterations are needed to converge.

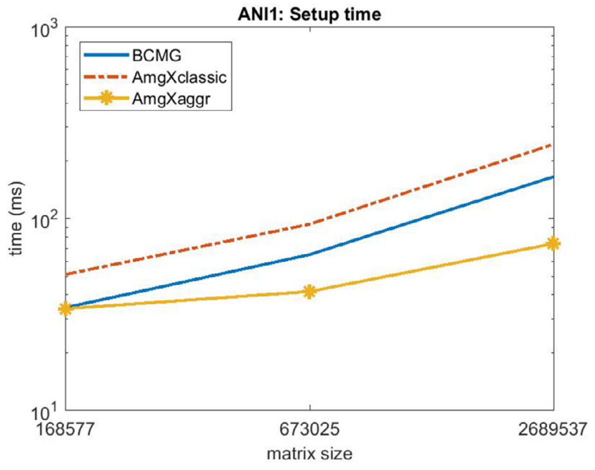
5.3. Comparison with AmgX

In this section we show a performance comparison with preconditioners implemented in the Nvidia AmgX package, when they run in a single-node setting. We considered the two different configurations available for preconditioner setup in AmgX: classical AMG and aggregation-based AMG. For classical AMG we used default configurations including *D1*-interpolation and *AHAT* strength of connection metric (see AmgX Reference Manual for details [37]), default parameters are also used for plain aggregation AMG, where aggregates of size 4 are required. We refer to them as *AmgXclassic* and *AmgXaggr*, respectively. Both the preconditioner types are applied as V-cycle within preconditioned CG iterations and the same choices for pre/post-smoother and coarsest solver applied for *BCMG* are considered for our comparisons. The parameters of Section 5 are also set for monitoring convergence of the preconditioned CG.

Note that we also applied the AmgX preconditioners as W-cycle. Furthermore, all the other available choices for smoothers and coarsest solvers have been considered for our test cases. In all cases, the best results in terms of execution times have been obtained with the AmgX preconditioners discussed in the present section.

In Fig. 5 we compare execution times for setup (left) and solve (right) phases of the preconditioners for the test cases ANI1. We observe that *AmgXclassic* shows the longest times both for setup of the preconditioner and for solving the systems; its convergence behaviour degrades significantly when the matrix size increases: the number of iterations increases from 533 up to 2885 going from the smallest to the largest size, as shown in Fig. 6 (left). Better algorithmic scalability properties are shown both for *AmgXaggr* and for *BCMG*, where a much more limited increase in the number of iterations is observed for increasing size. In particular, *BCMG* generally shows the best behaviour in terms of number of iterations, which results in the best execution times for the solve phase and in total times better than or comparable with *AmgXaggr*, as shown in Fig. 6 (right).

To gain a better understanding of our performance results, we analyzed some efficiency parameters of the various preconditioners. In Table 3, for increasing matrix size, we summarize the number of levels nl of the AMG preconditioner and the V-cycle operator complexity $Vcmplx = \frac{\sum_{k=1}^{nl} nnz(A^k)}{nnz(A^1)}$, where A^k is the matrix at level k and $nnz(A^k)$ is the number of nonzeros of A^k, which give an estimate of the cost, in terms of both memory and computation requirements, of the preconditioner. We also report the average coarsening ratio of the AMG preconditioner $cratio = \frac{1}{nl}\sum_{k=2}^{nl} \frac{n(A^{k-1})}{n(A^k)}$, where $n(A^k)$ is the size of matrix A^k, which



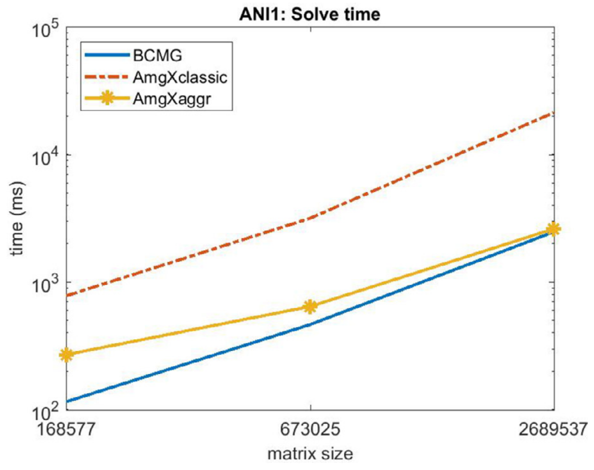
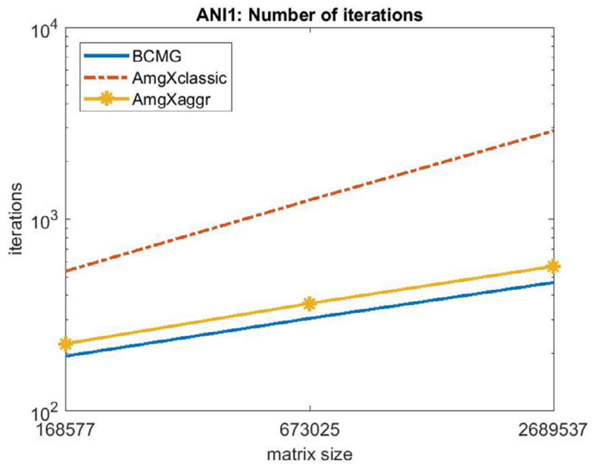


