Extending ILUPACK with a GPU version of the BiCGStab method

José Aliaga, Enrique S. Quintana-Ortí
Dept. de Ingeniería y
Ciencia de los Computadores
Universidad Jaime I
Castellón, Spain
{aliaga,quintana}@icc.uji.es

Ernesto Dufrechou, Pablo Ezzatti
Instituto de Computación (INCO)
Universidad de la República
Montevideo, Uruguay
{edufrechou,pezzatti}@fing.edu.uy

***Abstract*—The solution of sparse linear systems of large dimension is an important stage in problems that span a diverse range of applications. For this reason, a number of software packages have been developed, among which ILUPACK stands out due to its inverse-based multilevel ILU preconditioner with appealing numerical properties. In this work we extend the iterative methods available in ILUPACK. Concretely, we develop a data-parallel implementation of the BiCGStab method for GPUs that expands the set of ILUPACK-preconditioned solvers for general linear systems. The experimental evaluation, carried out in a hardware platform equipped with a high-end multicore CPU and a Nvidia GPU, shows that our novel proposal reaches speedups values between 5 and 10× when it is compared with the CPU counterpart and values of up to 8.2× runtime reduction over other GPU solvers.**

***Index Terms*—Linear Systems, preconditioning technique, massively parallel processing**

I. Introduction

The solution of large-scale sparse linear systems is a challenging task appearing in many engineering and scientific applications. Examples of this particular type of problem appear, among others, in the discretization of partial differential equations (PDEs), quantum physics and circuit simulation [1]. In many of these cases, the linear system itself is the most computationally-demanding stage, requiring a fast and accurate numerical solver when the coefficient matrix of the system presents a large dimension [2].

A common numerical approach to tackle large and sparse linear systems employs Krylov subspace-based methods, in conjunction with some sort of preconditioning. Approximate matrix factorizations stand out among a wide variety of preconditioners, due to the acceleration they provide on the convergence rate of iterative solvers, specially for problems derived from discretized elliptical PDEs [1]. Although this class of preconditioners can be also applied to other problems, standard ILUs (incomplete LU) are likely to face difficulties for highly indefinite or ill-conditioned matrices. For this reason, much effort has been dedicated over the years in order to improve their robustness and numerical stability. A relevant example is ILUPACK,[1] a package for the solution of sparse linear systems via Krylov subspace methods that relies on an inverse-based multilevel ILU preconditioning technique for general as well as Hermitian positive definite/indefinite linear systems [3].

Although interesting from a mathematical perspective, the computation of ILUPACK's preconditioner and its application in the context of an iterative Krylov subspace solver are expensive procedures, especially for sparse linear systems of large dimension. Their large computational cost motivated the design of parallel variants of ILUPACK's CG method [1], for symmetric positive definite (s.p.d.) systems, on shared-memory and message-passing platforms [4], [5], [6]. In order to expose task-parallelism, these implementations calculate a preconditioner which differs from that computed by the original (sequential) ILUPACK, offering distinct convergence rates (though not necessarily slower for the parallel versions). Moreover, the task-parallel variants usually require more floating-point arithmetic operations (flops) than the original ILUPACK, with the overhead cost rapidly growing with the degree of task-parallelism that is exposed [4].

In [6] we also introduced a version of ILUPACK's CG method, for s.p.d. systems that exploits the data-parallelism intrinsic to the most expensive kernels, off-loading their execution to a graphics processing unit (GPU). Compared with the task-parallel solvers, the data-parallel version preserves the computational cost, semantics, and convergence rate of the sequential ILUPACK. Additionally, in [14] we recently followed a similar approach to accelerate ILUPACK's solvers for general and symmetric indefinite linear systems on GPUs, providing data-parallel implementations of GMRES, BiCG and SQMR [1].

