Parallel Processing Letters
Vol. 29, No. 1 (2019) 1950001 (15 pages)
© World Scientific Publishing Company
DOI: 10.1142/S0129626419500014

Efficient Algebraic Multigrid Preconditioners on Clusters of GPUs

Ambra Abdullahi Hassan* and Valeria Cardellini†

Dipartimento di Ingegneria Civile e Ingegneria Informatica
Università degli Studi di Roma "Tor Vergata", via del Politecnico 1, 00133 Roma, Italy
**ambra.abdullahi@uniroma2.it*
†cardellini@ing.uniroma2.it

Pasqua D'Ambra

Istituto per le Applicazioni del Calcolo "Mauro Picone"
Consiglio Nazionale delle Ricerche, via P. Castellino 111, 80131 Napoli, Italy
pasqua.dambra@cnr.it

Daniela di Serafino‡

Dipartimento di Matematica e Fisica
Università degli Studi della Campania "Luigi Vanvitelli"
viale A. Lincoln 5, 81100 Caserta, Italy
daniela.diserafino@unicampania.it

Salvatore Filippone

Centre for Computational Engineering Sciences, School of Aerospace,
Transport and Manufacturing, Cranfield University, Whittle Bldg. 52
Cranfield MK43 0AL, United Kingdom
salvatore.filippone@cranfield.ac.uk

Received 4 October 2018
Accepted 6 January 2019
Published 10 May 2019

ABSTRACT

Many scientific applications require the solution of large and sparse linear systems of equations using Krylov subspace methods; in this case, the choice of an effective preconditioner may be crucial for the convergence of the Krylov solver. Algebraic MultiGrid (AMG) methods are widely used as preconditioners, because of their optimal computational cost and their algorithmic scalability. The wide availability of GPUs, now found in many of the fastest supercomputers, poses the problem of implementing efficiently these methods on high-throughput processors. In this work we focus on the application phase of AMG preconditioners, and in particular on the choice and implementation of

‡Corresponding author.

smoothers and coarsest-level solvers capable of exploiting the computational power of clusters of GPUs. We consider block-Jacobi smoothers using sparse approximate inverses in the solve phase associated with the local blocks. The choice of approximate inverses instead of sparse matrix factorizations is driven by the large amount of parallelism exposed by the matrix-vector product as compared to the solution of large triangular systems on GPUs. The selected smoothers and solvers are implemented within the AMG preconditioning framework provided by the MLD2P4 library, using suitable sparse matrix data structures from the PSBLAS library. Their behaviour is illustrated in terms of execution speed and scalability, on a test case concerning groundwater modelling, provided by the Jülich Supercomputing Center within the Horizon 2020 Project EoCoE.

Keywords: Clusters of GPUs; algebraic multigrid; block-Jacobi smoothers; sparse approximate inverses.

1. Introduction

Many engineering and scientific applications require the solution of linear systems

$$Ax = b\,, \quad A \in \mathbb{R}^{n\times n}\,, \quad x, b \in \mathbb{R}^n\,, \tag{1}$$

where the matrix A is large and sparse. In our context, *large* means millions or even billions of unknowns, whereas *sparse* means that most coefficients in A are zero and it is convenient to employ special storage formats.

The most widely used solvers for systems of this type are Krylov subspace methods [1]. Their time to solution is determined by the number of iterations performed and by the time per iteration, and hence their efficiency is the result of a tradeoff between iteration complexity and speed of convergence. A critical feature is the application of preconditioning, corresponding, e.g., to a transformation of the form

$$M^{-1}Ax = M^{-1}b\,, \quad M \in \mathbb{R}^{n\times n}\,, \tag{2}$$

which is aimed at speeding up the convergence of Krylov methods through the improvement of the "quality" of the system matrix. Note that the product $M^{-1}A$ is not computed explicitly, but the Krylov solvers are modified so that it is obtained by performing at each iteration the computation of $M^{-1}v$, where v is a suitable vector.

Multigrid methods are among the most efficient numerical preconditioners for solving large-scale systems of equations. In particular, they are optimal, in the sense that their computational cost grows linearly with the number of unknowns, when the linear systems come from the discretization of elliptic partial differential equations [2]. Here we focus on the Algebraic MultiGrid (AMG) approach, which, unlike the geometric one, does not make explicit use of information about the problem which the linear system comes from, but exploits only the linear system, with the goal of achieving methods that can be applied to wide classes of problems [3, 4]. Furthermore, we consider systems with *symmetric positive definite* matrices, which until now have been the main target of the AMG research activity.

The linear complexity of AMG preconditioners generally translates into algorithmic scalability, i.e., the number of iterations of AMG-preconditioned Krylov solvers does not depend on the size of the problem. This allows efficient parallel implementations on multiple CPUs, for example through a domain decomposition approach, in which rows of the matrix are assigned to different computing nodes. Because of their linear complexity and parallelization scalability, AMG preconditioners are expected to be methods of choice in the emerging exascale scenario [5].

