A hybrid CPU/GPU approach for the parallel algebraic recursive multilevel solver pARMS

Aygul Jamal
Université Paris-Sud
91405, Orsay, France
Email: aygul.jamal@lri.fr

Marc Baboulin
Université Paris-Sud
91405, Orsay, France
Email: baboulin@lri.fr

Amal Khabou
Université Paris-Sud
91405, Orsay, France
Email: amal.khabou@lri.fr

Masha Sosonkina
Old Dominion University
Norfolk, VA, 23529, United States
Email: msosonki@odu.edu

***Abstract*—We illustrate how the distributed parallel Algebraic Recursive Multilevel Solver based on MPI can be adapted for heterogeneous CPU/GPU architectures. The tasks performed on the GPU are related to the preconditioning of each part of the distributed matrix (local preconditioning) which is handled in the distributed version by each MPI process. The solving step remains on the CPU. In our implementation, the local preconditioning can be based either on the randomization of the last Schur complement system in the multilevel recursive process, or on an Incomplete LU factorization from the MAGMA library. Numerical experiments show that a promising performance improvement can be obtained using either randomized multilevel recursive preconditioning or Incomplete LU preconditioning for large enough matrices. Each preconditioning method ensures a good performance for a given set of matrices.**

I. Introduction

In this work, we are concerned with the use of GPU computing for preconditioned Krylov subspace methods and more specifically for the pARMS solver package [1]. We focus on the preconditioning phase of this solver, which represents the main computational part and study whether it can take advantage of GPU capabilities by using efficient kernels based on randomization or incomplete LU factorization. The main goal is to accelerate the preconditioning step, and as a consequence to improve the performance of the pARMS solver.

The Algebraic Recursive Multilevel Solver (ARMS) is one of the solvers that applies iterative Krylov subspace methods to solve sparse linear systems. It relies on multilevel partial elimination. The preconditioning separates the entries into two parts, the first part, called fine set, is composed of block independent sets, and the second part, called coarse set, contains the rest of the entries. The coarse set can be used to build the Schur complement, which allows us to perform a block LU factorization. The inter-level LU factorization can be built from the upper level LU factorization and the fine set, up to the first level.

Parallel ARMS (pARMS) is a distributed-memory implementation of ARMS, which relies on distributed independent sets. It provides a set of standard preconditioners [2] such as additive Schwartz, Schur complement and block Jacobi. However, to our knowledge, there is no version of pARMS that exploits the possibility of using accelerators such as GPUs.

In recent years, several packages which include GPU implementations have been developed for iterative methods to solve sparse linear systems. As examples, we can mention the **CUsparse** library [3], which is a collection of routines for sparse linear algebra computations on NVIDIA GPUs, or **ViennaCL** [4], which is a free open-source linear algebra library written in C++. For the GPU kernels integrated into our implementation, we will use the **MAGMA** library [5], [6], which is a public domain linear algebra library for heterogeneous architectures. In addition to being well-known for the dense linear algebra (e.g., factorizations and solvers for linear systems, least squares and eigen problems), MAGMA also provides a large variety of solvers, preconditioners, and eigensolvers for sparse linear systems. Comprehensive support for NVIDIA GPUs is provided, as well as some basic routines and functionalities in OpenCL and for Intel's Xeon Phi manycore accelerators (MIC). We will use MAGMA routines for integrating preconditioning based either on ARMS and Random Butterfly Transformations (RBT) or Incomplete LU factorization into pARMS. We analyze the performance improvement obtained using each method on some test matrices.

The rest of the paper is organized as follows. In Section 2, we describe the general features of the pARMS solver. Section 3 presents the new GPU functions that we have integrated into the pARMS solver. In Section 4, we report some experimental results on a set of test matrices to compare the CPU and the CPU/GPU versions of the code. Section 5 gives concluding remarks and future work.

