Efficient Algebraic Multigrid Preconditioners on Clusters of GPUs

Ambra Abdullahi Hassan[1], Valeria Cardellini[1], Pasqua D'Ambra[2], Daniela di Serafino[3], and Salvatore Filippone[4]

[1] Università degli Studi di Roma "Tor Vergata", Roma, Italy
{ambra.abdullahi,cardellini}@uniroma2.it
[2] Istituto per le Applicazioni del Calcolo "Mauro Picone", CNR, Napoli, Italy
pasqua.dambra@cnr.it
[3] Università degli Studi della Campania "Luigi Vanvitelli", Caserta, Italy
daniela.diserafino@unicampania.it
[4] Cranfield University, Cranfield, UK
salvatore.filippone@cranfield.ac.uk

Abstract Many scientific applications require the solution of large and sparse linear systems of equations using iterative methods, this operation accounting for a large percentage of the computing time. In these cases, the choice of the preconditioner is crucial for the convergence of the iterative method. Given the broad range of applications, great effort has been put in the development of efficient preconditioners ensuring algorithmic scalability: in this sense, multigrid methods have been proved to be particularly promising. Additionally, the advent of GPUs, now found in many of the fastest supercomputers, poses the problem of implementing efficiently these algorithms on highly parallel architectures; this is made more difficult by the fact that the solution of sparse triangular systems, a common kernel in many types of preconditioners, is extremely inefficient on GPUs.
In this paper, we use the PSBLAS and MLD2P4 libraries to explore various issues that affect the efficiency of multilevel preconditioners on GPUs, both in terms of execution speed and exploitation of computational cores, as well as the algorithmic efficiency in guaranteeing convergence to solution in a number of iterations independent of the parallelism degree. We investigate these issues in the context of linear systems arising from groundwater modeling application of the filtration of 3D incompressible single-phase flows through porous media.

1 Introduction

2 Algebraic MultiGrid (AMG) Preconditioners

MultiGrid methods are widely used as preconditioners of iterative Krylov solver for sparse linear systems. They are very efficient methods, having linear computational complexity and hence perfect scalability, when sparse and large linear

systems stemming from the discretization of some scalar elliptic Partial Differential Equations (PDEs) have to be solved [12]. Many efforts are related to extend their efficient applicability also to more general systems, in particular in the direction of a complete algebraic approach, where no information on a possible geometric origin of the problem is exploited. In this last case, they are named Algebraic MultiGrid (AMG) or Algebraic Multilevel methods [11]. MultiGrid methods rely their efficiency on the recursive application of two complementary processes: relaxation and coarse-grid correction.

Relaxation consists in the application of an iterative method, such as 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. In the classical multigrid approach, the coarser grid and the interpolation operator for transfer the coarse-grid solution to the original (fine) grid are predefined by the geometry of the problem. On the contrary, AMG methods address this setup phase, known as *coarsening process*, in an automatic way without using any explicit knowledge of the given problem and relying only on system matrix entries. In this work we refer to an algebraic coarsening process based on aggregation of unknowns, where coarse-grid unknowns are agglomerates of the original unknowns. In particular, AMG preconditioners available in MLD2P4 rely on a decoupled version of *the smoothed aggregation* algorithm described in [9,2]. This procedure is currently implemented on the host CPUs, therefore, for sake of space, we refer the reader to [3] for more details on the algorithm and its parallel implementation. Our main aim here is to describe all the main Linear Algebra operations needed for the application of the AMG preconditioner within an iterative linear solver, since their efficient implementation on GPUs was the main focus of this work.