Fig. 5. AmgX vs *BCMG*: ANI1 test cases.



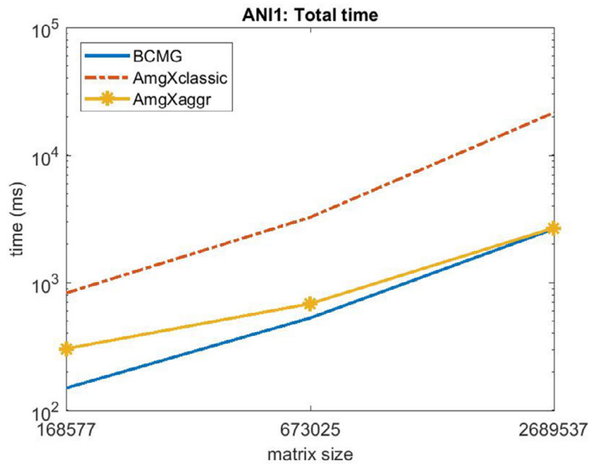


Fig. 6. AmgX vs *BCMG*: ANI1 test cases.

Table 3
AmgX vs *BCMG*: ANI1 test cases.

BCMG			*AmgXclassic*			*AmgXaggr*		
nl	*Vcmplx*	*cratio*	*nl*	*Vcmplx*	*cratio*	*nl*	*Vcmplx*	*cratio*
ANI1								
4	1.40	3.43	4	1.88	3.04	3	1.29	4.48
5	1.40	3.11	5	1.89	3.02	4	1.30	4.48
6	1.40	3.14	6	1.95	2.88	5	1.30	4.48

Table 4
AmgX vs *BCMG*: LE2D test cases.

BCMG			*AmgXclassic*			*AmgXaggr*		
nl	*Vcmplx*	*cratio*	*nl*	*Vcmplx*	*cratio*	*nl*	*Vcmplx*	*cratio*
LE2D								
3	1.42	3.55	2	1.86	3.38	3	1.37	4.05
4	1.44	3.51	3	1.90	3.63	4	1.38	3.96
5	1.44	3.21	4	1.88	3.72	5	1.38	3.85

measures the ability of the coarsening schemes to obtain efficient AMG preconditioners with few levels and a limited operator complexity.

We can observe that *AMGXaggr* is able to build an AMG preconditioner with fewer levels and smaller operator complexities, with respect to *BCMG* and *AmgXclassic*, due to its ability to obtain larger coarsening ratios. This behaviour is the main reason of the shorter setup times of *AMGXaggr*, indeed it requires one less coarsening step than the other preconditioners for all matrix sizes. On the other hand, the quality of *BCMG* appears better, indeed it requires fewer iterations for the preconditioned CG convergence leading to shorter solving times. Similar results are obtained for the ANI2 test cases, therefore, we omit them for sake of space.

In Figs. 7 and 8 we compare results obtained by the different preconditioners on the LE2D test cases. General behaviour is very similar to that obtained in the previous test cases. We observe that, also in this case, *AmgXaggr* shows better scalability for the setup of the preconditioner, whereas *BCMG* outperforms both AmgX preconditioners in the solve phase, due to the better convergence behaviour. Indeed, for all matrix sizes, *BCMG* requires the smallest number of iterations. In this case, the significant reduction in the number of iterations is able to balance the longer setup times of *BCMG*, resulting in the best total execution times.