The diffusion of General Purpose Graphics Processing Units (GPGPUs), currently found in many of the fastest supercomputers in the Top 500 list (https://www.top500.org/lists/), requires the exposition of a high degree of parallelism, because GPUs are high-throughput many-core processors. Since AMG methods are obtained by combining different components (smoother, coarsening algorithm, coarsest-level solver, restriction and prolongation operators), a full exploitation of GPU capabilities requires each component to be optimized for this type of architecture.

We focus on the application phase of AMG preconditioners, and in particular on the choice and implementation of AMG smoothers and coarsest-level solvers capable of harnessing the computational power offered by a cluster of GPUs. We consider block-Jacobi smoothers and solvers that use sparse approximate inverses to perform the local solves required by each block, instead of the usual factorization methods. The choice of sparse approximate inverses is motivated by the much larger amount of parallelism exposed by sparse matrix-vector products as compared to the parallelism available in sparse triangular solves. Furthermore, suitable implementations of sparse approximate inverse preconditioners have already proved their efficiency on a single GPU (see, e.g., [6, 7]). In our work, the smoothers and local solvers are used within the AMG framework offered by the MLD2P4 package of preconditioners [8–10] and exploiting sparse matrix data structures and Krylov solvers from the PSBLAS library [11]. We conducted weak scalability tests on the JURECA supercomputer at the Jülich Supercomputing Centre (JSC), obtaining good weak scalability on up to 128 GPUs and 256 million equations.

Previous research efforts have been devoted to study AMG on GPUs, showing the acceleration that can be achieved by exploiting these devices (see, e.g., [12–18]). However, to the best of our knowledge, the AMG smoothers and local solvers we employ in this work have not been considered in other AMG preconditioners tailored for GPUs. Furthermore, in most cases, only a single GPU or small-scale GPU clusters have been considered.

In this work we do not consider the setup phase of the preconditioner, which will be the subject of future work. Of course, an efficient AMG setup on multiple GPUs is important for the overall efficiency of the preconditioner. Nevertheless, in situations where we need to solve multiple linear systems with the same coefficient matrix or sequences of linear systems with slowly varying matrices that allow reuse of the preconditioner, it is desirable to obtain the best possible efficiency during the solution phase even at the expense of a greater setup time, because this will be amortized over multiple solution steps.

The remainder of this article is organized as follows. In Sec. 2 we briefly describe AMG preconditioners, identifying their main sparse-matrix kernels. In Sec. 3 we discuss the main issues concerning the implementation of these kernels on GPUs. In Sec. 4 we illustrate the behaviour of the AMG preconditioners on a cluster of GPUs using linear systems coming from a groundwater modelling application. Finally, we provide some concluding remarks in Sec. 5.

2. AMG Preconditioners

Multigrid methods achieve their efficiency by the recursive application of two complementary processes: *relaxation* and *coarse-grid correction*. The relaxation (or smoothing) consists in the application of an iterative method, e.g., Jacobi or Gauss-Seidel, to reduce highly oscillatory error components, while the coarse-grid correction corresponds to the solution of the resulting residual equation in an appropriately chosen coarse space, aimed at reducing the leftover error components. This procedure is applied recursively, generating a sequence of spaces and corresponding residual equations of smaller and smaller size.

In the geometric multigrid approach, the coarse spaces and the associated restriction and prolongation operators, needed for transferring information from a space to the next coarser one and vice versa, are defined by the geometry of the problem. Conversely, AMG methods build the hierarchy of spaces and the corresponding transfer operators by performing an algebraic coarsening process, which uses only the entries of the system matrix, without assuming explicit knowledge of the problem which the linear system originates from. We consider an algebraic coarsening procedure based on the aggregation technique, where coarse-space unknowns are aggregates of the original unknowns. In particular, the AMG preconditioners available in the MLD2P4 library [10] use a decoupled version of the smoothed aggregation algorithm described in [19, 20]. Since the implementation on GPUs of the setup of the AMG hierarchy is beyond the scope of this work, we refer the reader to [21] for details on this aggregation algorithm and its current implementation in MLD2P4.

Here we briefly describe the so-called application phase of the AMG preconditioner, also known as multigrid cycle, which is the objective of our implementation on clusters of GPUs, using the notation introduced below.