II. Preconditioned Krylov Methods and the pARMS Solver

A. Preconditioned Krylov Methods

A preconditioned Krylov subspace method is used to solve the linear system $Ax = b$, where A is square non-symmetric matrix, in general. If M is a preconditioning matrix, then the right-preconditioned system may be expressed as:

$$AM^{-1}y = b, \text{ where } y = Mx \; , \quad (1)$$

which is solved instead of the original system $Ax = b$. To solve this system by using iterative methods, first, we compute the residual $r_0 = b - Ax_0$ [7] after initializing x_0, then we may use a right-preconditioned Krylov subspace method to find an approximate solution from the affine subspace [8]:

$$x_m = x_0 + \text{span}\{r_0, \; AM^{-1}r_0, \ldots, (AM^{-1})^{m-1}r_0\} \; , \quad (2)$$

which satisfies certain conditions. For instance, the GMRES algorithm [2] requires that the residual $r_m = b - Ax_m$ has a minimal 2-norm. The flexible GMRES, abbreviated as FGMRES [2], differs from GMRES by allowing the preconditioning to change at each iteration.

One way to obtain the preconditioning matrix M is to use an incomplete LU factorization. This ILU factorization is constructed by performing an approximate Gaussian Elimination (GE) [9] on a sparse matrix A and dropping certain nonzero entries of the factorization according to different dropping strategies. A dropping strategy that relies on levels of the matrix fill-in results in a factorization called $ILU(K)$. For example, $ILU(0)$ is obtained by performing the LU factorization of A and dropping all fill-in elements generated during the elimination process. Conversely, if the nonzeros are dropped according to their numerical value magnitudes, then the resulting factorization is called ILU with the threshold or, if combined with the dropping strategy based on the number of remaining nonzero, with dual threshold ($ILUT$) and is performed as follows. In the algorithm $ILUT(k, \tau)$, there are two important rules. (1) If an element is less than relative tolerance τ_i ($\tau \times$ the norm of the ith row), it is dropped. (2) Keep only the k largest elements in the L and U parts of the row along with the diagonal element.

In this work, we use a preconditioner called Algebraic Recursive Multilevel Solver (ARMS) [10], which is based on a block incomplete LU factorization with different dropping strategies. First, the matrix is permuted in order to obtain a 2×2 block structure under the form

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \times \begin{pmatrix} u \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix}, \tag{3}$$

where the submatrix B comes from a group-independent set reordering (see, e.g., [2]), thereby generating a block-diagonal matrix B [11]. Then, it is convenient to eliminate the u block of variables to obtain a system with only y variables. The coefficient matrix for the resulting "reduced system" is the Schur complement $S = C - EB^{-1}F$ [12]. A recursion can now be exploited, such that dropping is applied to S to limit the fill-in followed by the reordering of the resulting reduced system into the form (3) using the group-independent set discovery again. This process is repeated for several levels of recursion until the Schur-complement system is small enough or until a maximum number of recursion levels is reached. Then, the last Schur complement may be solved by a direct or an iterative solver. Note that the sparsification of the Schur complement may be undertaken at each level of recursion, to keep down the preconditioning costs.

B. Parallel Implementation of ARMS

Algorithm 1 oulines the steps to go through to solve the linear system $Ax = b$, using the Schur Complement (SC) preconditioning in the pARMS package. First, each processor loads the input matrix and performs Distributed Site Expansion (DSE) partitioning [13], which attributes to each process a set of equations corresponding to the rows of the global linear system, as well as the associated unknown variables. We note that when the number of processors is not a power of 2, the load balance between the different MPI processes is not ensured. As depicted in Figure 1, in the rows assigned to each processor, two parts may be distinguished: a local submatrix A_i that acts only on the local variables (u_i) and an external interface matrix X_i that acts only on the external interface variables, which are communicated from neighboring processors at each matrix-vector multiplication.

To construct the global SC preconditioning, first the local LU factorization is performed on the local matrix held by each processor (Step 4 in Algorithm 1) in order to precondition the internal part of the local system and to obtain a factorization of the local Schur complement matrices $S_i = C_i - E_i B_i^{-1} F_i$, where $A_i = \begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix}$. More details about this step can be found in [14]. Then the different parts of the distributed SC are constructed and *left to reside* in each processor (Step 5). In essence, the global SC system is never assembled or gathered in one processor and is being solved using a preconditioned GMRES method. Its distributed implementation, employed to solve the global SC, may be viewed as an "inner" accelerator with respect to the "outer" accelerator FGMRES used to solve the original linear system with the input matrix A. The solution of the global SC system yields a different preconditioning at each iteration, and hence the need for the outer flexible GMRES.