The application phase of an AMG preconditioner is also known as multigrid cycle, the most widely used is the so called symmetric V-cycle, described in Fig. 1, where an AMG hierarchy of *nlev* prolongator operators P^k and of corresponding coarse matrices obtained by the standard variational approach $A^{k+1} = (P^{k+1})^T A^k P^{k+1}$ have been built in the setup phase. **Immagino che la formulazione del sistema lineare Ax=b (con A spd) sia stata definita prima da qualche parte**. We observe that in Fig. 1, M^k represents the matrix operator corresponding to the basic iterative method applied at the level k for relaxation. Main computational kernels in the application of the preconditioner are then sparse matrix-vector multiplications and inversion of the matrix operator M^k. In the simple case of Jacobi method, $M^k = diag(A^k)$, therefore it corresponds to a highly parallel vector update operation. On the other hand, more robust iterative methods, such as the Gauss-Seidel method or incomplete factorizations, are often required for improving convergence rate of the preconditioner in the solution of general linear systems. In that cases the inversion of the corresponding matrix operators require the solution of triangular systems that is an intrinsically sequential kernel. Therefore, an efficient parallel implementation of the V-cycle application is strictly related to an efficient implementation of the

```
procedure V-cycle(k, A^k, b^k, x^k)
  if (k ≠ nlev) then
    x^k = x^k + (M^k)^{-1} (b^k − A^k x^k)
    b^{k+1} = (P^{k+1})^T (b^k − A^k x^k)
    x^{k+1} = V-cycle(k + 1, A^{k+1}, b^{k+1}, 0)
    x^k = x^k + P^{k+1} x^{k+1}
    x^k = x^k + ((M^k)^T)^{-1} (b^k − A^k x^k)
  else
    x^k = (A^k)^{-1} b^k
  endif
  return x^k
end
```

Figure 1: V-cycle preconditioner.

sparse matrix-vector multiplication and to the availability of iterative methods which can be formulated in terms of that kernel and possible vector updates. **Immagino che l'utilita' di una fase di applicazione particolarmente efficiente, causa necessita' frequente di applicazione in metodi non lineari e/o risoluzione di sistemi multipli sia stata evidenziata nell'in troduzione**.

3 Sparse Linear Algebra and Preconditioners on GPUs

General Purpose Graphics Processing Units (GPGPUs) [7] are today an established and attractive choice in the world of scientific computing, found in many among the fastest supercomputers on the Top 500 list, and offered in standard Cloud infrastructure services, e.g. in Amazon EC2. It is therefore not surprising that they are at the focus of many research efforts revolving around the efficient implementation of scientific software, including solvers for sparse linear system of equations.

Many applications use iterative methods to solve sparse linear systems; the most popular ones are those based on Krylov subspace projection methods [10].

From a software point of view, all Krylov methods employ the matrix A to perform matrix-vector products $y \leftarrow Ax$, whose efficient implementation provides significant challenges on all modern computer architectures. The SpMV kernel is well-known to be a memory-bounded application; and its bandwidth usage is strongly dependent on both the input matrix and on the underlying computing platform(s). For a detailed overview of the implementation issues and a survey of available techniques for this kernel on GPUs, see [6].

For most real-world problems,it is necessary to combine the Krylov solvers with *preconditioners*. A preconditioner is a transformation applied to the coefficient matrix such that the resulting linear system has better convergence properties. Examples of popular preconditioners include Jacobi and Gauss-Seidel iterations, Block Jacobi and Additive Schwarz coupled with incomplete factorizations, and Algebraic Multigrid.

To implement a linear system solver on a parallel computing architecture, we typically use a *domain decomposition* approach, in which subsets of the computational domain are assigned to different computing nodes. Domain decomposition requires to achieve a number of trade-offs among multiple factors:

- Many preconditioners, e.g. Gauss-Seidel or incomplete factorizations, employ kernels for the solution to sparse triangular systems; in using sparse triangular solvers, we normally want the sparsity pattern to be of the same order as that of the original coefficient matrix;
- However, the amount of parallelism available in a sparse triangular solution is dependent on the number of nonzeros in the sparse factors, and is usually very small;
- To overcome this issue, most preconditioners use a Block Jacobi or hybrid Gauss-Seidel strategy, in which the sparse triangular solve is used only within the local subdomain, while employing a global Jacobi-like correction;
- Unfortunately, the convergence properties of these local schemes deteriorate with increased parallelism.