Let

$$A_1 \equiv A, A_2, \ldots, A_{nlev} \tag{3}$$

be the hierarchy of matrices resulting from the coarsening procedure, where A_k has dimension n_k, with $n_1 > n_2 > \ldots > n_{lev}$, and $n_1 = n$, where A and n are defined in Eq. (1). We assume that the matrices have been built by using the Galerkin approach, i.e., for each level $k < nlev$,

$$A_{k+1} = P_k^T A_k P_k\,, \tag{4}$$

Algorithm 1 (V-cycle)

```
1:  procedure VCYCLE($k, A_k, b_k, x_k$)
2:      if ($k \neq nlev$) then
3:          $x_k = x_k + S_k\,(b_k - A_k x_k)$
4:          $b_{k+1} = P_k^T\,(b_k - A_k x_k)$
5:          $x_{k+1} =$ VCYCLE($k+1, A_{k+1}, b_{k+1}, 0$)
6:          $x_k = x_k + P_k x_{k+1}$
7:          $x_k = x_k + S_k^T\,(b_k - A_k x_k)$
8:      else
9:          $x_k = A_k^{-1} b_k$
10:     end if
11:     return $x_k$
12: end procedure
```

where

$$P_k \in \mathbb{R}^{n_k \times n_{k+1}} \tag{5}$$

is a linear prolongation operator from level $k+1$ to level k, and its transpose is a restriction operator from level k to level $k+1$. This is a common choice in AMG methods and is used in the MLD2P4 library. Finally, we denote by $S_k \in \mathbb{R}^{n_k \times n_k}$ the relaxation operator at level k, i.e., the pre-smoothing step has the following form:

$$x_k = x_k + S_k\,(b_k - A_k x_k)\,, \tag{6}$$

with obvious meaning of x_k and b_k, and the post-smoothing step has the same structure as in Eq. (6), with S_k^T in place of S_k. For the sake of simplicity, in Eq. (6) we describe the application of a single smoothing iteration. Different multigrid cycles can be obtained by combining the previous operators. The most widely used one is the so-called V-cycle, described in Algorithm 1.

Several relaxation methods can be chosen, defined by operators S_k with different characteristics. For example, in the Jacobi method $S_k = \mathrm{diag}(A_k)^{-1}$, where $\mathrm{diag}(A_k)$ is the diagonal matrix with diagonal entries equal to the diagonal entries of A_k; in this case, the application of S_k corresponds to a highly parallel vector update operation. More robust iterative methods, such as Gauss-Seidel and block-Jacobi, are often needed to obtain effective preconditioners; however their application requires a kernel to solve sparse triangular systems, and such a kernel is not well suited to effective implementation on GPUs (see the next section for more details). Block-Jacobi iterations also require the factorization of sparse matrices, which can be done once during the setup of the preconditioner. We also note that the solution of the coarsest-level system ($k = nlev$) must be carefully addressed in parallel settings. Direct solvers usually lead to preconditioners that are more effective in reducing the iterations of Krylov solvers, but this does not guarantee parallel efficiency. On the other hand, the iterative methods usually applied as smoothers can be used as coarsest-level solvers too, with the aim of achieving a better tradeoff between preconditioner quality and parallel performance.

3. AMG on GPUs

The term GPGPU was first introduced in connection with devices produced by NVIDIA Co. The NVIDIA GPUs are made up of scalable arrays of multithreaded, streaming multiprocessors. Each multiprocessor is composed of a fixed number of scalar processors, one or more instruction fetch units and on-chip fast memory, as illustrated in Fig. 1. The host computer is connected to the GPU device using a bus, as shown in Fig. 2, normally having a much smaller bandwidth than that of

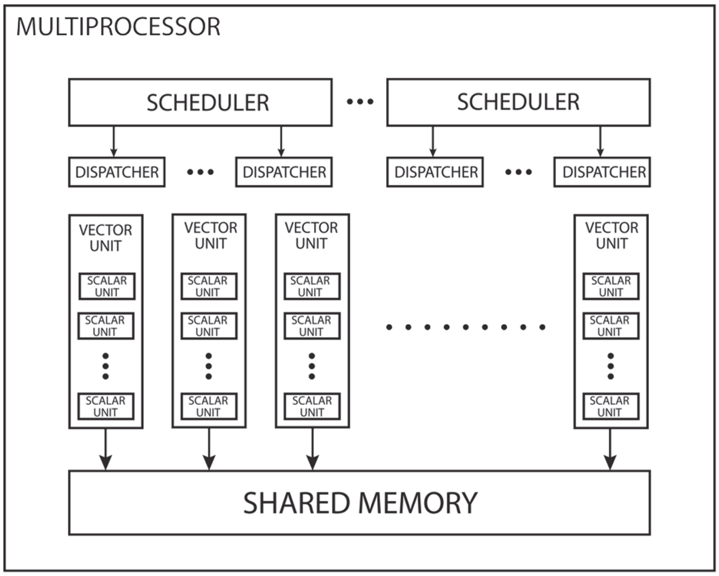

Fig. 1. GPU device: SIMT architectural model.

