CLEI electronic journal, Volume 24, Number 1, Paper 6 , April 2021

Accelerating advanced preconditioning methods on hybrid architectures

Ernesto Dufrechou
Universidad de la República, Facultad de Ingeniería,
Montevideo, Uruguay, 11300,
edufrechou@fing.edu.uy

Abstract

Many problems, in diverse areas of science and engineering, involve the solution of large-scale sparse systems of linear equations. In most of these scenarios, they are also a computational bottleneck, and therefore their efficient solution on parallel architectures has motivated a tremendous volume of research.
This dissertation targets the use of GPUs to enhance the performance of the solution of sparse linear systems using iterative methods complemented with state-of-the-art preconditioned techniques. In particular, we study ILUPACK, a package for the solution of sparse linear systems via Krylov subspace methods that relies on a modern inverse-based multilevel ILU (incomplete LU) preconditioning technique.
We present new data-parallel versions of the preconditioner and the most important solvers contained in the package that significantly improve its performance without affecting its accuracy. Additionally we enhance existing task-parallel versions of ILUPACK for shared- and distributed-memory systems with the inclusion of GPU acceleration. The results obtained show a sensible reduction in the runtime of the methods, as well as the possibility of addressing large-scale problems efficiently.

Keywords: Linear Systems, preconditioning technique, massively parallel processing

1 Introduction

Sparse systems of linear equations appear in many areas of knowledge, such as circuit simulation, optimal control, quantum mechanics or economics [1, 2]. For example, they emerge as a natural consequence of the discretization of PDEs, which makes their efficient solution extremely relevant. A number of current real-world applications (as for example three-dimensional PDEs) involve linear systems with millions of equations and unknowns. Direct solvers such as those based on Gaussian Elimination (GE) [3], which apply a sequence of matrix transformations to reach an equivalent but easier-to-solve system, today fall short when solving large-scale problems because of their excessive memory requirements, impractical time to solution and complexity of implementation.

In these cases, iterative methods pose an appealing alternative. These sort of methods can be interpreted as searching for the solution of the linear system inside a predefined subspace, by taking one step in a given search direction at each iteration of the solver. A large number of methods exist that differ from each other in how this subspace is defined, and the criteria under which this search direction and step size are chosen.

In general, it is desired that the iterative method rapidly improves the initial guess, with the purpose of finding an acceptable solution to the system in a small number of steps. However, this is often impaired by unavoidable rounding errors due to the use of finite-precision arithmetic, which are amplified according to the numerical properties of each problem.

To remedy these shortcomings, preconditioning techniques are applied. In a broad sense, these techniques aim to transform the original linear system into an equivalent one that presents better numerical properties, so that an iterative method can be applied and converge faster to a solution.

The development of effective preconditioners is an active field of research in applied mathematics. Among the most versatile and widely-used general-purpose preconditioners, Incomplete LU (ILU) factorizations occupy a place of privilege. These are based on computing an LU factorization of the matrix where some of the entries that become nonzero during the process are dropped (replaced by 0) to keep the factors sparse.

However, an active line of research is devoted to make ILUs more efficient and reliable, so they can be applied in a wider spectrum of problems.

Among the most recent advances in this field, ILUPACK (`http://ilupack.tu-bs.de`) stands out as a package for the solution of sparse linear systems via Krylov subspace methods that relies on an inverse-based multilevel ILU (incomplete LU) preconditioning technique [4]. A remarkable characteristic of ILUPACK is its unique control of the growth in the magnitude of the inverse of the triangular factors during the approximate factorization process.

Unfortunately, the favorable numerical properties of ILUPACK's preconditioner in the context of an iterative solvers come at the cost of expensive construction and application procedures, especially for large-scale sparse linear systems. This high computational cost motivated the development of task-parallel implementations of ILUPACK for shared-memory and message-passing platforms [5, 6, 7] but, despite showing good performance and scalability results, these variants of ILUPACK are limited to the solution of symmetric and positive-definite (SPD) linear systems, and they slightly modify the preconditioner to exploit task-parallelism. This means that these modifications can impact the numerical properties and convergence of the preconditioner.