The last phenomenon is often described as a loss of *algorithmic scalability*, i.e., the ability to obtain a convergence history that is independent of the degree of parallelism. In summary, simple preconditioning schemes, such as Point Jacobi, have good algorithmic scalability, because their convergence properties do not change much with an increasing number of processes, but the more sophisticated Gauss-Seidel and incomplete factorizations are a lot more efficient in serial computations.

Multilevel corrections, or Algebraic Multigrid preconditioners, offer a way to recover algorithmic scalability: it is possible to apply efficient local approximate solvers, while a good convergence history is recovered by relying on the multilevel corrections.

Finding the preconditioning combination that gives the best overall performance is thus a very complex enterprise. In this context, the ability to combine multiple options in a flexible framework such as the one we described in [4] is an essential ingredient.

Additional considerations are necessary when implementing preconditioning operations on GPUs. First of all, implementing the solution of sparse triangular systems on GPUs is extremely difficult. As an example, the conjugate gradient preconditioned with incomplete LU available in CUDA CuSPARSE since version 4.0 [8] achieves at best a speedup factor of about 2 over a standard CPU implementation. The main reason is that the GPU requires a massive amount of data parallelism to be exploited, and in the sparse triangular factors the amount

of available parallelism is limited by the sparsity. The two main options that are available involve the use of matrix-vector products as their main kernel:

- Point-Jacobi iterations;
- (Local) preconditioning through sparse approximation of the inverses.

For a discussion of sparse approximate inverses on GPUs see [1].

The implementation of the multilevel preconditioners for which we will present results in this paper is based on MLD2P4 [4] and PSBLAS [5]. Thanks to the modular architecture of both packages, it is possible to define *plugins* for

- Operators on the GPU, including sparse matrix-vector product supporting multiple storage formats and vector-vector operations;
- Approximate inverses as local solvers.

By combining these ingredients in a multilevel framework we can obtain interesting scalability results.

Three additional practical obstacles must be mentioned here. First of all, the preconditioner is invoked at least once per iteration of the Krylov method. Because the allocation of memory on the GPU is an expennse synchronization point, this means that the memory space for internal work areas must be preallocated for all invocations of the preconditioner; allocation of auxiliary data areas at each invocation of the prevonditioner is perfectly doable on the CPU side, but leads to a slowdown by a factor of 2 on the GPU.

The second issue has to do with the implementation of the matrix-vector product on the GPU: as mentioned in [6], the most effective storage formats are often derived from ELLPACK and assign one thread per row of the matrix; this means that to exploit a GPU it is necessary to have matrices which are sufficiently large to keep all computing units busy, and this is very difficult at the coarse level of the preconditioner hierarchy because the size of the aggregates is small. Indeed, it may be convenient to limit the number of levels to be less than the default that would be used in a normal CPU implementation. **DA VERIFICARE CON I DATI DI PERFORMANCE**.

The third issue is related to the efficiency of communication between the various computing nodes; to proceed with the computation of the matrix-vector product in parallel between 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 between different levels of the preconditioning hierarchy we are precisely going towards smaller matrices, hence the surface-to-volume ratio and the efficiency of the parallelization decreases. The speed ratio of the GPU with respect to the CPU only makes things worse; again, it may be necessary to limit the number of levels to less than what would be efficient when using CPUs.

4 Results

(Distribuzione dati fatta tenendo conto della geometria (3D) del problema e giustificandola rispetto al vantaggio che fornisce in termini di rapporto calcolo/comunicazione.)