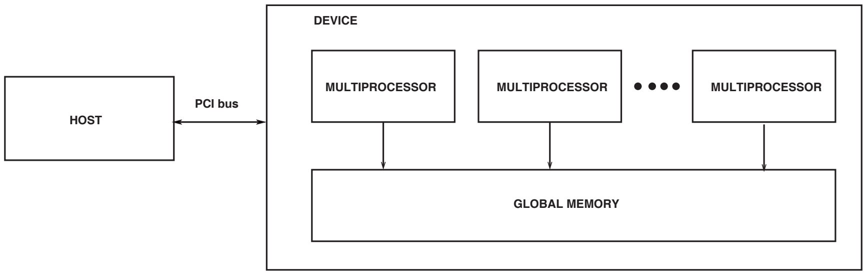

Fig. 2. GPU device to host connection.

the device global memory. The programming model proposed by NVIDIA is Single Instruction Multiple Threads (SIMT), as implemented in the CUDA programming language; a CUDA program consists of a *host* program that runs on the CPU host, and a *kernel* program that executes on the GPU *device*. Each thread has an identifier, and each vector instruction is executed on a set of threads called a *warp*.

In the sequel we will present results obtained on a parallel machine where each node is equipped with a standard (multicore) CPU and one or more GPUs. In our implementation there is one of the CPU cores acting as the *host* for each GPU *device*.

The overall data parallelization strategy is dictated by the MLD2P4/PSBLAS programming environment, i.e., a generalized row-block distribution is used [10]. In particular, the parallel matrix-vector products require on each process the availability of the so-called *halo* data, i.e., vector entries located on different processes but whose values are necessary to complete the local computation, as specified by the sparsity pattern of the matrix. From the description of the multigrid cycle in Sec. 2, it is clear that its efficient implementation on clusters of GPUs depends on the efficient implementation of the following computational kernels:

- vector sum;
- sparse matrix by vector product;
- relaxation and coarsest-level solve.

Note that the sparse matrix by vector product is involved not only in the transfer of data across the levels of the AMG hierarchy, but also in the usual computation of residuals in a Krylov method.

The GPU implementation of these kernels is handled through a plugin for the underlying PSBLAS library [11, 22]. As discussed at length in [22], an efficient implementation of these computational kernels cannot be achieved unless we restrict data transfers between the GPU device and the CPU host. Each GPU-enabled data structure (vector or sparse matrix) is equipped with memory allocated on both the host (CPU) and the device (GPU); at the start of the computation the contents of the vector or sparse matrix are usually only available on the host, but, as soon as a computational kernel is invoked, the data is copied to the device side and is subsequently kept there. All supported operations are executed purely on the GPU; vector and matrix data remain on the GPU unless the program explicitly requires to extract an external copy. Thus, for most of the computation, the only data that are regularly transferred from GPU to CPU and vice-versa include:

- scalars specified on the CPU and passed to the GPU for the execution of vector sums of the type $y \leftarrow \alpha x + \beta y$ and matrix-vector products of the type $y \leftarrow \alpha Ax + \beta y$;
- portions of vectors to implement the halo data exchange.

The sparse matrix by vector (SpMV) product is a fundamental kernel of the multigrid cycle. It is well known to be a memory-bound kernel, and its efficient implementation is the subject of an immense amount of research. For a thorough discussion of the software design of the SpMV kernel techniques we refer the reader to [23] and the references therein. In our experiments we use variants of the ELLPACK storage format, in particular the HLL format described in [23, 24] and implemented in the PSBLAS GPU plugin.

The choice and the implementation of the smoother is the most critical issue of the multigrid cycle on clusters of GPUs. Previous experience on clusters of CPUs has shown that the use of block-Jacobi iterations, either as smoothers or coarsest-level solvers, allows to obtain AMG preconditioners that achieve a good balance between acceleration of Krylov solvers and parallel performance [9, 25–28]. In this case the relaxation operator is the block-diagonal matrix

$$S_k = \mathrm{diag}(A_k^1, A_k^2, \dots, A_k^p)^{-1}\,, \tag{7}$$

where A_k^i is the diagonal block of A_k corresponding to the block of rows assigned to the compute node i in the data distribution. This requires the inversion of each block A_k^i, which can be carried out by performing a Cholesky factorization followed by triangular solves. In the context of preconditioning, a sparse incomplete factorization is usually computed, obtaining an approximation of A_k^i. As already observed, the incomplete factorization is carried out only once, during the setup phase of the preconditioner, while the triangular solves must be performed at each relaxation and coarsest-level solution.