This work intends to expand the family of Krylov subspace iterative methods for sparse general linear systems included in ILUPACK by proposing a data-parallel version of the Bi-Conjugate Gradient Stabilized Method [8] (BiCGStab). To accomplish this, we first develop a CPU version of the solver to then produce a variant that runs entirely on the graphics accelerator. For this reason, in the new solver we depart from the reverse-communication scheme followed by our existing implementations of GMRES and BiCG. The experimental analysis compares the performance of the new

[1] Available at http://ilupack.tu-bs.de.

solver with our previous developments using a set of real problems extracted from the University of Florida (Sparse) Matrix Collection (UFMC) [9], and test problems of scalable size derived from the discretization of PDEs. This comparison shows an improvement of the speed-up values that ranges from 1.3 to 3.0×. On the other hand, when the CPU and GPU version of the BiCGStab method are compared, the differences between their runtime are in range of 5 and 10×.

The rest of the paper is structured as follows. In Section II we review the iterative solvers integrated into ILUPACK and the use of GPUs to accelerate them. This is followed by the description of our proposal in Section III, and the experimental evaluation performed in Section IV. Finally, a few remarks and some lines of future work close the paper in Section V.

II. Accelerated Solution of Sparse Linear Systems with ILUPACK

Consider the linear system $Ax = b$, where the $n \times n$ coefficient matrix A is large and sparse, and both the right-hand side vector b and the sought-after solution x contain n elements. ILUPACK provides software to calculate an inverse-based multilevel ILU preconditioner M, of dimension $n \times n$, which can be applied to accelerate the convergence of Krylov subspace-based iterative solvers. The package includes numerical methods for different matrix types, precisions, and arithmetic, covering Hermitian positive definite/indefinite and general real and complex matrices. When using an iterative solver enhanced with the ILUPACK preconditioner, the most demanding task from the computational point of view is the application of the preconditioner, which occurs (at least once) per iteration of the solver.

A. Computation of the Preconditioner

Let us focus on the real case, where $A, M \in \mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^n$. The computation of ILUPACK's preconditioner proceeds following three steps:

1) Initially, a preprocessing stage scales A by a diagonal matrix $\tilde{D} \in \mathbb{R}^{n\times n}$ and reorders the result by a permutation $\tilde{P} \in \mathbb{R}^{n\times n}$: $\hat{A} = \tilde{P}^T \tilde{D} A \tilde{D} \tilde{P}$.
2) An incomplete factorization next computes $\hat{A} \approx LDU$, where $L, U^T \in \mathbb{R}^{n\times n}$ are unit lower triangular factors and $D \in \mathbb{R}^{n\times n}$ is (block) diagonal. In some detail, $\hat{A}$ is processed in this stage to obtain the partial ILU factorization:
$$\hat{P}^T \hat{A} \hat{P} \equiv \begin{pmatrix} B & F \\ G & C \end{pmatrix} = LDU + E = \begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & S_c \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} + E. \quad (1)$$
Here, $\hat{P} \in \mathbb{R}^{n\times n}$ is a permutation matrix, $\|L^{-1}\|, \|U^{-1}\| \lessapprox \kappa$, with κ a user-predefined threshold, E contains the elements "dropped" during the ILU factorization, and S_C represents the approximate Schur complement assembled from the "rejected" rows and columns.
3) The process is then restarted with $A = S_c$, (until S_c is void or "dense enough" to be handled by a dense solver,) yielding a multilevel approach.

At level l, the multilevel preconditioner can be recursively expressed as
$$M_l \approx \tilde{D}^{-1} \tilde{P} \hat{P} \begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} \hat{P}^T \tilde{P}^T \tilde{D}^{-1}, \quad (2)$$
where L_B, D_B and U_B are blocks of the factors of the multilevel LDU preconditioner (with L_B, U_B^T unit lower triangular and D_B diagonal); and M_{l+1} stands for the preconditioner computed at level $l+1$.

A detailed explanation of each stage of the process can be found in [3].

B. Application of the Preconditioner during the Iterative Solve

For the review of this operation, we consider its application at level l, e.g. to compute $z := M_l^{-1} r$. This requires solving the system of linear equations:
$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} \hat{P}^T \tilde{P}^T \tilde{D}^{-1} z = \hat{P}^T \tilde{P}^T \tilde{D} r. \quad (3)$$

Breaking down (3), we first recognize two transformations to the residual vector, $\hat{r} := \hat{P}^T \tilde{P}^T (\tilde{D} r)$, before the following block system is defined:
$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} w = \hat{r}. \quad (4)$$

This is then solved for $w (= \hat{P}^T \tilde{P}^T \tilde{D}^{-1} z)$ in three steps,
$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} y = \hat{r}, \quad \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} x = y, \quad \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} w = x, \quad (5)$$
where the recursion is defined in the second one.