We illustrate the behaviour of the AMG preconditioner and the INVK solver on GPUs by showing the results obtained on a linear system arisen from a groundwater modelling application developed at the Juelich Supercomputing Centre (JSC) dealing with numerical simulation of the filtration of 3D incompressible single-phase flows through anisotropic porous media. It is an elliptic equation with no flow boundary conditions. The linear systems arises from the discretization of the equation performed by a cell-centered finite colume scheme (two-point flux approximation) on a Cartesian grid, with nonzero entries distributed over seven diagonals. In these tests, we will consider a homogeneous permeability tensor. Weak scalability tests are performed, using approximately 2 millions equations per process. The total dimension ranges from 2 millions up to 256 millions of equations **??**. This application comes from the framework of the Horizon 2020 EoCoE Project

We ran experiments on the JURECA supercomputer at the Juelich Supercomputing Centre (JSC). Each GPU compute node consists of two NVIDIA Tesla K80 GPUs with a dual-GPU design, for a total of four available GPU devices per compute nodes. We used PSBLAS 3.5 (combined with its extension plugin for extended matrix formats and GPU plugin) and MLD2P4 2.1 (combined with AINV plugin), compiled with GNU compilers (C and Fortran) version 5.4.0, MVAPICH2 version 2.3 and with CUDA 8.0.61.

Linear systems are solved using Conjugated Gradient (CG) and the stopping criterion is based on the ratio bewteen 2-norms of the residual and the right-hand side vector (when smaller than 10^{-6}). CG is use in conjuction with a V-cycle preconditioner. We consider two different configurations. In the first case the V-cycle preconditioner applies 1 pre and post sweep of Block Jacobi smoother with INVK (approximate inverse based on the inversion of triangular factors based on a positional drop strategy) as a solver. In the second configuration it applies 2 per and post sweeps of Jacobi solver. In both cases, we use 10 sweeps of Block Jacobi with INVK as coarsest level solver. For each configuration, we ran tests only on CPUs (with matrices in CSR format) and using CPUs+GPUs (with matrices in ELG format). This allows us to quantify the gain in efficiency in using GPUs.

First, we show results for the case using BJAC(INVK) smoother on the inner levels. In table 2 the last column shows the speed-up of the solver phase on the gpu with respect to the cpu solve time.

Secondly, the results for the case using two sweep of a Jacobi smoother on inner levels.

Table 1: Numerical results for CG + ML preconditioner, runs on CPUs, with 1 sweep of BJAC(INVK) as smoother on inner levels.

np	levels	It	Prec time (s)	Solve Time (s)	Time per Iteration (s)	Total Time (s)
1	4	14	17.74	4.42	0.32	22.15
2	4	15	19.43	5.05	0.34	24.47
4	4	18	21.85	6.96	0.39	28.81
8	4	20	28.32	8.05	0.40	36.37
16	4	24	26.38	9.59	0.40	35.98
32	4	29	23.52	11.42	0.39	34.93
64	4	34	29.61	14.21	0.42	43.82
128	5	29	28.39	13.19	0.45	41.58

Table 2: Numerical results for CG + ML preconditioner, runs on GPUs, with 1 sweep of BJAC(INVK) as smoother on inner levels.

np	levels	It	Prec time (s)	Solve Time (s)	Time per Iteration (s)	Total Time (s)	Speed up cpu/gpu
1	4	14	18.80	0.45	0.03	19.24	9.92
2	4	15	20.55	0.54	0.04	21.09	9.40
4	4	18	22.84	0.69	0.04	23.54	10.08
8	4	20	29.18	0.86	0.04	30.03	9.40
16	4	24	27.38	1.12	0.05	28.50	8.57
32	4	29	23.74	1.29	0.04	25.02	8.85
64	4	34	30.53	1.79	0.05	32.32	7.94
128	5	29	29.48	2.22	0.08	31.70	5.94

Table 3: Numerical results for CG + ML preconditioner, runs on CPUs, with 2 sweeps of JACOBI as smoother on inner levels.

np	levels	It	Prec time (s)	Solve Time (s)	Time per Iteration (s)	Total Time (s)
1	4	19	2.83	3.76	0.20	6.59
2	4	21	3.65	4.57	0.22	8.23
4	4	20	4.21	5.13	0.26	9.35
8	4	25	4.75	6.58	0.26	11.33
16	4	27	4.97	7.21	0.27	12.18
32	4	31	4.88	8.21	0.26	13.09
64	4	37	5.79	10.20	0.28	16.00
128	5	31	6.72	11.16	0.36	17.88