Unfortunately, the efficient implementation of sparse triangular solves on GPUs is a difficult task. For example, in [29] it is reported that Krylov solvers preconditioned with the incomplete LU and Cholesky factorizations available in the CUDA cuSPARSE library achieve on average only a speedup of 2 over the CPU implementation provided by the Intel MKL library; furthermore, the sparsity pattern of the matrix strongly affects the performance of the triangular solves. The difficulties associated with sparse triangular solves on GPUs are confirmed by the study in [30], where it is also shown that the performance of a sparse triangular solver on a GPU is strongly dependent on the features of the system matrix, while it is almost independent of the floating-point precision or the GPU architecture employed. The main problem is that GPUs require a massive amount of data parallelism to be exploited, and the sparsity of the triangular factors produces strong data dependencies that limit the amount of available parallelism and yield unbalanced computations. Moving towards a denser matrix would improve the triangular solve, but would be undesirable in terms of memory footprint, time to setup and time to apply the preconditioner.

Since the sparse matrix-vector product is much more amenable to an efficient implementation on GPUs, it is quite appealing to use sparse approximate inverses for the local blocks of the block-Jacobi methods,

$$(A_k^i)^{-1} \approx Z_k^i D_k^i (Z_k^i)^T\,, \tag{8}$$

with Z_k^i upper triangular and D_k^i diagonal, since their application requires matrix-vector products.

The approximate inverses are computed with a plugin for MLD2P4 that has been described in [6, 31]. In particular, implementations of the following methods are available:

- INVK(I,J): the approximate inversion of an ILU factorization based on pattern-levels; for example, an INVK(0,1) would start with an ILU(0) factorization, and then compute an approximate inverse of its factors with 1 level of fill-in [32];
- INVT($\epsilon_1, n_1, \epsilon_2, n_2$): a similar approximate inverse of triangular factors computed through ILUT(ϵ, n) [32];
- AINV-LLK: the method of biconjugation proposed by Benzi and Tuma [33, 34], modified as described in [6].

The plugin also provides an interface, called AINV, to the biconjugation software by the authors of [33, 34]. In most of our experiments the INVK method achieves the best trade-off between convergence properties, setup, and application speed.

The previous kernels have been combined by using the AMG framework provided by the MLD2P4 library, to get multigrid cycles suitable for clusters of GPUs. This has been made possible by the modular architecture of MLD2P4, which has allowed the extension with new data storage formats and new local solvers for block-Jacobi iterations, as well as the use of the PSBLAS GPU plugin.

We conclude this section by observing that two limiting factors affect the efficiency of multilevel preconditioners on clusters of GPUs. The first issue has to do with the implementation of the matrix-vector product on a single GPU: as mentioned in [23], to run at full speed on a GPU, the matrices must be sufficiently large to keep all computing units busy, and this becomes increasingly difficult as we go through the levels of the hierarchy, because the size of the space becomes smaller. The second issue is related to the efficiency of the communications between the various computing nodes; to proceed with the parallel computation of the matrix-vector product on the different nodes, it is necessary to exchange the data of the halo. For good data partitions, the amount of data to be exchanged displays a surface-to-volume effect, and therefore large products can be computed effectively. However, when moving towards smaller matrices in the preconditioning hierarchy, the communication to computation ratio, and hence the efficiency of the parallelization, decreases. Note that we most often use smoothed aggregation, which improves the convergence features, but has the negative side effect of increasing communication.

4. Results

We illustrate the behaviour of the AMG preconditioner described so far on linear systems arising from a groundwater modelling application developed at the Jülich Supercomputing Centre (JSC) and made available in the framework of the Horizon 2020 Project EoCoE (Grant Agreement no. 676629). This application is concerned with the numerical simulation of the filtration of 3D incompressible single-phase flows through porous media. The linear systems come from the discretization, over the unit cube, of an elliptic equation modelling the pressure field, which is obtained by combining the continuity equation with Darcy's law; a homogeneous permeability tensor is considered and no-flow boundary conditions are imposed. The discretization is performed by a cell-centered finite volume scheme (two-point flux approximation) on a Cartesian grid and the resulting matrix is symmetric positive definite, with nonzero entries distributed over seven diagonals [35]. The computational domain is uniformly partitioned along the three coordinate planes, obtaining a row-block distribution of the matrices which minimizes the surface-to-volume ratio. We performed *weak scalability* tests, keeping approximately 2 million equations per process. The results obtained are representative of the behaviour of the AMG preconditioner on linear systems coming from more general isotropic elliptic equations.

The experiments were carried out using the cluster module of the JURECA supercomputer at JSC, comprising 75 compute nodes with 2 NVIDIA Tesla K80 GPUs with a dual-GPU design, 1872 compute nodes with 2 Intel Xeon E5-2680 v3 Haswell CPUs per node, and a Mellanox EDR Infiniband network. PSBLAS 3.5, combined with the extended matrix formats and GPU plugins, and MLD2P4 2.1, combined with the AINV plugin, were installed using the GNU 5.4.0 C and Fortran compilers, MVAPICH2 2.3 and CUDA 8.0.61. The tests were run using both CPUs and GPUs, as well as CPUs only, to quantify the gain in efficiency due to the use GPUs. Up to 128 GPUs were used for our experiments, thus the dimensions of the linear systems range from 2 million to 256 million.