Algorithm 1 The linear system solution using a global Schur Complement preconditioning in pARMS

```
1. Each processor loads the sparse matrix $A$.
2. Partition the input matrix $A$ using DSE partitioner.
3. Each processor exchanges boundary variables with neighboring processors.
4. Each processor performs local LU factorization on its local submatrix $A_i = L_i \times U_i$ (see Fig. 1).
5. Each processor constructs its portion $S_i$ of the global Schur complement system from the result of the local LU factorization and the external interface submatrix $X_i$.
6. Solve the global Schur complement system iteratively by distributed GMRES as "inner" solver.
7. Each processor back-substitutes its interface variables to recover the internal variables.
8. Each processor calculates its local residual.
9. If the global residual norm is not small enough, repeat from Step 6.
```

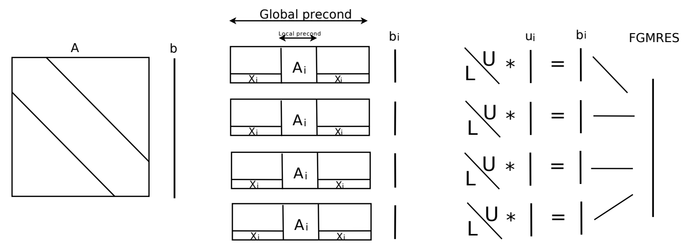


Figure 1: Sketch of the distributed linear system solution using pARMS (example using four processors).

III. Integration of GPU kernels into pARMS

In our hybrid CPU/GPU approach, we use GPU computing in the preconditioning step of the pARMS solver. We mainly focus on the two following functions. The first function is related to the local preconditioner ARMS where we notice that the last Schur complement system becomes denser compared to the previous levels, and thus needs more time to be solved. Similarly to [15], we propose the use of Random Butterfly Transformation to avoid pivoting when solving the last Schur complement system.

The second function is related to the local preconditioner $ILU(0)$ which represents, by profiling the pARMS solver execution time, a significant part of the global time for a large set of test matrices. For instance, we analyze the time breakdown for solving the test problem `flame2p3d80x4` (described in Section IV-A) using one MPI process, where we use block Jacobi as a global preconditioner, ILU0 as a local preconditioner, and FGMRES to solve the global preconditioned system. We observe in Figure 2 that, for this test matrix, the preconditioning application (see Steps 6 and 7 of Algorithm 1) represents about 27% of the total time needed to solve the problem. We note that, for other test matrices, the preconditioning represents between 20% and 50% of the time for solution. This observation motivated our interest in accelerating the preconditioning phase using GPU computing.

A. *GPU Implementation of Random Butterfly Transformations in ARMS preconditioning*

In matrix factorizations, pivoting is a classical technique to avoid division by zero or too-small entries. Random Butterfly Transformation (RBT) can be used as an alternative to pivoting [16] and is particularly efficient for heterogeneous architectures [17] since it decreases the amount of data movement comparing to pivoting. The RBT approach consists in transforming the initial matrix into a matrix that would be sufficiently random to avoid pivoting, then using a Gaussian elimination with no pivoting to solve the randomized linear system. In [15], we explained how the RBT technique can be applied to the ARMS preconditioning where the original $ILUT$ factorization is replaced by the RBT preprocessing in the solution of the last Schur complement. First, the last Schur complement which is a sparse matrix is converted into

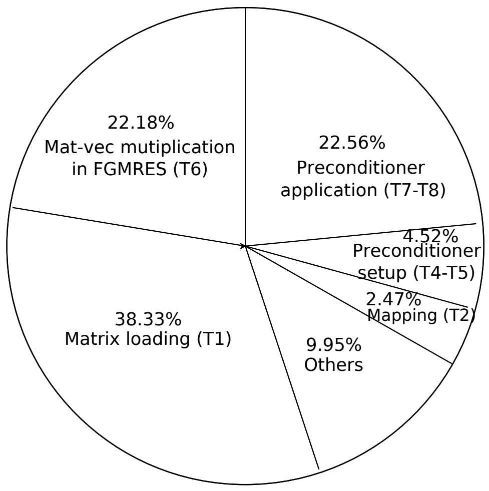


Figure 2: Time breakdown for the pARMS solver (matrix `flame2p3d80x4`) and corresponding task numbers in Algorithm 1.