Due to its rapid development in the last two decades, GPUs are now ubiquitous, making their efficient use crucial in order to obtain good performance on the most recent hardware for scientific computing. Even in sparse linear algebra where the computational intensity of the operations is generally low and does not allow to take full advantage of the computational power that GPUs provide, the high memory bandwidth of these devices offers significant acceleration opportunities.

Previous to this dissertation, no parallel versions of ILUPACK existed aside the above-mentioned task-parallel implementations. It is therefore interesting to analyze the data-level parallelism in ILUPACK to develop new efficient parallel versions that do not suffer from the limitations of the previous variants on the one hand, and to enhance the performance of these variants on the other. In this line, the use of hardware accelerators, and in particular of GPUs, is a valuable tool, and a clear path to explore.

This work extends the summary of the dissertation submitted to the Latin-American Contest of PhD. Theses (CLTD) of the CLEI in 2019. The main new contributions summarized in this manuscript are:

1. The evaluation of a task- and data-parallel implementation of the BiCG solver for single-GPU systems that relies on GPU Streams (Section 5.2).
2. A detailed evaluation of the data-parallel ILUPACK preconditioner using synchronization-free sparse triangular solvers, performed on 86 matrices from the SuiteSparse matrix collection (Section 6).
3. The development of a data-parallel variant of the BiCGStab method with the ILUPACK preconditioner, especially tailored for low-power devices. The experimental evaluation is performed on the Jetson AGX Xavier board, one of the latest low-power computing platform from NVIDIA (Section 8).

1.1 Objectives

The main goal of the thesis is to advance the state-of-the-art in the efficient parallel implementations of modern preconditioning techniques. In particular, we are interested in the use of hardware accelerators to leverage the data-parallelism of iterative solvers working together with advanced incomplete-factorization methods. As exposed previously, ILUPACK is a prominent example of such solvers, but its remarkable numerical results come at the expense of a high complexity, which implies costly construction and application procedures.

Therefore, to achieve our principal goal we have set the following specific objectives:

- Enable the use of the GPU to accelerate ILUPACK's multilevel-preconditioner, harnessing the data-parallelism in the operations that compose its application for different matrix types.
- Evaluate and improve ILUPACK solvers, identifying the principal factors that limit their performance.
- Extend ILUPACK by adding new accelerated solvers.
- Enhance the existing task-parallel versions of ILUPACK by enabling the exploitation of data-parallelism.

2 Sparse systems of linear equations

Gaussian Elimination can only take partial advantage of the coefficient matrix being sparse. Although there are some exceptions regarding specific classes of matrices like banded or tridiagonal, the problem of fill-in makes it necessary to abandon this technique to solve the general sparse case for large-scale scenarios.

An iterative method for the solution of linear systems is one that, starting from an initial guess x_0 to the solution vector x, is capable of iteratively refining it until (under certain conditions) an approximation x_k that is acceptably close to x is reached, i.e. $\|Ax_k - b\|$ is relatively small for some vector norm. In order to

avoid the difficulties that make GE impractical, this approximation should preserve the number of nonzero entries of the problem throughout the process. This is achieved by exploiting a matrix primitive that is well suited for sparse matrices, concretely, the matrix-vector product.

The intuitive concept behind these methods is that one should be able to improve an approximate solution x_k by finding an appropriate linear combination of b, x_k and Ax_k. If one is to use all the information available in the expression $Ax = b$, it is reasonable to set the initial guess x_0 to some multiple of b, which implies that

$$x_0 \in \text{span}\{b\}. \tag{1}$$

Then, following the previous idea, we should replace the current solution x_0 with a linear combination of itself with Ax_0. In this case the current solution x_1 belongs to the subspace $\text{span}\{b, Ab\}$. After k iterations, it is clear that the current iterate will belong to

$$\mathcal{K}_k(A, b) = \text{span}\{b, Ab, A^2b, \ldots, A^{k-1}b\}. \tag{2}$$

This is called the Krylov subspace of dimension k for A and b.