In turn, the expressions in (5) also need to be solved in two steps. Assuming y and $\hat{r}$ are split conformally with the factors, for the expression on the left of (5) we have
$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} y_B \\ y_C \end{pmatrix} = \begin{pmatrix} \hat{r}_B \\ \hat{r}_C \end{pmatrix} \Rightarrow \quad L_B y_B = \hat{r}_B \ , \ y_C := \hat{r}_C - L_G y_B. \quad (6)$$
Partitioning the vectors as earlier, the expression in the middle of (5) involves the diagonal-matrix multiplication and the effective recursion:
$$\begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} x_B \\ x_C \end{pmatrix} = \begin{pmatrix} y_B \\ y_C \end{pmatrix} \Rightarrow \quad x_B := D_B^{-1} y_B \ , \ x_C := M_{l+1}^{-1} y_C. \quad (7)$$
In the recursion base step, M_{l+1} is void and only x_B has to be computed. Finally, after an analogous partitioning, the expression on the right of (5) can be reformulated as
$$\begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} \begin{pmatrix} w_B \\ w_C \end{pmatrix} = \begin{pmatrix} x_B \\ x_C \end{pmatrix} \Rightarrow \quad w_C := x_C \ , \ U_B w_B = x_B - U_F w_C, \quad (8)$$

where z is simply obtained from $z := \tilde{D}(\tilde{P}(\hat{P}w))$.

To save memory, ILUPACK discards the off-diagonal blocks L_G and U_F once the level of the preconditioner is calculated, keeping only the rectangular matrices G and F, which are often much sparser. Thus, (6) is changed as:

$$\begin{aligned} L_G = GU_B^{-1}D_B^{-1} \quad \Rightarrow \\ y_C := \hat{r}_C - GU_B^{-1}D_B^{-1}y_B = \hat{r}_C - GU_B^{-1}D_B^{-1}L_B^{-1}\hat{r}_B, \end{aligned} \tag{9}$$

while the expressions related to (8) are modified as

$$\begin{aligned} U_F = D_B^{-1}L_B^{-1}F \quad \Rightarrow \\ U_B w_B = D_B^{-1}y_B - D_B^{-1}L_B^{-1}Fw_C. \end{aligned} \tag{10}$$

Operating with care, over the previous expressions, the final expressions are thus obtained,

$$\begin{aligned} L_B D_B U_B s_B = \hat{r}_B, L_B D_B U_B \hat{s}_B = Fw_C \ \Rightarrow \\ y_C := \hat{r}_C - Gs_B, \ w_B := s_B - \hat{s}_B \end{aligned} \tag{11}$$

To summarize the previous description of the method, the application of the preconditioner requires, at each level, two SPMV, solving two linear systems with coefficient matrix of the form LDU, and a few vector kernels.

C. Data-Parallel Variants of ILUPACK

ILUPACK has proved to be highly effective at reducing the number of iterations necessary for Krylov subspace-based iterative methods to converge to an acceptable solution for many sparse linear systems [3], [10], [11]. The primary goal of our efforts is to reduce the cost of the iteration, by exploiting the data-level parallelism present in the application of the preconditioner and remaining operations of the solver using GPUs. In this sense, this work expands the set of GPU-aware implementations of iterative methods for general systems offered by ILUPACK, by adding a data-parallel version of BiCGStab. Let us first review the data-parallel implementations of ILUPACK that were already presented in [7], before we introduce our new contributions in the remaining sections of the paper.

