Paper Title

***Abstract*—**
***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Very often numerical simulations from applied science and engineering imply the solution a sparse and large linear system of equations. This happens in particular when a finite elements or finite difference discretization of partial differential equations (PDEs) is involved. Very often it accounts for a great percentage of the whole computing time. A sparse linear system is one of the form: $Ax = b$ where A is a sparse matrix. According to Wilkinson's definition a sparse matrix is 'Any matrix with enough zeros that it pays to take advantaged of them'. This is the case of matrices originating by PDEs, where usually the number of entries $a_{i,j}$ per row different from zero are much smaller that the number of variables. Of course, Wilkinson definition depends on the algorithm the user wishes to employ [1]. This includes both the use of an ad-hoc storage format or a specialized method.

The notion of 'large' expands of order of magnitudes by time and now it is not uncommon to have linear systems of equations with hundreds of millions or even billion of unknowns. In these cases, when it is essential to achieve good efficiency on parallel architectures, direct methods such as Gaussian elimination are not feasible and iterative solvers are preferred. Iterative methods produce a sequence of approximate solutions ideally converging the real one, starting from a given approximate solution. The convergence rate depends upon the condition number of the matrix and iterative solvers are likely to suffer from slow convergence when used in practice. The success of iterative methods lies in their application in conjugaction with a preconditioner [2] that can improve their efficiency and robustness. A preconditioner is a trasnformation of the original linear system into one wich is equivalent (has the same solution) but which is more likely to achieve fast convergence with an iterative solver. Usually, the choice of the preconditioner determines the success of the method. Among iterative methods Krylov methods are probably the most popular. They are based on minimization of the residuals over a Krylov subspace [3]. The Conjugated Gradient (CG) [4], the Generalizide Minimal Residual (GMRES) [5] and the Biconjugated Gradient (BiCG) [6] are all Krylov subspace methods. Multigrid methods, based on the creation of a hierarchy of coarse matrices, are also a popular choice. They can be used as stand-alone methods [7] or as preconditioners [8] and they have the nice property that they can achieve convergence rates which are, in theory, independent of the mesh size [9]. With their widespread diffusion, the interest in accelerating sparse solvers using GPUs has constantly increased. Given their high peak computational throughput, programmer need to expose fine-grain parallelism in sparse linear algebra operations, thus rethinking both sparse storage schemes and classical preconditioners.

A. Multilevel preconditioners

Multigrid methods are among the most popular choices when solving sparse linear systems arising from the discretization of elliptic PDEs.They show algorithmic scalability, which means that they have a computational cost per iteration which is independent of the problem size. This nice property makes them particularly suited for high performance computers. At the basic of multigrid methods there is the observation that most relaxation-type methods cause a rapid dump of the components of the errors in the direction of eigenvectors corrispondin to large eigenvalue (high frequency-mode) while it is extremely difficult to damp compenents corresponding to small eigenvalue (low frequency-mode). Multigrid methods exploits the fact that low frequency modes are naturally mapped onto high-frequency mode in a coarser mesh. Thus the idea is to move the residual to a coarse space to corresponding error components. Another way of thinking of a multilevel method is in terms of the recursive application of a two-level method. This consists of:

1) Pre-smoothing iterations: In this phase an approximate solution of the fine level linear system is computed.
2) Restriction of the residual onto a coarse level space.
3) Solution of the coarse level linear system
4) Project the error back to the fine level space and update of the solution
5) Post-smoothing iterations.

Pre and post smoothing iterations can be performed using both simple iterative methods such as Jacobi or Gauss-Seidel or complex subspace-correction methods such as Schwarz smoothers. When solving the coarse level linear system, a two-level method can be applied leading to the multilevel method. In the setup phase the coarse space, the coarse level matrix as well as interpolation operators are determined. In this phase we can distinguish between geometric multigrid (where information about the physical grid is used to build the coarse level space) and algebraic multigrid (where only information about the coefficient matrix is used). The latter is a more general approach in which we can see the multilevel methods as a 'black blox'. In the following, we will focus only on algebraic multigrid (AMG) methods. We can make a further distinction according to the coarsening strategy: in classic AMG the coarse space is built by splitting variables in coarse and fine points while in aggregation AMG coarsening takes place by aggregating two or more fine level variables.