The challenge of this sort of procedure lies on how to construct these linear combinations such that an acceptable approximation is reached in just a few iterations, and this is what gives place to a myriad of different methods. In fact, although there are optimal methods that are able to find the actual solution of the linear system in exact arithmetic, the convergence of the iterations will be determined by the spectral properties of the matrix, and the use of finite precision arithmetic can complicate things even further.

Fortunately, when the properties of the matrix A are not perfect, one can still multiply the coefficient matrix by a certain matrix M^{-1} and solve

$$M^{-1}Ax = M^{-1}b \tag{3}$$

instead of $Ax = b$. Now the approximate solution x_k will belong to the subspace $\mathcal{K}_k(M^{-1}A, x_0)$. Matrix M is known as a preconditioner as its purpose is to improve the "condition number" of the iteration matrix, which is closely related with the way a matrix amplifies the numerical errors due to finite precision when, for instance, performing a matrix-vector product. Broadly speaking, M^{-1} is chosen so that it resembles A^{-1} in some way, but this choice is nothing but trivial and is the subject of active research. Moreover, if inverting M or solving a linear system with M is required, such inversion or system should be easier to solve than the original one, or the whole exercise would be pointless.

2.1 ILUPACK

Consider the linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is large and sparse, and both the right-hand side vector b and the sought-after solution x contain n elements. ILUPACK provides software routines 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 iterative solvers. 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 user. This is aligned with the decision adopted by ILUPACK to employ SPARSKIT[1] as the backbone of the solvers. When using an iterative solver enhanced with ILUPACK preconditioner, the application of the preconditioner, which occurs (at least once) per iteration of the solver, is the most demanding task from the computational point of view.

The computation of the preconditioner proceeds following three steps:

1. Initially, a pre-processing 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 $\hat{A} = LDU + E$, where $L, U^T \in \mathbb{R}^{n\times n}$ are unit lower triangular factors, $D \in \mathbb{R}^{n\times n}$ is (block) diagonal, and E is a small term that contains the dropped entries during the process. In some detail, $\hat{A}$ is processed in this stage to obtain the partial ILU factorization:

$$\hat{P}^T\hat{A}\hat{P} \equiv \begin{pmatrix} B & F \\ G & C \end{pmatrix} = LDU + E = \begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & S_c \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} + E. \tag{4}$$

   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, and S_C represents the approximate Schur complement assembled from the "rejected" rows and columns.