For many of the systems that we evaluated in [7], the computational cost required to apply the preconditioner was dominated by the sparse triangular system solves (SPTRSV) and the SPMV appearing in (11). We relied on NVIDIA CUSPARSE to perform these operations in the GPU, since this library provides efficient implementations of the necessary kernels and supports the most common sparse matrix formats. The rest of the operations are mainly vector operations: diagonal matrix scalings and reorderings, which gained mild importance only for highly sparse matrices of large dimension, and were accelerated in our codes via *ad-hoc* CUDA kernels.

The use of CUSPARSE forced us to adapt the sparse storage layouts employed by ILUPACK to those supported by CUSPARSE. The data structure that ILUPACK employs to maintain the multilevel preconditioner is essentially a linked list that contains the information computed at each level. This data includes, for each level, the pointers to the submatrices that form the ILU factorization (the submatrix that comprises the LDU-factored of B, and the G and F rectangular matrices), pointers to the vectors involved in the permutations and diagonal scalings ($\tilde{D}$, $\tilde{P}$, and $\hat{P}$), and some metadata (sizes, types of matrix, etc.); see subsection II-A. ILUPACK stores the factorization of B in a modified CSR format [12], with the L_B and U_B^T factors kept, by columns, in an interlaced manner. Concretely, only the strict lower triangular part of L_B is recorded, as it is unit diagonal; furthermore, the diagonal entries contain the inverses of those of U_B.

In order to invoke CUSPARSE, we thus needed to split each factorization into separate $\bar{L}$ and $\bar{U}$ factors, stored by rows in the conventional CSR format. This transformation was done only once, during the calculation of each level of the preconditioner, and occurred entirely in the CPU. After that, the $\bar{L}$ and $\bar{U}$ factors in CSR format were transferred to the GPU, where the triangular systems involved in the preconditioner application were solved via two consecutive calls to `cusparseDcsrsv_solve`. The analysis phase required by the CUSPARSE solver, which gathers information about the data dependencies and aggregates the rows of the triangular matrix into levels, was executed only once for each level of the preconditioner, and it ran asynchronously with respect to the host CPU.

Considering the computation of the SPMV in the GPU, G and F were also transferred to the device during the computation of the preconditioner. As these matrices are maintained in ILUPACK using the CSR format, no reorganization was needed prior to the invocation of the CUSPARSE kernel for SPMV.

In addition to the application of the preconditioner, we further enhanced the iterative solvers by off-loading the SPMV involving A to the GPU. For this purpose, the coefficient matrix was transferred to the GPU memory before the iterative solve commences, residing there until completion. The coefficient matrix A was stored in CSR format, and the SPMV was computed via the kernel for this purpose in CUSPARSE. The BiCG solver also involves a SPMV with the transposed matrix A^T, which was computed by calling the kernel in CUSPARSE with the transposed parameter set. As in the case of the products that involve F^T and G^T, a performance improvement could be obtained by explicitly storing A^T in the accelerator, at the cost of a considerable memory overhead.

III. IMPLEMENTATION OF GPU-BASED BICGSTAB

The BiCGStab method [8] is one of the most widespread iterative solvers for general linear systems [1] for which, unfortunately, there is no support in the current distribution of ILUPACK.

The implementation of all solvers in ILUPACK follows a reverse communication approach, in which the backbone of the method is performed by a serial routine that is repeatedly called. This routine is re-entered at different points, and sets flags before exiting so that operations such as SPMV, the application of the preconditioner, and convergence checks can be then performed by external routines implemented by the

Operation	kernel
$A \rightarrow M$	Compute preconditioner
Initialize $x_0, r_0, \hat{r}_0, v_0, \rho_0, \alpha_0, \omega_0, \tau_0; k := 0$	
while $(\tau_k > \tau_{\max})$	
$\rho_{k+1} = (\hat{r}_0, r_k)$	DOT product
$\beta = (\rho_{k+1}/\rho_k)(\alpha/\omega_k)$	
$p_{k+1} = r_k + \beta(p_k - \omega_k v_k)$	2×AXPY
$v_{k+1} = M^{-1}Ap_{k+1}$	SPMV + apply prec.
$\alpha = \rho_{k+1}/(\hat{r}_0, v_{k+1})$	DOT product
$s = r_k - \alpha v_{k+1}$	AXPY
$t = M^{-1}As$	SPMV + apply prec.
$\omega_{k+1} = (t, s)/(t, t)$	2×DOT product
$x_{k+1} = x_k + \alpha p_k - \omega_{k+1}s$	2×AXPY
$r_{k+1} = s - \omega_{k+1}t$	AXPY
$\tau_{k+1} :=\| r_{k+1} \|_2$	DOT product
$k := k + 1$	
end while	