Aggregation AMG usually has lower operator complexity but at expense of convergence rate. To counterbalance it, a smoothed aggregation is used. Another strategy is to combine the unsmoothed aggreation with a K-cycle [10]. In the following we will consider only aggregation AMG.

Assuming we are building a hierarchy of matrices $A^k \equiv A, A^1, ..., A^{n_{lev}}$ once we have determined the coarse variables for level k and calling n_k the size of the coarse level space, prolongator and restrictor operators can be determined.:

$P^k \in R^{n_k x n_{k+1}}, P^k \in R^{n_{k+1} x n_k}$

and the coarse level matrix can be computed according to the Galerkin approach:

$A^{k+1} = R^k A^k P^k$.

where usually $R^k = (P^k)^T$

B. aggregation based on maximum weighted matching

C. Approximate Inverse

Most of the methods used as smoothers in a multilevel method, are based on an incomplete LU factorization. The basic idea is to compute an approximate factorization $A \approx LU$ where L and U lower an upper triangular matrices respectively. An approximate solution can be computed by solving the sparse triangular linear system $Ly = b$ and $Ux = y$ and the matrix $M = LU$ can be used as a preconditioner in an iterative solver. Unfortunately, implementing an efficient triangular solver for sparse matrices is an extremely difficult task on highly parallel architectures such as GPUs. As an example, conjugated gradient preconditioned with incomplete LU factorization is already available in CUDA CuSPARSE, but it achieves at best a speedup factor of about 2 [11]. Another approach is to prondition the linear system $Ax = b$ by using a direct approximation of A^{-1}. This approach is called approximate inverse (AINV) and has been proved to show better results in preconditioning of iterative methods on GPUs. The main advantage of this approach is that the main kernel for the application of an AINV preconditioner is a Sparse Matrix-Vector multiplication (SpMV); this has two main implications:

1) sparse matrix vector multiplication is also the main kernel in the application of a Krylov iterative methods. This means that efficient kernel for AINV is a by-product of the effort to implement Krylov methods.
2) On GPUs is quite possible to implement efficiently SpMV, but implementation for triangular systems is much less efficient.

There are different algorithms for computing approximate inverse. Some versions are still based on sparse inversion of triangular factors using some drop strategy. We can call these versions INVK and INVT. Considering the INVK variant, it needs two input parameters $INVK(N_1, N_2)$:

1) First, we perform an incomplete factorization $ILU(K)$ with $K = N_1$. Input parameter N_1 sets the level of fill in this first phase.
2) Secondly, we perform the approximate sparse factorization. Input parameter N_2 sets the inversion accepting fill-in for N_2 levels.

Considering that the inverse of a sparse matrix is usually full, it is usually convenient to allow additional fill in the inversion phase rather than in the factorization phase. This strategy helps in avoiding to store unnecessary nonzeros entries. The case of INVT is analogous, with four parameters regulating the drop threshold and the number of additional nonzeros ($INVT(N_1, \epsilon_1, N_2, \epsilon_2)$)

Approximate inverse plug-in is available in MLD2P4 and it can be combined with the GPU plugin in PSBLAS. This allows us to use approximate inverse in a flexible way, both on CPU and on GPU. In [12] the performances of the AINV preconditioner applied to a CG solver are compared with ILU. It has been shown that ILU outperforms AINV when both the preconditioning and the solution phases are both carried on the CPU. However AINV becomes very effective if the solution phase is carried out on GPUs, leaving the preconditioner building phase on the CPU. In this work, we will test the efficiency of the AINV method as a smoother in the framework of a multilevel preconditioner.