In Table 4, we report efficiency parameters of the preconditioners for the LE2D test cases. We note that in this case, *BCMG* shows better averaged coarsening ratio than the ANI1 test cases and it is able to obtain preconditioners with the same number of levels as *AmgXaggr*. However, operator complexity of *BCMG* is yet slightly larger, showing that coarse matrices are slightly more dense than *AmgXaggr*. This behavior explains the slightly larger setup times for *BCMG*. On the other hand, the quality of our preconditioner is

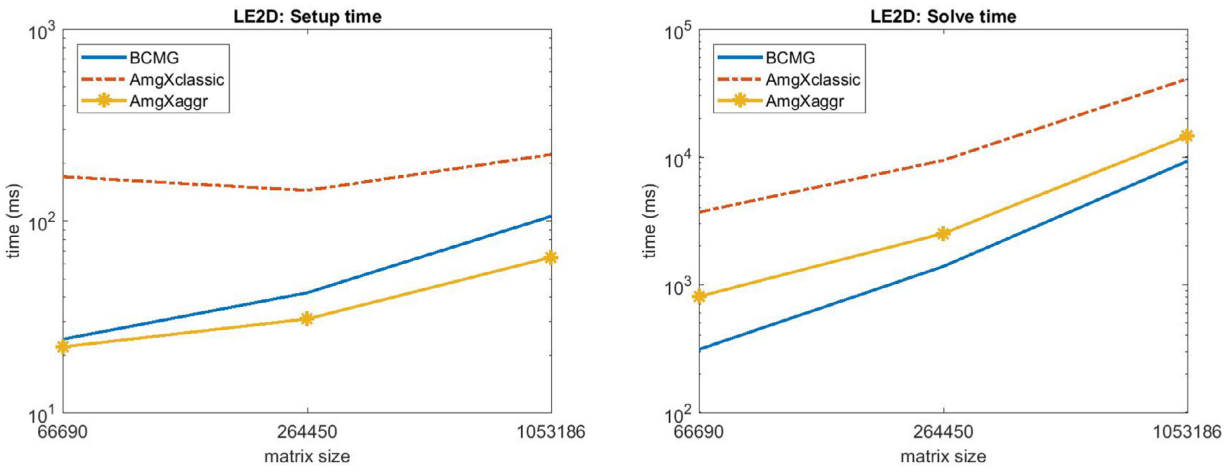


Fig. 7. AmgX vs *BCMG*: LE2D test cases.

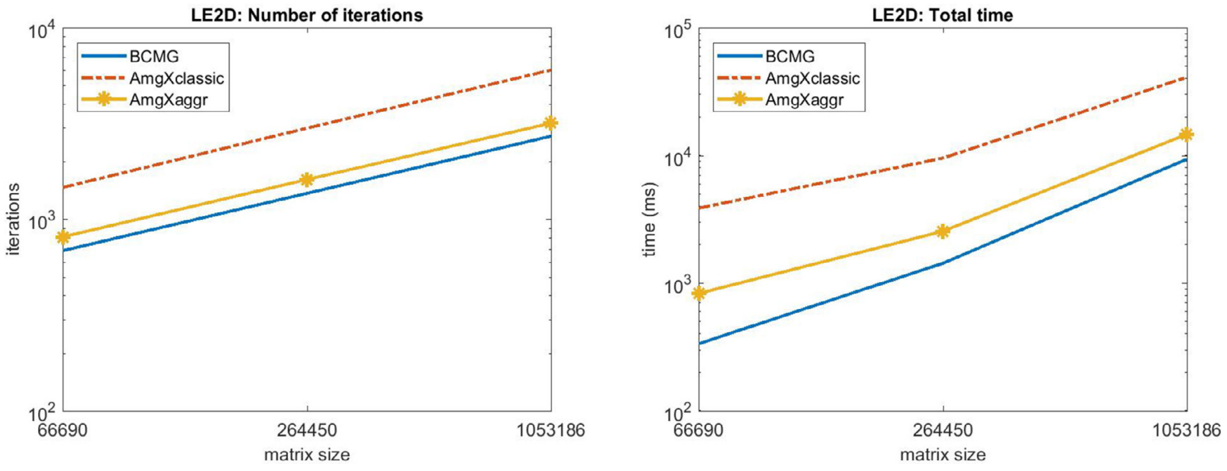


Fig. 8. AmgX vs *BCMG*: LE2D test cases.