Fig. 1. Algorithmic formulation of the preconditioned BiCGStab method. $\tau_{\max}$ is an upper bound on the relative residual for the computed approximation to the solution.

user. This is aligned with the decision adopted by ILUPACK to employ SPARSKIT[2] as the backbone of the solvers. The rationale behind this *modus operandi* is that the implementation of the SPMV and the application of the preconditioner can vary according to the characteristics of the coefficients matrix of the linear system, so it makes sense to provide as much flexibility as possible. In fact, special classes of sparse matrices, like band matrices, can greatly benefit from custom implementations of these kernels, and there are even cases, such as Toeplitz matrices, where the coefficents do not need to be stored explicitly and the SPMV is performed by using closed form expressions.

Despite of the flexibility provided regarding the most computationally important kernels, SPARSKIT solvers have the disadvantage of performing the backbone of the method, along with most vector operations, in serial Fortran code. Sticking to this model when incorporating GPU implementations of the SPMV and the SPTRSV means having to transfer data to and from the GPU.

As BiCGStab can be expressed and efficiently implemented in terms of BLAS operations and the SPMV and SPTRSV kernels, and our main motivation is that of incorporating an implementation of BiCGStab that can exploit the benefits of ILUPACK multilevel preconditioner, we can safely sacrifice the flexibility provided by the reverse communication strategy and depart from that model. In this sense, our implementation encapsulates the method in one monolithic procedure that receives as inputs, among other parameters, the right-hand side vector, a tolerance and the initial guess, and produces the approximate solution to the system in response.

The implementation offloads the entire solver to the GPU, and is based on the algorithm described in Figure 1. The right hand side of the system is transferred to the GPU before the iteration commences and the solution vector is sent back to the CPU memory once the iteration is completed. This strategy allows to minimize the required data transferences, which is especially important in sparse algebra kernels, where the computational effort dedicated to data transferences is in general large in comparison with the one corresponding to actual floating point operations

Following the main ideas in [13] we rely on CUSPARSE library to perform the SPMV and the sparse triangular solvers included in the preconditioner. For the vector operations we also rely on CUBLAS library. Although more efficient implementations of these kernels are sometimes possible, relying on a mature and continuously improved library has several advantages as, for example, adapting to new GPU technologies without having to modify the code.

With the purpose of counting with a baseline version to compare our data-parallel implementation, we developed a CPU version of the algorithm. In particular, our CPU implementation of BiCGStab relies on a subset of the BLAS routines distributed with ILUPACK and ad-hoc SPMV and SPTRSV kernels. However, a different implementation of BLAS as well as other implementation of the sparse kernels could be used without any major modification.

IV. EXPERIMENTAL EVALUATION

In this section we initially describe the test cases and hardware platform employed in the experimental evaluation, next we briefly present the data parallel versions with which this proposal will be compared and, finally we present and analyze the experimental results.

A. Experimental Setup

All experiments in this paper were carried out in IEEE double-precision arithmetic, using a server equipped with an Intel(R) Xeon(R) CPU E5-2620 v2 (six cores at 2.10GHz), and

[2]Available at http://www-users.cs.umn.edu/~saad/software/SPARSKIT/.

128 GB of DDR3 RAM memory. The platform also contains a NVIDIA "Kepler" K40m GPUs, each with 2,880 CUDA cores and 12 GB of GDDR5 RAM.

The CPU codes were compiled with the Intel(R) Parallel Studio 2016 (update 3) with the `-O3` flag set. The GPU compiler and the CUSPARSE library were those in version 6.5 of the CUDA Toolkit. In all cases, the total execution time includes the cost of transferring the problem data and final result between the main memory and the GPU.