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

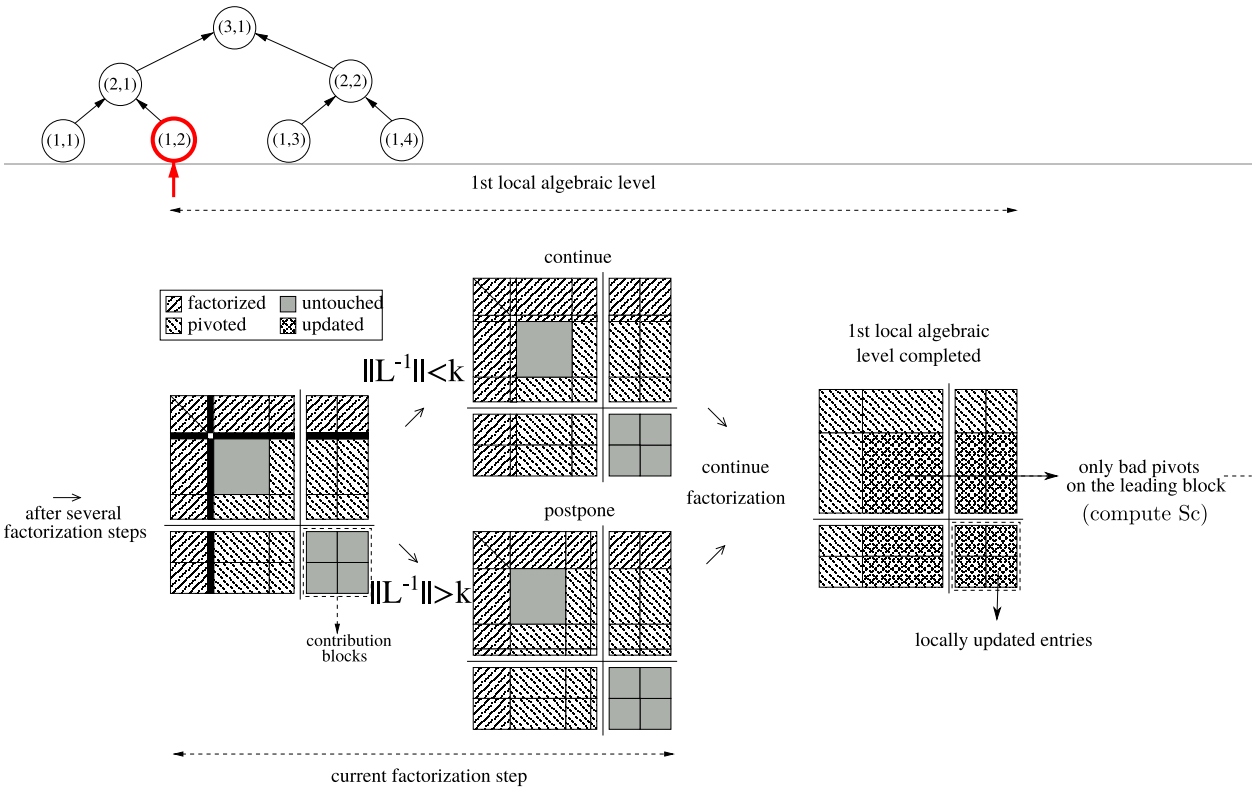

Figure 1: A step of the computation of the preconditioner in the task-parallel version of ILUPACK [5].

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

$$M_l \approx \tilde{D}^{-1}\tilde{P}\hat{P}\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix}\begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix}\begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix}\hat{P}^T\tilde{P}^T\tilde{D}^{-1}, \tag{5}$$

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 $l+1$.

This means that the application of the preconditioner requires, at each level, two sparse matrix-vector products (SpMv), solving two linear systems with coefficient matrix of the form LDU, and a few vector kernels. The derivation of the procedure is described in [8].

2.2 Task-parallel ILUPACK

The task-parallel version of ILUPACK employs Nested Dissection (ND) [9] to reorder the original matrix such that it can be partitioned into a number of decoupled blocks, and then delivers a partial Incomplete Cholesky (IC) factorization with some differences with respect to the sequential procedure. The main change is that the computation is restricted to the leading block, and therefore the rejected pivots are moved to the bottom-right corner of the leading block, instead of being moved to the bottom-right corner of the whole matrix; see Figure 1. Although the recursive definition of the preconditioner is still valid in the task-parallel case, some recursion steps are now related to the edges of the corresponding preconditioner DAG. Therefore different DAGs involve distinct recursion steps yielding distinct preconditioners, which nonetheless exhibit close numerical properties to that obtained with the sequential ILUPACK.

As the definition of the recursion is maintained, the operations to apply the preconditioner remain valid. However, to complete the recursion step in the task parallel case, the DAG has to be crossed two times per solve $z_{k+1} := M^{-1}r_{k+1}$ at each iteration of the Preconditioned Conjugate Gradient (PCG) solver: once from bottom to top and a second time from top to bottom (with dependencies/arrows reversed in the DAG). More details can be found in [6].