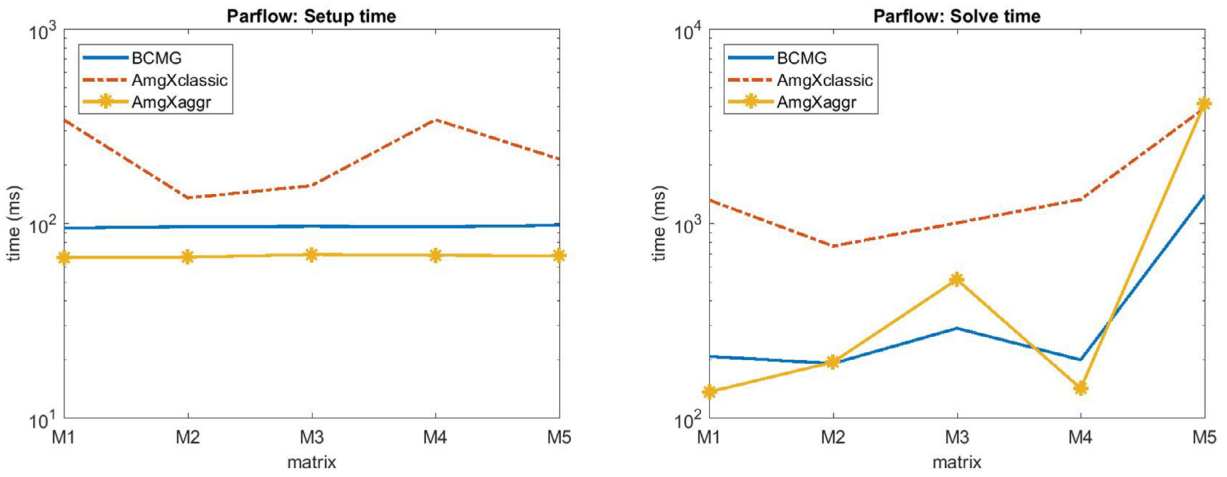


Fig. 9. AmgX vs *BCMG*: Parflow test cases.

significative better, leading to good scalability and the best total execution times.

Finally, in Figs. 9 and 10, we report performance results of the preconditioners on the Parflow test cases. We observe that, also in this case, *AmgXclassic* has the worst behavior, both in the setup and in the solve phase. In all cases *AmgXaggr* has the best setup times. However, we note that while *AmgXaggr* builds preconditioners with 4 levels for all the Parflow test cases, *BCMG*, due to smaller coarsening ratios, builds preconditioners with 6 levels. This results in a better ratio between setup times and number of levels for *BCMG*, showing a good efficiency in the implementation of the basic parallel kernels of the setup phase of our preconditioner. For M1 and M4, *AmgXaggr* requires slightly fewer iterations than *BCMG*, whereas *BCMG* largely outperforms *AmgXaggr* in the case of

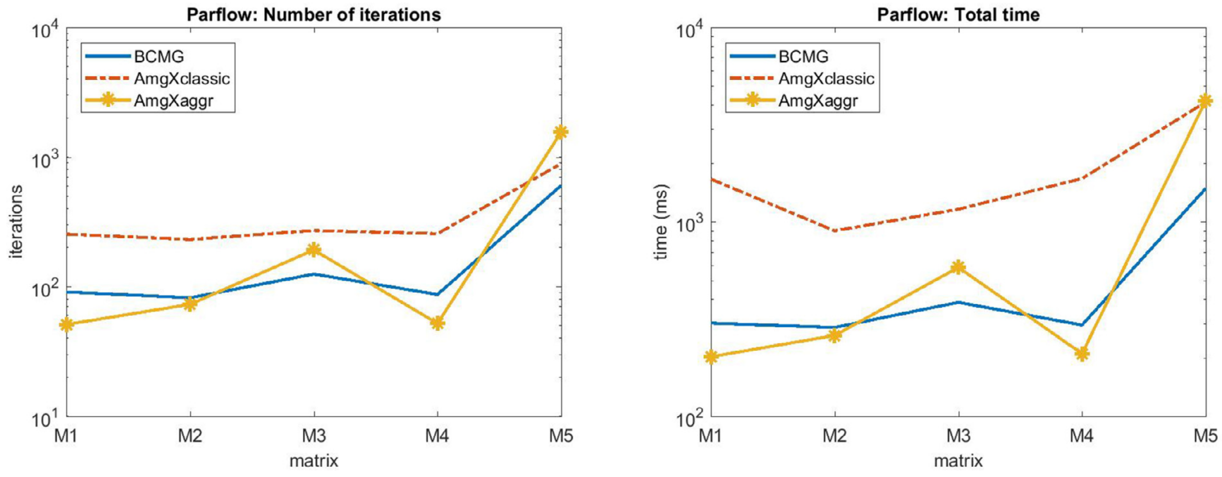


Fig. 10. AmgX vs *BCMG*: Parflow test cases.