B. Test Cases

1) Laplacian: We considered the Laplacian equation $\Delta u = f$ in a 3D unit cube $\Omega = [0,1]^3$ with Dirichlet boundary conditions $u = g$ on $\delta\Omega$. The discretization consists in a uniform mesh of size $h = \frac{1}{N+1}$ and a seven-point stencil is used. The resulting linear system $Au = b$ has an s.p.d. coefficient matrix with seven nonzero elements per row, and $n = N^3$ unknowns. We performed experiments with $N = 200$ and 252, which results in two s.p.d. linear systems of order $n \approx$ 8M and 16M, respectively; see Table I for details.

2) UFMC: We selected a variety of large-scale matrices from the UFMC benchmark collection; see Table I.

3) Convection-Diffusion Problems (CDP): In addition, we considered the PDE $\varepsilon\Delta u + b * u = f$ in Ω, where $\Omega = [0,1]^3$. For this example, we use homogeneous Dirichlet boundary conditions, i.e. $u = 0$ on $\partial\Omega$. The diffusion coefficient ε is set to 1, and the convective functions $b(x,y,z)$ are given by:

$$\begin{array}{ll} \text{conv. in } x\text{-direction:} & [1,0,0], \\ \text{diagonal convection:} & \frac{1}{\sqrt{3}}[1,1,1], \\ \text{circular convection:} & [\frac{1}{2} - z, x - \frac{1}{2}, \frac{1}{2} - y]. \end{array}$$

The domain is discretized with a uniform mesh of size $h = \frac{1}{N+1}$ resulting in a linear system of size N^3. For the experiments we chose a value of $N = 200$; see Table I. For the diffusion part $-\varepsilon\Delta u$ we use a seven-point-stencil. The convective part $b * u$ is discretized using up-wind differences.

C. Evaluation

We start the experimental evaluation by comparing the performance of the CPU and GPU versions of the BiCGStab method.

Table II compares our CPU and GPU variants of BiCGStab. In addition to the total execution time of the solver, we present the average time (and speed-up) per iteration, as in some cases the number of iterations of the CPU and GPU versions differ slightly. The discrepancies are small and occur for those cases with higher condition number, which are more susceptible to floating-point rounding errors.

Regarding the use of the GPU, the acceleration factor for the solver iteration, when it is compared with the baseline CPU variant, varies between 5 and $10\times$, with the exact speed-up depending on characteristics of the problem such as its dimension, the sparsity pattern of the coefficient matrix, and the sparsity of the incomplete factors produced by the multilevel ILU factorization underlying ILUPACK. Speedup values in this range should be expected since, although the CPU version uses only one core, this is a memory-bound problem and the peak memory bandwith of the K40 GPU is only $5.6\times$ larger than that of the Xeon processor.

In the other hand, it is interesting to compare the performance of our data-parallel version of BiCGStab with the previous GPU accelerated solvers developed for ILUPACK. Table III summarizes the runtime taken by two of these solvers, the BiCG and GMRES methods, which were presented in [14]. In both cases, only the SpMV and the application of the preconditioner are offloaded to the accelerator, while performing the rest of the solver in the CPU. The experiments were re-executed on the current platform so the runtime values may differ with the ones in [14].

The execution times observed for BiCGStab highlight the benefits of including this method in the suite of CPU solvers supported by ILUPACK, as it generally attains better convergence rates and delivers lower execution times than the GMRES and BiCG counterparts. Specifically, the BiCGStab reaches a better relative residual in all cases except for the Freescale1, where the BiCG method, is slightly better. From the performance perspective the runtime reductions are all in favour of the BiCGStab with benefits between 1.3 and $8.2\times$.

V. Concluding Remarks and Future Work

We have proposed and evaluated a data-parallel implementation of BiCGStab method, a well-known iterative method, enhanced with the inverse-based multilevel preconditioner of ILUPACK, for the solution of large and sparse linear systems of equations. Our results report considerable speed-ups with respect to the massively parallel solvers included into ILUPACK in previous works. At this point, we remind that, except for the previous version of our data-parallel solvers upon which our new codes are built on, there exists no other parallel version of ILUPACK solvers for general linear systems. Therefore, the acceleration factors that we report are those that a user of the solvers integrated into ILUPACK can presently expect.