a dense matrix. Then, RBT is applied and the dense matrix is randomized using two recursive butterfly matrices. Finally, the randomized dense matrix is factorized using a LAPACK-like [18] routine that performs Gaussian elimination with no pivoting, followed by two triangular solves. Note that RBT requires the size of the matrix to be a power of 2, which can be obtained by augmenting the initial matrix with additional ones on the diagonal. In the following, we describe how RBT is applied to the last Schur complement S.

We first recall that a butterfly matrix is a random $n \times n$ matrix of the form

$$B^{<n>} = \frac{1}{\sqrt{2}} \begin{bmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{bmatrix},$$

where R_0 and R_1 are random diagonal $\frac{n}{2} \times \frac{n}{2}$ matrices. A *recursive butterfly matrix* of size n and depth d is defined recursively as follows

$$W^{<n,d>} = \begin{bmatrix} B_1^{<n/2^{d-1}>} & & \\ & \ddots & \\ & & B_{2^{d-1}}^{<n/2^{d-1}>} \end{bmatrix} \cdot W^{<n,d-1>},$$

where $W^{<n,1>} = B^{<n>}$, the $B_i^{<n/2^{d-1}>}$ are butterflies of size $n/2^{d-1}$, and $B^{<n>}$ is a butterfly of size n.

To apply RBT to the matrix S, we generate two recursive butterfly matrices W_1 and W_2 of size $n \times n$. We set d to 2 since, as explained in [16], two recursions are in general sufficient to obtain satisfactory results. Instead of solving the initial system, $S\ x = b$, using Gaussian elimination with partial pivoting, we first solve the randomized system, $W_1{}^T\ S\ W_2\ z = W_1{}^T\ b$, using Gaussian elimination with no pivoting, then the system $W_2\ z = x$.

In Figure 3a, we describe the data movements (numbered chronologically) between the CPU and the GPU in our implementation of the solution of the last Schur complement using

RBT (S denotes the last Schur complement, S' is a dense format of S, and S'_r is the randomized matrix, rhs is the right-hand side of the system).

We note that to get performance improvement using the RBT GPU implementation, the last Schur complement should be large enough, as it will be illustrated in the numerical experiments section.

B. *GPU implementation for $ILU(0)$ preconditioning in pARMS*

In this implementation, we use MAGMA routines to perform the $ILU(0)$ factorization and the triangular solves. The routine `magma_dcumilusetup` prepares the ILU preconditioner via the CUsparse library to factorize the local system of each processor. Then the routines `magma_dapplycumilu_l` and `magma_dapplycumilu_r` perform the left and right triangular solves, respectively, by using the $ILU(0)$ preconditioner.

In Figure 3b, we describe the data movements between the CPU and the GPU for the $ILU(0)$ preconditioning in pARMS, where A denotes the local matrix held by a processor. In this figure, the data movements are numbered chronologically. Note that the factorized LU system is sent back to the CPU host to perform the separation between the L and U factors (steps 2 and 3).