The linear systems were solved using the GPU implementation of the Conjugate Gradient (CG) method provided by PSBLAS with the GPU plugin, coupled with two versions of the V-cycle preconditioner: one using 1 sweep of the block-Jacobi method as pre-smoother and as post-smoother, with INVK(0,1) as local solver, and the other using 2 sweeps of the simple Jacobi method in each smoothing phase. In the following, we refer to the previous smoothers as BJAC(INVK) and JAC2, respectively. In both versions, 10 sweeps of BJAC(INVK) were applied as coarsest-level solver. The multilevel hierarchy was built by running (on CPUs) the decoupled smoothed aggregation algorithm mentioned in Section 2, using its default parameters. The resulting number of levels was 4, except for 128 processing elements (i.e., for the largest matrix), where 5 levels were obtained. The zero vector was chosen as starting guess and the CG iterations were halted as as soon as the ratio between the 2-norm of the residual and the 2-norm of the right-hand side became smaller than 10^{-6}. The HLL sparse matrix format was used when running the experiments on CPUs plus GPUs, while the CSR format was used for tests on CPUs only.

Figure 3 shows the execution times, in seconds, required by the solution of the linear systems on GPUs and on CPUs, using the two V-cycle versions described above. We see that on GPUs the use of BJAC(INVK) and JAC2 as smoothers leads to about the same execution time, while BJAC(INVK) appears more expensive on

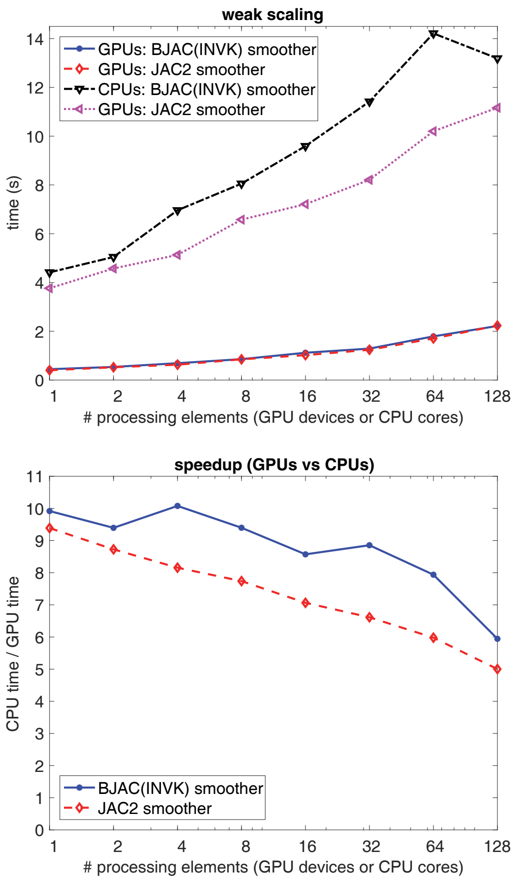

Fig. 3. Solution of the linear systems using CG with the two versions of V-cycle: weak scaling on GPUs and CPUs (top) and speedup of GPUs vs CPUs (bottom).

CPUs. Furthermore, there is a significant gain in using GPUs vs CPUs: the speedup over CPUs ranges from 5.9 to 10 when BJAC(INVK) is used as smoother and from 5 to 9.4 when JAC2 is applied. In our opinion, this is a satisfactory result, given the memory-bound nature of the computation and the decreasing size of the matrices as the AMG hierarchy is traversed, which decreases the amount of computation on GPUs and increases the percentage of data to be exchanged among the processing elements.

To better compare the two V-cycles, we report, in Table 1, the number of CG iterations obtained with the two versions of V-cycle as the number of processing elements varies. Of course, the number of iterations is the same for the GPU and the CPU implementations, since the same coarsening strategy is run on CPUs in both cases. We observe that the number of iterations corresponding to BJAC(INVK) is smaller than the number of iterations with JAC2. This compensates for the larger time required by a single smoothing step with BJAC(INVK). We expect that

Table 1. CG iterations with the two versions of V-cycle.