As part of future work, we plan to address the new bottlenecks, in particular, by developing optimized SpTrSV and SpMV kernels, specifically tailored to the operations that appear during the application of ILUPACK's preconditioner, as an alternative to those from CUSPARSE.

Acknowledgement

J. I. Aliaga and E. S. Quintana-Ortí were supported by project TIN2017-82972-R of the MINECO and FEDER. E. Dufrechou y P. Ezzatti acknowledge the support of Programa de Desarrollo de las Ciencias Básicas, the Agencia Nacional de Investigación e Innovación, and the Universidad de la República of Uruguay.

References

[1] Y. Saad", *"Iterative Methods for Sparse Linear Systems"*, 2nd ed. Philadelphia, PA, USA: SIAM, 2003.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

TABLE I
MATRICES USED IN THE EXPERIMENTS: DIMENSION n AND NUMBER OF NONZEROS nnz.

Collection	Matrix	n	nnz	nnz/n
Laplacian	A200	8,000,000	31,880,000	3,99
	A252	16,003,008	63,821,520	3,99
UFMC	cage14	1,505,785	27,130,349	18.02
	Freescale1	3,428,755	17,052,626	4.97
	rajat31	4,690,002	20,316,253	4.33
	cage15	5,154,859	99,199,551	19.24
CDP	circular	8,000,000	55,760,000	6,97
	diagonal	8,000,000	55,760,000	6,97
	unit-vector	8,000,000	55,760,000	6,97

TABLE II
EXECUTION TIME OF NEW DATA-PARALLEL VERSION OF BICGSTAB: NUMBER OF ITERATIONS FOR CONVERGENCE; TOTAL EXECUTION TIME (IN SEC.); AVERAGE TIME PER ITERATION; AVERAGE SPEED-UP PER ITERATION OF BICGSTAB GPU VERSION; AND RELATIVE RESIDUAL. IN THE GPU VARIANT ALL KERNELS PROCEED IN THE GPU.

Matrix	Device	#Iter.	Total time	Avg. time per iter.	Avg. speed-up per iter. with respect to...	Relative residual
A200	CPU	3	4.27	1.423	8.21	1.40E-11
	GPU	3	0.52	0.173	–	1.40E-11
A252	CPU	3	7.16	2.386	7.02	1.30E-11
	GPU	3	1.02	0.340	–	1.30E-11
cage14	CPU	3	1.93	0.643	5.36	3.40E-10
	GPU	3	0.36	0.120	–	3.40E-10
Freescale1	CPU	92	54.21	0.589	9.07	2.30E-03
	GPU	87	5.65	0.064	–	2.20E-03
rajat31	CPU	2	1.62	0.810	5.40	7.80E-09
	GPU	2	0.30	0.150	–	7.80E-09
cage15	CPU	3	7.28	2.426	6.07	5.40E-10
	GPU	3	1.20	0.400	–	5.40E-10
circular	CPU	115	239.84	2.085	8.07	6.90E-08
	GPU	100	25.84	0.258	–	3.90E-08
diagonal	CPU	111	281.80	2.538	10.09	6.60E-08
	GPU	114	28.68	0.251	–	5.20E-08
unit-vector	CPU	115	240.07	2.087	8.22	4.70E-08
	GPU	108	27.42	0.253	–	4.00E-08

[3] M. Bollhöfer and Y. Saad, "Multilevel preconditioners constructed from inverse–based ILUs," *SIAM J. Sci. Comput.*, vol. 27, no. 5, pp. 1627–1650, 2006.

[4] J. I. Aliaga, M. Bollhöfer, A. F. Martín, and E. S. Quintana-Ortí, "Exploiting thread-level parallelism in the iterative solution of sparse linear systems," *Parallel Computing*, vol. 37, no. 3, pp. 183–202, 2011.

[5] ——, "Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors," in *Applied Parallel and Scientific Computing, LNCS*, 2012, vol. 7133, pp. 162–172.