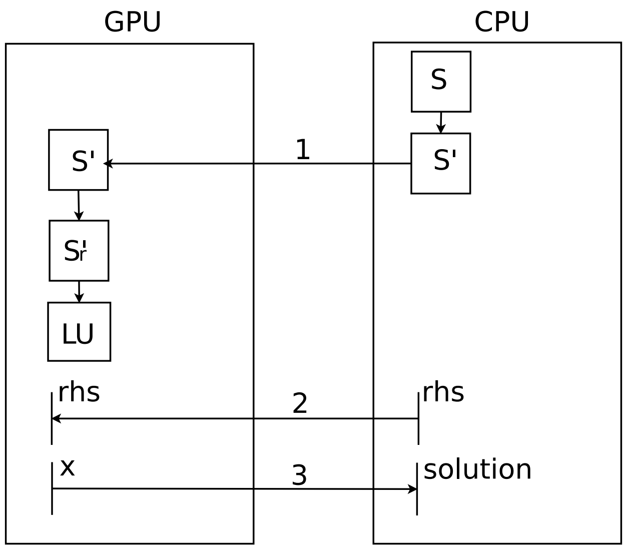
IV. Numerical experiments

A. *Experimental framework*

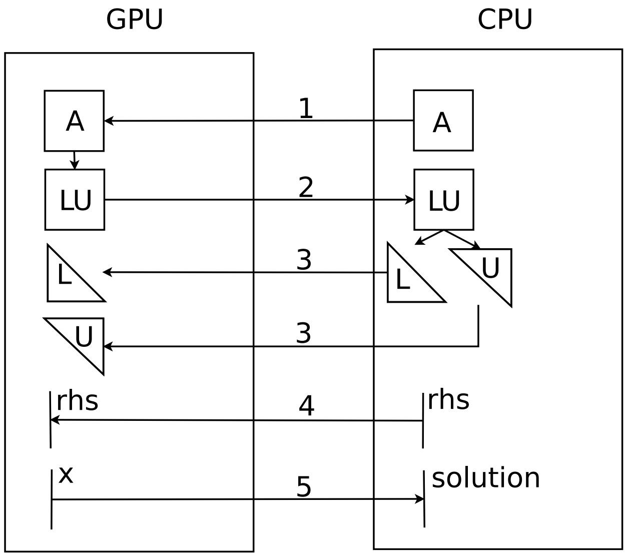
The experiments were carried out in double precision arithmetic on a system composed of one NVIDIA Kepler K40m GPUs and a dual Intel Xeon E5-2620 system. The GPU implementations are based on CUDA version 7.5 [19] and MAGMA version 2.0 [20]. We use one MPI process per core and no multi-threading. Thus, in our experiments, all the MPI processes are sharing the same GPU.

The first test matrix `edf` comes from a computational fluid dynamic (CFD) application [21] and is generated using a regular hexahedral mesh. This matrix is symmetric and of size $16,384$. The second test matrix `flame2p3d80` arizes from using a finite-difference scheme with local approximation (called FLAME) [22] to screened electrostatic interactions of spherical colloidal particles governed by the Poisson-Boltzmann equation (PBE). Specifically, this matrix corresponds to a two-particle simulation on an 80^3 grid and has a regular sparsity structure but it is not symmetric and not diagonally-dominant due to the characteristics of FLAME. This simulation has been studied in [23], where it has been shown that parallel preconditioning is imperative for its solution and that even modest levels of ILU fill-in already yield a good convergence. The third test matrix `epb3` is a matrix from the University of Florida Sparse Matrix Collection [24]. It represents a large case of a plate-fin heat exchanger.

Here, we test these matrices on hybrid CPU/GPU architectures. As detailed in the previous section, we use GPU computing to perform the local preconditioning. We point out



(a) $arms\_rbt$



(b) $magma\_ilu0$

Figure 3: CPU-GPU communication for $arms\_rbt$ and $magma\_ilu0$ in pARMS.

that for our experiments, we increase the size of the original matrices `edf`, `flame2p3d80`, and `epb3` in order to study the scalability. In particular, to preserve the properties of the original matrices, we increase the size by simply duplicating the original matrix several times on the diagonal. For example, to obtain the matrix `flame2p3d80x4`, which is four times larger, we duplicate the original `flame2p3d80` three times along the diagonal. Table I contains the size and number of nonzeros of the "scaled" test matrices.

Table I: The set of test matrices.

Matrix ╲ Characteristics	Dimension	# non-zeros	Structure
`edfx128`	2,097,152	14,155,776	symmetric
`flame2p3d80x4`	2,125,764	13,808,916	unsymmetric
`epb3x64`	5,415,488	29,672,000	unsymmetric

B. *RBT combined with ARMS preconditioning*

In this section, we compare the execution time of the original pARMS solver (CPU) to that of the CPU/GPU version which uses RBT to solve the last Schur complement system in the recursive process, as described in Section III-A.