M5, showing a better robustness with respect to anisotropy levels. We finally observe that *BCMG* generally shows the best total execution time per iteration which ranges from 2.46 ms of M5 to 3.49 ms of M2.

6. Conclusions

We presented the *BootCMatchG* package, for preconditioning and solving sparse s.p.d. linear systems on modern GPU architectures. The code implements an iterative linear solver of Krylov type coupled with an AMG preconditioner based on the so-called compatible weighted matching aggregation algorithm. We exploited fine-grained parallelism and optimized global memory access in each kernel both for the setup and the application of the AMG preconditioner, as well as in the implementation of the Krylov solver. We have rethought all main algorithms of the original package available for standard CPU, by selecting and optimizing numerical kernels for effective use of modern GPUs. To this aims, highly parallel approximate matching algorithm and a robust version of the Jacobi relaxation method were employed. Furthermore, we introduced the concept of *miniwarp* for accessing GPU global memory and using the available computing resources in an effective way. We discussed results for a large set of s.p.d. scalar and vector linear systems and demonstrated that our solver outperforms the single-node Nvidia AmgX library. Future work includes the exploitation of further parallel smoothers, such as sparse approximate inverses, and a multi-GPU version of the code.

The current version of the source code is available on request, by sending an email to one of the authors. In the near future we will make it available in a public repository.

Declaration of Competing Interest

The authors certify that they have NO affiliations with or involvement in any organization or entity with any financial interest (such as honoraria; educational grants; participation in speakers' bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

Acknowledgments

The authors wish to thank Stefan Kollet (JSC, Jülich) for making available the Matlab miniapp needed to generate Parflow test cases. We also thanks the anonymous reviewers for their comments which helped us to obtain an improved version of the paper.

Appendix A. Aggregation algorithm based on maximum weight matching

Let $\mathcal{M} = \{e_1, \dots, e_{n_p}\}$ be a matching in the adjacency graph $G = (V, E)$ of the matrix A, with n_p the number of index pairs, and let $\mathbf{w} = (w_k)_{k=1,\dots,n}$ be a given (smooth) vector; for each edge $e = (i, j)$, we can define the following two local vectors:

$$\mathbf{w}_e = \frac{1}{\sqrt{w_i^2 + w_j^2}} \begin{bmatrix} w_i \\ w_j \end{bmatrix}$$

and

$$\mathbf{w}_e^{\perp} = \frac{1}{\sqrt{w_j^2/a_{ii} + w_i^2/a_{jj}}} \begin{bmatrix} -w_j/a_{ii} \\ w_i/a_{jj} \end{bmatrix},$$

where $D_e = \begin{bmatrix} a_{ii} & 0 \\ 0 & a_{jj} \end{bmatrix}$ is the diagonal of the restriction of A to the edge e. Based on the above vectors, we can define two *prolongators*:

$$P_c = \begin{pmatrix} \tilde{P}_c & 0 \\ 0 & W \end{pmatrix} \in \mathbb{R}^{n \times n_c}, \quad P_f = \begin{pmatrix} \tilde{P}_f \\ 0 \end{pmatrix} \in \mathbb{R}^{n \times n_p},$$

where:

$$\tilde{P}_c = \text{blockdiag}(\mathbf{w}_{e_1}, \dots, \mathbf{w}_{e_{np}}), \quad \tilde{P}_f = \text{blockdiag}(\mathbf{w}_{e_1}^{\perp}, \dots, \mathbf{w}_{e_{np}}^{\perp}).$$

$W = \text{diag}(w_l/|w_l|)$, $l = 1, \dots, n_s$, is related to possible unmatched nodes in the case $\mathcal{M}$ is not a perfect matching for G_C and $n_c = n_p + n_s$.

The matrix P_c represents a piecewise-constant interpolation operator whose range includes the original (smooth) vector $\mathbf{w}$; furthermore, by construction $(P_c)^T D P_f = 0$, i.e., $\mathcal{R}ange(P_c)$ and $\mathcal{R}ange(P_f)$ are orthogonal with respect to the D-inner product on $\mathbb{R}^n$, with $D = diag(A)$. Exploiting the above decomposition, the matrix A admits the following two-by-two block form:

$$[P_c, P_f]^T A [P_c, P_f] = \begin{pmatrix} P_c^T A P_c & P_c^T A P_f \\ P_f^T A P_c & P_f^T A P_f \end{pmatrix} = \begin{pmatrix} A_c & A_{cf} \\ A_{fc} & A_f \end{pmatrix}. \tag{A.1}$$

In the above setting, given a smoother M, the relaxation scheme defined by the following error propagation matrix:

$$E_f \equiv (I - P_f (P_f^T M P_f)^{-1} P_f^T A) \tag{A.2}$$

is a compatible relaxation, i.e., it is a smoother that keeps the coarse variable invariant for the two-level method whose coarse variables ($\mathbf{x}_c = P_c^T \mathbf{x}$) are defined by the prolongator P_c. We observe that the sparsity pattern of the above prolongator is completely defined by the pairwise aggregation stemming from the matching $\mathcal{M}$, while its values depend on the (smooth) vector $\mathbf{w}$.