5 Conclusions

References

1. Bertaccini, D., Filippone, S.: Sparse approximate inverse preconditioners on high performance GPU platforms. Computers & Mathematics with Applications 71(3),

Table 4: Numerical results for CG + ML preconditioner, runs on GPUs, with 2 sweeps of JACOBI as smoother on inner levels.

np	levels	It	Prec time (s)	Solve Time (s)	Time per Iteration (s)	Total Time (s)	Speed up cpu/gpu
1	4	19	3.13	0.40	0.02	3.53	9.39
2	4	21	4.06	0.52	0.02	4.59	8.73
4	4	20	4.65	0.63	0.03	5.28	8.16
8	4	25	5.20	0.85	0.03	6.05	7.74
16	4	27	5.39	1.02	0.04	6.42	7.06
32	4	31	5.35	1.24	0.04	6.59	6.61
64	4	37	6.25	1.71	0.05	7.96	5.98
128	5	31	7.19	2.23	0.07	9.43	5.00

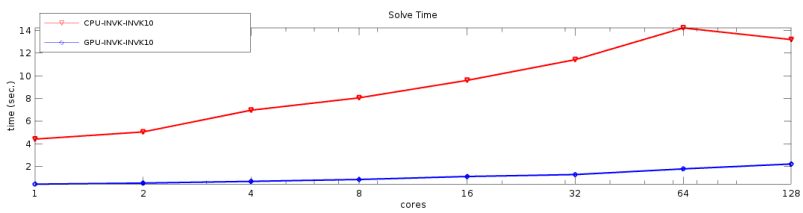

Figure 2: INVK-INVK

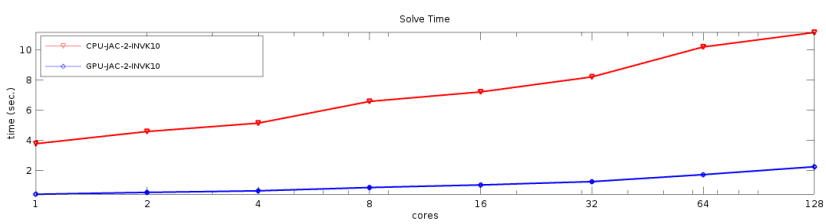

Figure 3: JAC2-INVK

693–711 (2016)
2. Brezina, M., Vaněk, P.: A black-box iterative solver based on a two-level schwarz method. Computing 63, 233–263 (1999)
3. D'Ambra, P., di Serafino, D., Filippone, S.: MLD2P4 User's and Reference Guide, version 2.1 (September 2017), available from https://github.com/sfilippone/mld2p4-2
4. D'Ambra, P., di Serafino, D., Filippone, S.: MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. ACM Trans. Math. Softw. 37(3), 7–23 (2010)
5. Filippone, S., Buttari, A.: Object-oriented techniques for sparse matrix computations in Fortran 2003. ACM Trans. Math. Softw. 38(4), 23:1–23:20 (2012)
6. Filippone, S., Cardellini, V., Barbieri, D., Fanfarillo, A.: Sparse matrix-vector multiplication on gpgpus. ACM Trans. Math. Softw. 43(4), 30:1–30:49 (Jan 2017), http://doi.acm.org/10.1145/3017994
7. Luebke, D., Harris, M., Govindaraju, N., Lefohn, A., Houston, M., Owens, J., Segal, M., Papakipos, M., Buck, I.: GPGPU: general-purpose computation on graphics hardware. In: Proc. of 2006 ACM/IEEE Conf. on Supercomputing. SC '06 (2006)
8. Naumov, M.: Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS. Tech. rep., NVIDIA Corporation (June 2011)
9. P. Vaněk, J.M., Brezina, M.: Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Computing 56, 179–196 (1996)
10. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2nd edn. (2003)
11. Stüben, K.: An introduction to algebraic multigrid. In: Multigrid (2001)
12. Vassilevski, P.S.: Multilevel block factorization preconditioners: matrix-based analysis and algorithms for solving finite element equations. Springer, New York, USA (2008)