# procs	CG iterations	
	BJAC(INVK)	JAC2
1	14	19
2	15	21
4	18	20
8	20	25
16	24	27
32	29	31
64	34	37
128	29	31

BJAC(INVK) is more efficient with linear systems requiring more robust smoothers. We also note that the increase in the number of CG iterations is modest when going from 1 to 64 processing elements; furthermore, there is a slight decrease on 128 processing elements. This confirms that the use of BJAC(INVK) iterations at the coarsest level does not deteriorate too much the effectiveness of the preconditioner as compared to the application of a direct solver, while it allows good scalability. The small reduction of the iteration count on 128 processing elements depends on the higher density of the diagonal blocks at the fifth level of the AMG hierarchy, which leads to better approximations by INVK(0,1). We also forced the coarsening algorithm to generate 4 levels on 128 processing elements, obtaining a larger coarsest matrix. In this case, BJAC(INVK) was less effective at the coarsest level, leading to an increase in the number of CG iterations. However, while the time on CPUs increased, the time on GPUs became slightly smaller, because the diagonal blocks of the larger coarsest-level matrix expose more parallelism in matrix-vector products on GPUs.

5. Conclusions

We have presented an implementation of the application phase of AMG preconditioners for clusters of GPUs, obtained by integrating efficient sparse-matrix kernels for GPUs in the framework of the MLD2P4 package. The main issue has been the choice of smoothers and coarsest-level solvers able to achieve a good tradeoff between exploitation of the computational power of GPUs and communication among the processing elements. The results obtained on linear systems from a groundwater simulation, made available within the Horizon 2020 EoCoE project, show from 5$\times$ to 10$\times$ increase of speed versus CPUs and good scalability up to 128 GPUs and 256 million equations. Future work will be devoted to the implementation of the AMG setup phase, focusing on the coarsening algorithm and the construction of the hierarchy of matrices.

It is worth noting that the integration of GPU-tailored sparse-matrix kernels in the modular architecture of MLD2P4 has led to good results without the need to restructure the application phase of the AMG preconditioners. Although a complete re-coding in CUDA of the AMG preconditioners can in principle lead to higher efficiency in exploiting GPU capabilities, we believe that our approach is quite useful, because it allows running on GPU devices with close-to-optimal efficiency while at the same time making optimal reuse of existing software.

Acknowledgments

This work was supported by the Energy oriented Centre of Excellence (EoCoE), Grant Agreement no. 676629, funded within the Horizon 2020 framework of the European Union. We wish to thank Stefan Kollet and Wendy Sharples (JSC, Jülich) for providing the test problem and access to the JURECA supercomputer. We also thank the anonymous reviewers for their useful comments, which helped us improve the quality of this work.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd edition (SIAM, Philadelphia, PA, 2003).

[2] P. S. Vassilevski, *Multilevel Block Factorization Preconditioners: Matrix-based Analysis and Algorithms for Solving Finite Element Equations* (Springer, 2008).

[3] K. Stüben, A review of algebraic multigrid, *J. Comput. Appl. Math.* **128**(1) (2001) 281–309.

[4] J. Xu and L Zikatanov, Algebraic multigrid methods, *Acta Numerica* **26** (2017) 591–721.

[5] J. Park, M. Smelyanskiy, U. M. Yang, D. Mudigere and P. Dubey, High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems, in *Proc. of Int'l. Conf. on High Performance Computing, Networking, Storage and Analysis (SC'15)* (2015).

[6] D. Bertaccini and S. Filippone, Sparse approximate inverse preconditioners on high performance GPU platforms, *Comput. Math. Appl.* **71**(3) (2016) 693–711.

[7] M. Lukash, K. Rupp and S. Selberherr, Sparse approximate inverse preconditioners for iterative solvers on GPUs, in *Proc. of 2012 Symp. on High Performance Computing (HPC'12)* (2012), pages 13:1–13:8.

[8] P. D'Ambra, D. di Serafino and S. Filippone, On the development of PSBLAS-based parallel two-level Schwarz preconditioners, *Appl. Numer. Math.* **57**(11) (2007) 1181–1196.

[9] A. Buttari, P. D'Ambra, D. di Serafino and S. Filippone, 2LEV-D2P4: A package of high-performance preconditioners for scientific and engineering applications, *Appl. Algebra Engrg. Comm. Comput.* **18**(3) (2007) 223–239.

[10] P. D'Ambra, D. di Serafino and S. Filippone, MLD2P4: A package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95, *ACM Trans. Math. Softw.* **37**(3) (2010) 7–23.

[11] S. Filippone and A. Buttari, Object-oriented techniques for sparse matrix computations in Fortran 2003, *ACM Trans. Math. Softw.* **38**(4) (2012) 23:1–23:20.

[12] G. Haase, M. Liebmann, C. C. Douglas and G. Plank, A parallel algebraic multigrid solver on graphics processing units, in *High Performance Computing and Applications* (Springer, 2010), pages 38–47.

[13] N. Bell, S. Dalton and L. N. Olson, Exposing fine-grained parallelism in algebraic multigrid methods, *SIAM J. Sci. Comput.* **34**(4) (2012) C123–C152.