Relaxation defined by the matrix (A.2) is equivalent to an iteration of the form: $\mathbf{e}_{k+1} = (I - M_f^{-1} A_f)\mathbf{e}_k$, $k = 1, \dots,$ with $M_f = P_f^T M P_f$. Therefore, when matrix A_f is well conditioned or diagonally dominant, i.e., a Richardson-type relaxation method on A_f is fast convergent, the coarse variables defined by the prolongator P_c can be considered a suitable coarse set for an efficient two-level method (see [13] and references therein). To this aim, borrowing a technique widely used in sparse matrix direct solvers [22] to move large entries onto the main matrix diagonal, $\mathcal{M}$ can be chosen as a *maximum product matching*, i.e., a matching so that the product of the diagonal entries of A_f is as large as possible. Note that the diagonal of A_f is a subset of the entries of a matrix $C = C(A, \mathbf{w})$, whose entries are defined as follows:

$$\begin{aligned} c_{ij} &= \frac{1}{w_j^2/a_{ii} + w_i^2/a_{jj}} \begin{bmatrix} -w_j/a_{ii} \\ w_i/a_{jj} \end{bmatrix}^T \begin{bmatrix} a_{ii} & a_{ij} \\ a_{ji} & a_{jj} \end{bmatrix} \begin{bmatrix} -w_j/a_{ii} \\ w_i/a_{jj} \end{bmatrix} \\ &= \frac{1}{a_{jj}w_j^2 + a_{ii}w_i^2} \left(a_{jj}w_j^2 + a_{ii}w_i^2 - 2a_{ij}w_iw_j \right) \\ &= 1 - \frac{2a_{ij}w_iw_j}{a_{ii}w_i^2 + a_{jj}w_j^2}. \end{aligned}$$

The matrix C has the same adjacency graph as that of A and the computational cost for building it is $\mathcal{O}(nnz)$, where nnz is the number of nonzeros of A, or equivalently the size of the edge set E. Therefore, C is a feasible weight matrix for the edge set E to obtain a pairwise aggregation of the original vertex set, driven by a maximum product matching for the weighted graph $G = (V, E, C)$; the entries of C serve as edge weights, leading to a matrix A_f in (A.1) exhibiting (generalized) diagonal dominance. As shown in [22], maximizing the product $\prod_{(i,j)\in\mathcal{M}} c_{ij}$ is equivalent to minimizing the quantity:

$$\sum_{(i,j)\in\mathcal{M}} (\log max_i |c_{ij}| - \log |c_{ij}|), \quad c_{ij} \neq 0.$$

Then, by means of a sign change, this can be solved as a maximization problem, corresponding to the classic *maximum weight matching* or *assignment* problem.

The recursive application of the above two-level method defines the *coarsening based on compatible weighted matching* [12,13], that is a general aggregation-based AMG which does not use any a priori information on the system matrix. In the recursive application of the basic pairwise two-level method, at each new level the input weighted graph $G = (V, E, C(A, \mathbf{w}))$ corresponds to the adjacency graph of the computed coarse matrix whose weights are obtained by involving the restriction of the original vector $\mathbf{w}$ on the coarse space. More aggressive coarsening, with aggregates merging multiple pairs and having almost arbitrary large size of the type $n_c = 2^s$ for a given s, can be obtained by combining multiple steps of the basic pairwise aggregation, i.e., by computing the product of s consecutive pairwise prolongators.