In Figure 4, we evaluate the weak scaling of the CPU and CPU/GPU solvers. The tests are performed on the following matrices: `edfx16`, `edfx32`, `edfx64` and `edfx128`. Here the number of non-zeros of the sparse matrix increases with the number of cores, that is with the number of MPI processes used. The performance of `arms_rbt` is better than `arms`, except for 2 cores due to the small size of the problem that does not enable us to take advantage of the GPU. Figure 4 shows that using 12 MPI processes and one GPU, the execution time decreases by 30%. We add that for this specific test, the size of the last Schur complement held by each MPI process ranges from 2828 to 3724.

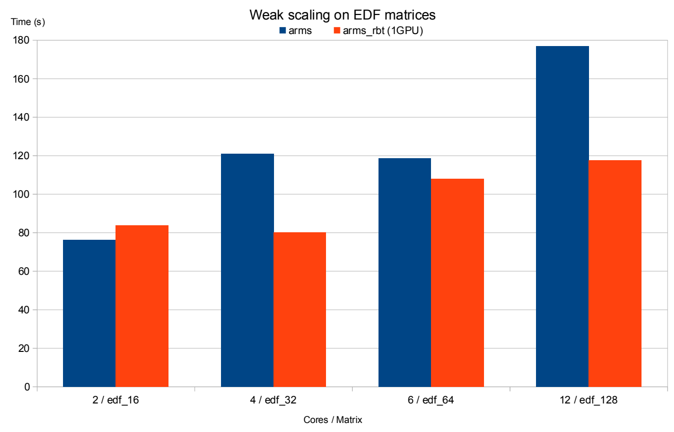


Figure 4: Execution time for `arms` and `arms_rbt` on different `edf` matrices.

Figure 5 shows that `arms_rbt` performs better than `arms` on the `epb3x64` matrix using different number of MPI processes varying from 6 to 12. In the best case, using `arms_rbt` decreases the execution time by 25%. We note that we can not use less than 6 MPI processes for this problem since the last Schur complement is too large to fit into the GPU memory.

C. *$ILU(0)$ preconditioning*

We compare the execution time for the two following solvers:

- pARMS with $ILU(0)$ local preconditioning on the CPU, referred to as `pARMS_ilu0`.
- pARMS with $ILU(0)$ local preconditioning on the GPU, referred to as `magma_ilu0`.

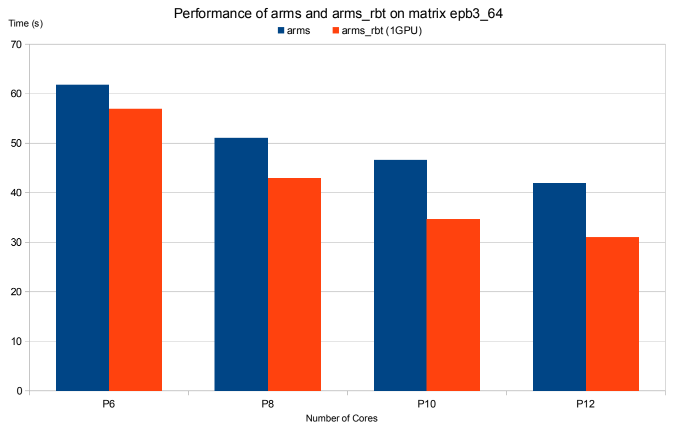


Figure 5: Execution time for `arms` and `arms_rbt` on the `epb3x64` matrix.

Figure 6 displays the execution time of the `pARMS_ilu0` and the `magma_ilu0` solvers for the `edfx128` matrix. We note that for this matrix, using more than 6 MPI processes does not bring any advantage. Indeed the processes are sharing the same GPU, which increases the communication amount to perform between the CPU and the GPU with respect to the problem size.