[6] J. I. Aliaga, R. M. Badia, M. Barreda, M. Bollhöfer, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, "Exploiting task and data parallelism in ILUPACK's preconditioned CG solver on NUMA architectures and many-core accelerators," *Parallel Computing*, vol. 54, pp. 97–107, 2016.

[7] J. I. Aliaga, M. Bollhöfer, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, "A data-parallel ILUPACK for sparse general and symmetric indefinite linear systems," in *Lecture Notes in Computer Science, 14th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms – HeteroPar'16*. Springer, 2016, to appear.

[8] H. 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*, vol. 13, no. 2, pp. 631–644, 1992. [Online]. Available: https://doi.org/10.1137/0913035

[9] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1–25, 2011.

[10] O. Schenk, A. Wächter, and M. Weiser, "Inertia Revealing Preconditioning For Large-Scale Nonconvex Constrained Optimization," *SIAM J. Scientific Computing*, vol. 31, no. 2, pp. 939–960, 2008.

[11] M. Bollhöfer, M. J. Grote, and O. Schenk, "Algebraic multilevel preconditioner for the helmholtz equation in heterogeneous media," *SIAM Journal on Scientific Computing*, vol. 31, no. 5, pp. 3781–3805, 2009.

[12] V. Eijkhout, "LAPACK working note 50: Distributed sparse data structures for linear algebra operations," Knoxville, TN, USA, Tech. Rep., 1992.

[13] M. Naumov, "Incomplete-LU and Cholesky preconditioned. Iterative methods using CUSPARSE and CUBLAS," NVIDIA white paper, 2011.

[14] J. I. Aliaga, M. Bollhöfer, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, "A data-parallel ILUPACK for sparse general and symmetric indefinite linear systems," in *Euro-Par 2016: Parallel Processing Workshops - Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers*, 2016, pp. 121–133. [Online]. Available: https://doi.org/10.1007/978-3-319-58943-5_10

TABLE III

EXECUTION TIME OF NEW DATA-PARALLEL VERSION OF BICGSTAB: NUMBER OF ITERATIONS FOR CONVERGENCE; TOTAL EXECUTION TIME (IN SEC.); AVERAGE TIME PER ITERATION; AVERAGE SPEED-UP PER ITERATION OF BICGSTAB GPU VERSION; AND RELATIVE RESIDUAL. IN THE GPU VARIANT ALL KERNELS PROCEED IN THE GPU.

Matrix	solver	#Iter.	Total time	Speed-up with respect to...	Relative residual
A200	GMRES	8	4.27	8.21	4.00E-09
	BiCG	14	1.84	3.54	5.30E-09
	BiCGStab	3	0.52	–	1.40E-11
A252	GMRES	8	7.16	7.02	3.90E-09
	BiCG	14	3.62	3.55	5.80E-09
	BiCGStab	3	1.02	–	1.30E-11
cage14	GMRES	3	0.49	1.36	2.40E-09
	BiCG	3	0.68	1.89	2.70E-09
	BiCGStab	3	0.36	–	3.40E-10
Freescale1	GMRES	92	7.33	1.30	6.30E-03
	BiCG	87	26.96	4.77	1.00E-03
	BiCGStab	87	5.65	–	2.20E-03
rajat31	GMRES	2	0.67	2.23	3.60E-07
	BiCG	2	0.88	2.93	1.40E-06
	BiCGStab	2	0.30	–	7.80E-09
cage15	GMRES	3	1.61	1.34	4.80E-09
	BiCG	3	2.33	1.94	5.50E-09
	BiCGStab	3	1.20	–	5.40E-10
circular	GMRES	115	95.37	3.69	1.40E-06
	BiCG	100	86.97	3.37	1.20E-07
	BiCGStab	100	25.84	–	3.90E-08
diagonal	GMRES	111	114.52	3.99	1.60E-06
	BiCG	114	89.77	3.13	2.00E-07
	BiCGStab	114	28.68	–	5.20E-08
unit-vector	GMRES	115	119.61	4.36	1.40E-06
	BiCG	108	95.33	3.48	4.10E-08
	BiCGStab	108	27.42	–	4.00E-08