[14] M. Emans, M. Liebmann and B. Basara, Steps towards GPU accelerated aggregation AMG, in *Proc. of 11th Int'l. Symp. on Parallel and Distributed Computing* (*ISPDC 2012*) (2012), pages 79–86.

[15] M. Wagner, K. Rupp and J. Weinbub, A comparison of algebraic multigrid preconditioners using graphics processing units and multi-core central processing units, in *Proc. of 2012 Int'l. Symp. on High Performance Computing* (*HPC'12*) (2012), pages 2:1–2:7.

[16] R. Gandham, K. Esler and Y. Zhang, A GPU accelerated aggregation algebraic multigrid method, *Comput. Math. Appl.* **68**(10) (2014) 1151–1160.

[17] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan and R. Strzodka, AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods, *SIAM J. Sci. Comput.* **37**(5) (2015 S602–S626.

[18] H. Liu, B. Yang and Z. Chen, Accelerating algebraic multigrid solvers on NVIDIA GPUs, *Comput. Math. Appl.* **70**(5) (2015) 1162 –1181.

[19] J. Mandel, P. Vaněk and M. Brezina, Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems, *Computing* **56** (1996) 179–196.

[20] M. Brezina and P. Vaněk, A black-box iterative solver based on a two-level Schwarz method, *Computing* **63** (1999) 233–263.

[21] P. D'Ambra, D. di Serafino and S. Filippone, *MLD2P4 User's and Reference Guide*, version 2.1 (2017). Available from https://github.com/sfilippone/mld2p4-2.

[22] V. Cardellini, S. Filippone and D. Rouson, Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms, *Sci. Program.* **22**(1) (2014) 1–19.

[23] S. Filippone, V. Cardellini, D. Barbieri and A. Fanfarillo, Sparse matrix-vector multiplication on GPGPUs, *ACM Trans. Math. Softw.* **43**(4) (2017) 30:1–30:49.

[24] D. Barbieri, V. Cardellini, A. Fanfarillo and S. Filippone, Three storage formats for sparse matrices on GPGPUs, Technical Report DICII RR-15.6 (Università di Roma Tor Vergata, 2015). http://hdl.handle.net/2108/113393.

[25] A. Borzì, V. De Simone and D. di Serafino, Parallel algebraic multilevel Schwarz preconditioners for a class of elliptic PDE systems, *Comput. Vis. Sci.* **16**(1) (2013) 1–14.

[26] P. D'Ambra, D. di Serafino and S. Filippone, Performance analysis of parallel Schwarz preconditioners in the LES of turbulent channel flows, *Comput. Math. Appl.* **65**(3) (2013) 352–361.

[27] A. Aprovitola, P. D'Ambra, F. M. Denaro, D. di Serafino and S. Filippone, SParC-LES: Enabling large eddy simulations with parallel sparse matrix computation tools, *Comput. Math. Appl.* **70**(11) (2015) 2688–2700.

[28] P. D'Ambra and S. Filippone, A parallel generalized relaxation method for high-performance image segmentation on GPUs, *Journal of Computational and Applied Mathematics* **293**(C) (2016) 35–44.

[29] M. Naumov, Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. Technical Report NVR-2011 1, NVIDIA Co., 2011.

[30] D. Erguiz, E. Dufrechou and P. Ezzatti, Assessing sparse triangular linear system solvers on GPUs, in *Proc. of 2017 Int'l. Symp. on Computer Architecture and High Performance Computing Workshops* (*SBAC-PADW*) (IEEE, 2017), pages 37–42.

[31] D. Bertaccini and S. Filippone, Approximate inverse preconditioners for Krylov methods on heterogeneous parallel computers, in *Parallel Computing: Accelerating Computational Science and Engineering*, eds. M. Bader, A. Bode, H. Bungartz, M. Gerndt, G. Joubert and F. Peters (IOS Press, 2014), pages 183–192.

[32] A. C. N. van Duin, Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices, *SIAM J. Matrix Anal. Appl.* **20**(4) (1999) 987–1006.

[33] M. Benzi and M. Tůma, A sparse approximate inverse preconditioner for nonsymmetric linear systems, *SIAM J. Sci. Comput.* **19**(3) (1998) 968–994.

[34] M. Benzi, J. K. Cullum and M. Tůma, Robust approximate inverse preconditioning for the conjugate gradient method, *SIAM J. Sci. Comput.* **22**(4) (2000) 1318–1332.

[35] J. E. Aarnes, T. Gimse and K.-A. Lie, An introduction to the numerics of flow in porous media using Matlab, in *Geometric Modelling, Numerical Simulation and Optimization*, eds. G. Hasle, K.-A. Lie and E. Quak (Springer, 2007), pages 265–306.