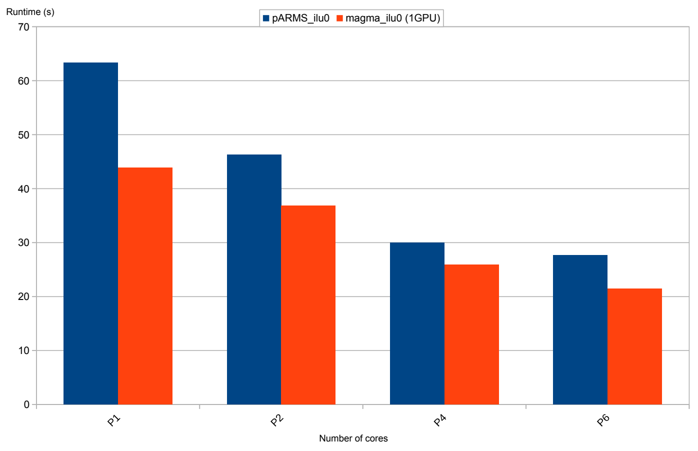


Figure 6: The execution time for pARMS with $ILU(0)$ on the `edfx128` matrix.

Figure 7 shows the execution time of the `pARMS_ilu0` and the `magma_ilu0` solvers for the `flame2p3d80x4` matrix. The best improvement is obtained using one MPI process and it is about 30%. In this figure, we observe that when we use 6 MPI processes, the execution time of the `pARMS_ilu0` solver increases with respect to the use of 4 MPI processes. This is because of the DSE partitioning that does not ensure good load balance between the processes when using a number of processes which is not a power of 2.

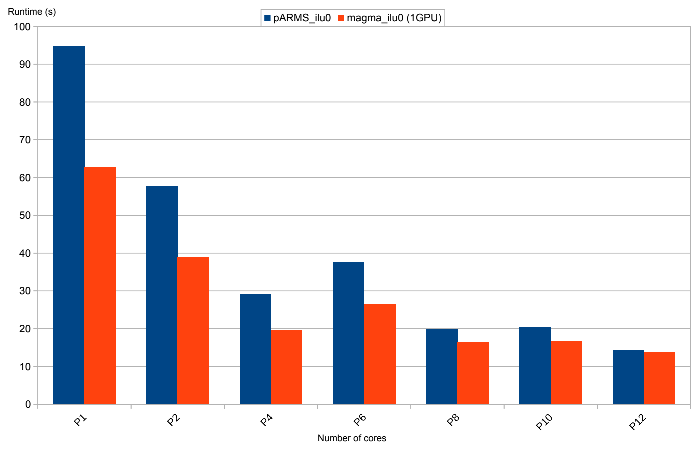


Figure 7: The execution time for pARMS with $ILU(0)$ on the `flame2p3d80x4` matrix.

V. Conclusion

We have illustrated how a non intrusive approach can be applied to integrate GPU computing into pARMS, more specifically for the local preconditioning phase that represents a significant part of the time to solve a given sparse linear system. The CPU-only and the hybrid CPU/GPU solvers have been compared on several test problems from physical applications. The performance results of the hybrid CPU/GPU solver using the ARMS preconditioning combined with RBT, or the $ILU(0)$ preconditioning, show a performance gain up to 25% and 30%, respectively, on the test problems considered in this paper. In a future work, extensive testing will be performed on other matrices and other local preconditioners (e.g., $ILUT$), also by using several nodes of a cluster of GPUs.

VI. Acknowledgements

The work of Sosonkina was supported in part by the Air Force Office of Scientific Research under the AFOSR award FA9550-12-1-0476, by the National Science Foundation grants 0904782, 1047772, 1516096, by the U.S. Department of Energy, Office of Advanced Scientific Computing Research, through the Ames Laboratory, operated by Iowa State University under contract No. DE-AC02-07CH11358, and by the U.S. Department of Defense High Performance Computing Modernization Program, through a HASI grant.

References

[1] Z. Li, Y. Saad, and M. Sosonkina, "pARMS: a parallel version of the algebraic recursive multilevel solver," *Numerical Linear Algebra with Applications*, vol. 10, no. 5-6, pp. 485–509, 2003.

[2] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[3] *CUsparse Toolkit Documentation v7.5*, NVIDIA Corporation, September 2015.