II. Sparse linear algebra on GPUs

Graphic Processing Units (GPUs) are now part of most high-performance computing systems due to their high computation power. In contrast to CPU architectures which are tailored for minimal latency, GPUs are manycore processors offering very high peak computational throughput. This means that, in order to realize this potential, programmers need to expose fine-grain parallelism. While dense linear algebra operations naturally lead to regular memory access patterns, performing sparse operations provides additional challanges. We can have, for example,some sparse matrices with regular pattern and others with large imbalances in the distribution of the nonzeros per row. Despite the irregularity of possible patterns, in order to take fully advantage of the GPU potential, it is essential to map them into the fine-grained parallel architecture employed by GPU [13]. In the following we will focus on Sparse Matrix Vector (SpMV) multiplication. This is an operation of tremendous importance when solving a sparse linear system of equations or in eigenvalue problems. SpMV performance largely depends on the memory bandwidth and since the memory bandwidth of GPUs exceeds that of the CPU, GPUs are particularly attractive for these operations. Great effort has been put in the optimization of sparse matrix vector multiplication for GPUs, see for example [14], [15] and [16] In this case the storage format for the sparse matrix involved in the multiplication can affect both the bandwidth utilization and the memory access pattern, resulting in a choice of significant importance.

A. Data formats for Sparse Matrices on GPUs

Fortran (as most programming languages) does not have a built-in representation for sparse matrices. Different storage schemes can be more suitable for different hardware including the possibility of exploiting special hardware instructions and the suitability of the format to be decomposed into independent, load-balancesd work unit. Even when the underlying

architecture is kept fixed, different storage formats can be more suitable to certain operations rather than others. This usually comes with a trade-off between memory requirements (overhead given to the explicit storage of indices, padding with explicit zeros,) and preprocessing computations. For this reason it is convenient to have a framework flexible enough to allow switching among different storage schemes [17]. Many storage schemes have been invented; Most popular schemes are [9]:

1) Coordinate (COO) - It consists of three arrayas. 1) An array containing the non-zero entries of the matrix 2) An integer array containing their row indices 3) An integer array containing their column indices.
2) Compressed Sparse Row (CSR)- Assuming that values are listed by row-order, In CSR format the array containing column indices is replaced with an array which points to the beginning of the row.
3) Compressed Sparse Column (CSC)- Assuming that values are listed by column-order, In CSR format the array containing row indices is replaces with an array which points to the beginning of the column.

ELLPACK and JAD formats were developed specially for vector machines. Great effort has been put in the last years in exploiting new and optimized data storage formats for sparse matrices on GPUs.These includes novel sparse matrix formats or GPU-specific optimizations of classic ones as well as many specific representations designed for particular sparsity patterns. A plugin to use computing capability of GPUs has been added to the PSBLAS library. This includes classic COO, CSR, CSC as well as ELLpack and the ELG storage formats. The latter is derived from the ELL storage specifically addressing sparse computations on GPUs. Object oriented techniques have been employed in order to allow dinamic choice of the representation format for the sparse matrix .

III. NUMERICAL RESULTS

A. PSBLAS and MLD2P4

The PSBLAS [18] is a library for Parallel Sparse Basic Linear Algebra Subroutines and it aims at facilitating the parallelization of computationally intensive scientific application requiring the solution of a sparse linear system of equations. It is written in Fortran 2003 following the object oriented programming paradigm, communication is based on MPI and it is tailored for distributed memory architectures. It implements basic dense and sparse matrix operations, such as multiplication of sparse by dense matrix, solution of block diagonal systems with triangular diagonal entries, preprocessing of sparse matrices. It also provides a great variety of iterative solvers: the Conjugated Gradient (CG) and its flexible variant (FCG), BiConjugated gradient and its stabilized variants, GMRES and the Generalized Conjugate Residual method (GCR). It also provides basic preconditioners such as diagonal scaling and block-Jacobi (BJAC) preconditioners. A plugin to use computing capability of GPUs has been added to the PSBLAS library. This includes new storage formats as well as techinques for dynamically choosing the representation of the sparse matrix. In PSBLAS classic COO, CSR, CSC and ELLpack storage formats are provided. The ELG storage format derived from the ELL storage format was specifically added for the GPUs. Additionally, it provides interfaces to the CSR and HYB storage formats in the NVIDIA cuSPARSE libary.