References

[1] BootCMatch, Bootstrap algebraic multigrid based on compatible weighted matching, 2017, https://github.com/bootcmatch/BootCMatch.
[2] K. Stüben, Algebraic multigrid (AMG): an introduction with applications, in: U. Trottenberg, C. Oosterlee, A. Schüller (Eds.), Multigrid, Frontiers in Applied Mathematics, Academic Press, 2001. Appendix A, 413–532
[3] P.S. Vassilevski, Multilevel Block Factorization Preconditioners: Matrix-based Analysis and Algorithms for Solving Finite Element Equations, Springer, New York, USA, 2008.
[4] A. Aprovitola, P. D'Ambra, F. Denaro, D. di Serafino, S. Filippone, Scalable algebraic multilevel preconditioners with application to CFD, in: D. Tromeur-Dervout, B. Gunther, D.R. Emerson, J. Erhel (Eds.), Parallel Computational Fluid Dynamics 2008: Parallel Numerical Methods, Software Development and Applications, Lecture Notes in Computational Science and Engineering, 74, Springer, Berlin, Heidelberg, D, 2011, pp. 15–27. Invited paper.
[5] J.W. Ruge, K. Stüben, Algebraic multigrid (AMG), in: S.F. McCormick (Ed.), Multigrid Methods, Frontiers in Applied Mathematics, SIAM, Philadelphia, USA, 1987, pp. 73–130.
[6] P. Vaněk, J. Mandel, M. Brezina, Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems, Computing 56 (3) (1996) 179–196.
[7] V.E. Henson, U.M. Yang, BoomerAMG:a parallel algebraic multigrid solver and preconditioner, Appl. Numerl Mathl 41 (2000) 155–177, doi:10.1016/S0168-9274(01)00115-5.
[8] Y. Notay, An aggregation-based algebraic multigrid method, Electron. Trans. Numer. Anal. 37 (2010) 123–146.
[9] R.D. Falgout, J.E. Jones, U.M. Yang, The design and implementation of hypre, a library of parallel high-performance preconditioners, in: A.M. Bruaset, A. Tveito (Eds.), Numerical Solutions of Partial Differential Equations on Parallel Computers, Lecture Notes in Computational Science and Engineering, 15, Springer-Verlag, Berlin, Germany, 2006, pp. 267–294.
[10] M. Gee, C. Siefert, J. Hu, R. Tuminaro, M. Sala, ML 5.0 Smoothed Aggregation's Guide, Technical Report SAND2006-2649, Sandia National Laboratories, 2006.
[11] Y. Notay, User's Guide to AGMG, Technical Report, Université Libre de Bruxelles, 2018.
[12] P. D'Ambra, P.S. Vassilevski, Adaptive AMG with coarsening based on compatible weighted matching, Comput. Vis. Sci. 16 (2) (2013) 59–76, doi:10.1007/s00791-014-0224-9.
[13] P. D'Ambra, S. Filippone, P.S. Vassilevski, BootCMatch: a software package for bootstrap AMG based on graph weighted matching, ACM Trans. Math. Softw. 44 (4) (2018) 39:1–39:25, doi:10.1145/3190647.
[14] Top 500 lists, 2019, https://www.top500.org/lists/.
[15] P. D'Ambra, S. Filippone, A parallel generalized relaxation method for high-performance image segmentation on GPUs, J. Comput. Appl. Math. 293 (C) (2016) 35–44, doi:10.1016/j.cam.2015.04.035.
[16] R. Gandham, K. Esler, Y. Zhang, A GPU accelerated aggregation algebraic multigrid method, Comput. Math. Appl. 68 (2014) 1151–1160, doi:10.1016/j.camwa.2014.08.022.
[17] J. Brannick, Y. Chen, X. Hu, L. Zikatanov, Parallel unsmoothed aggregation algebraic multigrid algorithms on GPUs, in: O. Iliev, S. Margenov, P. Minev, P.S. Vassilevski, L. Zikatanov (Eds.), Numerical Solution of Partial Differential Equations: Theory, Algorithms, and Their Applications, Springer Proceedings in Mathematics & Statistics, 45, Springer, New York, USA, 2013, pp. 81–102, doi:10.1007/978-1-4614-7172-1_5.
[18] W. Briggs, V. Henson, S.F. McCormick, A Multigrid Tutorial, SIAM, Philadelphia, PA, 2000.
[19] Y. Notay, P.S. Vassilevski, Recursive Krylov-based multigrid cycles, Numer. Linear Alg. Appl. 15 (2008) 473–487, doi:10.1002/nla.542.
[20] J. Xu, L. Zikatanov, Algebraic multigrid methods, Acta Numer. 26 (2017) 591–721, doi:10.1017/S0962492917000083.
[21] A. Brandt, General highly accurate algebraic coarsening, Electron. Trans. Numer. Anal. 10 (2000) 1–20.
[22] I.S. Duff, J. Koster, On algorithms for permuting large entries to the diagonal of a sparse matrix, SIAM J. Matrix Anal. Appl. 22 (2001) 973–996, doi:10.1137/S0895479899358443.
[23] M. Naim, F. Manne, M. Halappanavar, A. Tumeo, J. Langguth, Optimizing approximate weighted matching on Nvidia Kepler K40, in: Proceedings of the IEEE Twenty-second Annual International Conference on High Performance Computing, IEEE, 2015, pp. 105–114, doi:10.1109/HiPC.2015.15.
[24] F. Manne, M. Halappanavar, New effective mutithreaded matching algorithms, in: Proceedings of the IEEE Twenty-eighth International Parallel and Distributed Processing Symposium, IEEE, 2014, pp. 519–528, doi:10.1109/IPDPS.2014.61.
[25] Nvidia, cuSPARSE, URL https://docs.nvidia.com/cuda/cusparse/index.html.
[26] Y. Nagasaka, A. Nukada, S. Matsuoka, High-performance and memory-saving sparse general matrix-matrix multiplication for Nvidia Pascal GPU, in: Proceedings of the IEEE Forty-sixth International Conference on Parallel Processing, IEEE, 2017, pp. 101–110, doi:10.1109/ICPP.2017.19.
[27] Y. Notay, Flexible conjugate gradients, SIAM J. Sci. Comput. 22 (2000) 1444–1460, doi:10.1137/S1064827599362314.
[28] Y. Notay, A. Napov, A massively parallel solver for discrete Poisson-like problems, J. Comput. Phys. 281 (2015) 237–250.
[29] R. Li, Y. Saad, GPU-accelerated preconditioned iterative linear solvers, J. Supercomput. 63 (2013) 443–466, doi:10.1007/s11227-012-0825-3.
[30] M. Wagner, K. Rupp, J. Weinbub, A comparison of algebraic multigrid preconditioners using graphics processing units and multi-core central processing units, in: Proceedings of the 2012 Symposium on High Performance Computing, HPC '12, Society for Computer Simulation International, San Diego, CA, USA, 2012, pp. 1–8.
[31] J. Kraus, M. Förster, Efficient AMG on heterogeneous systems, in: R. Keller, D. Kramer, J. Weiss (Eds.), Facing the Multicore – Challenge II, Lecture Notes in Computer Science, 7174, Springer, Berlin, Heidelberg, D, 2012, pp. 133–146.
[32] M. Emans, M. Liebmann, B. Basara, Steps towards GPU accelerated aggregation AMG, in: Proceedings of the IEEE Eleventh International Symposium on Parallel and Distributed Computing, IEEE, 2012, pp. 79–86, doi:10.1109/ISPDC.2012.19.