[4] K. Rupp, F. Rudolf, and J. Weinbub, "ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs," in *Intl. Workshop on GPUs and Scientific Applications*, 2010, pp. 51–56.

[5] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5&6, pp. 232–240, 2010.

[6] M. Baboulin, J. Dongarra, J. Demmel, S. Tomov, and V. Volkov, "Enhancing the performance of dense linear algebra solvers on gpus in the magma project," Poster at Supercomputing (SC'08), Austin USA, November 15, 2008, http://www.lri.fr/ baboulin/SC08.pdf.

[7] M. Arioli, J. Demmel, and I. Duff, "Solving sparse linear systems with sparse backward error," *SIAM Journal on Matrix Analysis and Applications*, vol. 10, no. 2, pp. 165–190, 1989.

[8] B. N. Bond and L. Daniel, "Guaranteed stable projection-based model reduction for indefinite and unstable linear systems," in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 728–735.

[9] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki, "A Survey of Recent Developments in Parallel Implementations of Gaussian Elimination," *Concurrency and Computation: Practice and Experience*, p. 18, May 2014.

[10] Y. Saad and B. Suchomel, "ARMS: an algebraic recursive multilevel solver for general sparse linear systems," *Numerical Linear Algebra with Applications*, vol. 9, no. 5, pp. 359–378, 2002.

[11] Y. Bai, W. N. Gansterer, and R. C. Ward, "Block tridiagonalization of "effectively" sparse symmetric matrices," *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 326–352, Sep. 2004.

[12] Z.-H. Cao, "Constraint Schur complement preconditioners for nonsymmetric saddle point problems," *Appl. Numer. Math.*, vol. 59, no. 1, pp. 151–169, Jan. 2009.

[13] Y. Saad and M. Sosonkina, "Non-standard parallel solution strategies for distributed sparse linear systems," in *Parallel Computation: 4th International ACPC Conference*, ser. Lecture Notes in Computer Science, P. Z. *et al.*, Ed., vol. 1557. Springer-Verlag, 1999, pp. 13–27.

[14] ——, "Distributed Schur Complement techniques for general sparse linear systems," *SIAM J. Scientific Computing*, vol. 21, pp. 1337–1356, 1999.

[15] M. Baboulin, A. Jamal, and M. Sosonkina, "Using random butterfly transformations in parallel Schur complement-based preconditioning," in *2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, Łódz, Poland, September 13-16, 2015*, 2015, pp. 649–654.

[16] M. Baboulin, J. Dongarra, J. Herrmann, and S. Tomov, "Accelerating linear system solutions using randomization techniques," *ACM Trans. Math. Softw.*, vol. 39, no. 2, pp. 8:1–8:13, Feb. 2013.

[17] M. Baboulin, A. Khabou, and A. Rémy, "A randomized lu-based solver using GPU and Intel Xeon Phi accelerators," in *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*, 2015, pp. 175–184.

[18] E. Andersen, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchev, and D. Sorensen, "LAPACK users' guide," 3rd ed., SIAM Philadelphia, 1999.

[19] *CUDA Toolkit Documentation v7.5*, NVIDIA Corporation, September 2015.

[20] "Magma web page http://icl.cs.utk.edu/magma/index.html."

[21] H. Anzt, M. Baboulin, J. Dongarra, Y. Fournier, F. Hulsemann, A. Khabou, and Y. Wang, "Accelerating the conjugate gradient algorithm with GPU in CFD simulations," 2016, to appear in the proceedings of VECPAR 2016.

[22] I. Tsukerman, "A class of difference schemes with flexible local approximation," *J. Comput. Phys.*, vol. 211, no. 2, pp. 659–699, 2006.

[23] M. Sosonkina and I. Tsukerman, "Parallel solvers for flexible approximation schemes in multiparticle simulation," in *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, Proceedings, Part I*, ser. Lecture Notes in Computer Science, V. Alexandrov, G. van Albada, P. Sloot, and J. Dongarra, Eds., vol. 3991. Springer, 2006, pp. 54–62.

[24] "The University of Florida sparse matrix collection http://www.cise.ufl.edu/research/sparse/matrices/."