The Multilevel Domain Decomposition Parallel Preconditioners Package (MLD2P4) [**?**] is a framework for parallel algebraic multigrid and domain decomposition preconditioners to be used in the iterative solvers provided in PSBLAS. It extended from its original implementation and now a multilevel preconditioner is obtained as a combination of multigrid cycles with smoothers and coarsest-level solvers. V-, W-, and K- cycles [19] are available as well as Jacobi, block Jacobi, hybrid forward backward Gauss-Seidel and Schwarz methods available as smoothers (as well as stand-alone preconditioners). Approximate or exact solvers can be used at the coarsest level and the smoothed aggregation is based on a decoupled approach.

Approximate inverse plug-in is available in MLD2P4 and although not tied to the GPU usage, it can be combined with the GPU plugin in PSBLAS. This allows us to use approximate inverse in a flexible way, both on CPU and on GPU. In this way it is possible to test the efficiency of the AINV method as a smoother in the framework of a multilevel preconditioner. This is possible because of the structure of the MLD2P4 library, in which each multilevel preconditioner can be seen as a hierarchy of one-level preconditioners, each consisting of a smoother (applied at a given level) using a solver (applied on a given subdomain). AINV methods have been implemented as new variants of solver object in such a way that they can be easily plugged into the general multilevel framework.

IV. CONCLUSIONS

REFERENCES

[1] Salvatore Filippone and Alfredo Buttari. Object-oriented techniques for sparse matrix computations in fortran 2003. *ACM Transactions on Mathematical Software (TOMS)*, 38(4):23, 2012.

[2] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994.

[3] Ilse CF Ipsen and Carl D Meyer. The idea behind krylov methods. *American Mathematical Monthly*, pages 889–899, 1998.

[4] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[5] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[6] Henk A Van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing*, 13(2):631–644, 1992.

[7] Achi Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of computation*, 31(138):333–390, 1977.

[8] James H Bramble, Joseph E Pasciak, and Jinchao Xu. Parallel multilevel preconditioners. *Mathematics of Computation*, 55(191):1–22, 1990.

[9] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[10] Yvan Notay. An aggregation-based algebraic multigrid method. *Electronic transactions on numerical analysis*, 37(6):123–146, 2010.

[11] Maxim Naumov. Incomplete-lu and cholesky preconditioned iterative methods using cusparse and cublas. *Nvidia white paper*, 2011.

[12] Daniele Bertaccini and Salvatore Filippone. Approximate inverse preconditioners for krylov methods on heterogeneous parallel computers. *Adv. Parallel Comput*, 25:183–192, 2014.

[13] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[14] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*, page 18. ACM, 2009.

[15] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. *HiPEAC*, 5952:111–125, 2010.

[16] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo. Sparse matrix-vector multiplication on gpgpus. *ACM Transactions on Mathematical Software (TOMS)*, 43(4):30, 2017.

[17] Valeria Cardellini, Salvatore Filippone, and Damian WI Rouson. Design patterns for sparse-matrix computations on hybrid cpu/gpu platforms. *Scientific Programming*, 22(1):1–19, 2014.

[18] S. Filippone and M. Colajanni. PSBLAS: a library for parallel linear algebra computations on sparse matrices. 26(4):527–550, December 2000.

[19] William L Briggs, Van Emden Henson, and Steve F McCormick. *A multigrid tutorial*. SIAM, 2000.