[33] H. Liu, B. Yang, Z. Chen, Accelerating algebraic multigrid solvers on Nvidia GPUs, Comput. Math. Appl. 70 (2015) 1162–1181, doi:10.1016/j.camwa.2015.07.005.
[34] A. Abdullahi Hassan, V. Cardellini, P. D'Ambra, D. di Serafino, S. Filippone, Efficient algebraic multigrid preconditioners on clusters of GPUs, Parallel Process. Lett. 29 (1) (2019), doi:10.1142/S0129626419500014. 1950001-1–1950001-15.
[35] N. Bell, S. Dalton, L.N. Olson, Exposing fine-grained parallelism in algebraic multigrid methods, SIAM J. Sci. Comput. 34 (2012) C123–C152, doi:10.1137/110838844.
[36] S.R. Technology, GAMPACK (GPU accelerated algebraic multigrid package), https://stoneridgetechnology.com/company/resources/gampack-gpu-accelerated-algebraic-multigrid-package.
[37] Nvidia, Algebraic multigrid solver (AmgX) library, rel. 2, 2017, https://github.com/NVIDIA/AMGX.
[38] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, R. Strzodka, AmgX: a library for GPU accelerated algebraic multigrid and preconditioned iterative methods, SIAM J. Sci. Comput. 37 (2015) S602–S626, doi:10.1137/140980260.
[39] MFEM: Modular finite element methods, mfem.org.
[40] J. Aarnes, T. Gimse, K.-A. Lie, An introduction to the numerics of flow in porous media using Matlab., in: G. Hasle, K.-A. Lie, E. Quak (Eds.), Geometric Modelling, Numerical Simulation, and Optimization: Industrial Mathematics at SINTEF, Springer-Verlag, 2007, pp. 265–306.
[41] X.S. Li, An overview of SuperLU: algorithms, implementation, and user interface, ACM Trans. Math. Softw. 31 (3) (2005) 302–325.