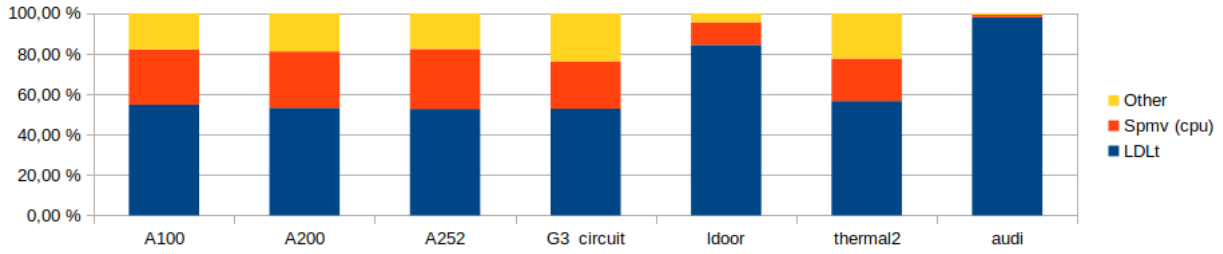

Figure 2: The cost of each iteration (for CG in this case) is dominated by the application of the preconditioner. The cost of the preconditioner application is, at the same time, dominated by triangular system solves and sparse matrix-vector products.

3 Related work

Many research works have reported important benefits for the solution of sparse linear algebra problems on many-core platforms. However, many of these efforts address only non-preconditioned versions of the methods, where the sparse matrix-vector product (SpMv) is the main bottleneck. For example, Buatois et al. [10] implemented the CG method in conjunction with a Jacobi preconditioner, using the block compressed sparse row (BCSR) format. Later, Bell and Garland [11] addressed the SpMv, including several sparse storage formats, that became the basis for the development of the CUSP library [12].

A parallel CG solver with preconditioning for the Poisson equation optimized for multi-GPU architectures was presented by Ament et al. [13]. Sudan et al. [14] introduced GPU kernels for the SpMv and the block-ILU preconditioned GMRES in flow simulations, showing promising speed-ups. At the same time, Gupta completed a master thesis [15] implementing a deflated preconditioned CG for Bubbly Flow.

Naumov [16] produced a solver for triangular sparse linear systems in a GPU, one of the major kernels for preconditioned variants of iterative methods such as CG or GMRES. Later, the same author extended the proposal to compute incomplete LU and Cholesky factorizations with no fill-in (ILU0) in a GPU [17]. The performance of the aforementioned algorithms strongly depends on the nonzero pattern of the coefficient matrix.

Li and Saad [18] studied the GPU implementation of SpMv with different sparse formats to develop data-parallel versions of CG and GMRES. Taking into account that the performance of the triangular solve for CG and GMRES, preconditioned with IC and ILU respectively, was rather low on the GPU, a hybrid approach was proposed in which the CPU is leveraged to solve the triangular systems.

In 2016, Lukarsky et al. [19] proposed a hybrid CPU-GPU implementation of multi elimination ILU preconditioners in GPU.

Recently, He et. al. [20] presented a hybrid CPU-GPU implementation of the GMRES method preconditioned with an ILU-threshold preconditioner to control the fill-in of the factors. In the same work, the authors also propose a new algorithm to compute SpMv in the GPU.

Some of these ideas are currently implemented in libraries and frameworks such as CUSPARSE, CUSP, CULA, PARALUTION and MAGMA-sparse.

4 Data-parallel variants

ILUPACK's multilevel preconditioner is stored as a linked list of structures that contain the information computed at each level. Concretely, a level contains pointers to the submatrices that form the ILU factorization: the B submatrix that comprises the LDU factored upper left block and the G, F rectangular matrices, along with the diagonal scaling and permutation vectors that correspond to $\tilde{D}$, $\tilde{P}$, and $\hat{P}$; see Section 2.1. The case of symmetric matrices is analogous, and the differences reside in that only the lower triangular factor L and the (block-)diagonal D of the LDL^T factorization are stored explicitly, and that G is not stored because it is equal to F^T.

As shown in Figure 2, the computational cost required to apply the preconditioner is dominated by the sparse triangular system solves (SpTrSV) and SpMVs. Nvidia cuSparse [21] library provides efficient GPU implementations of these two kernels that support several common sparse matrix formats. Therefore, it is convenient to rely on this library. The rest of the operations are mainly vector scalings and re-orderings, which gain certain importance only for highly sparse matrices of large dimension, and are accelerated in our codes via *ad-hoc* CUDA kernels.

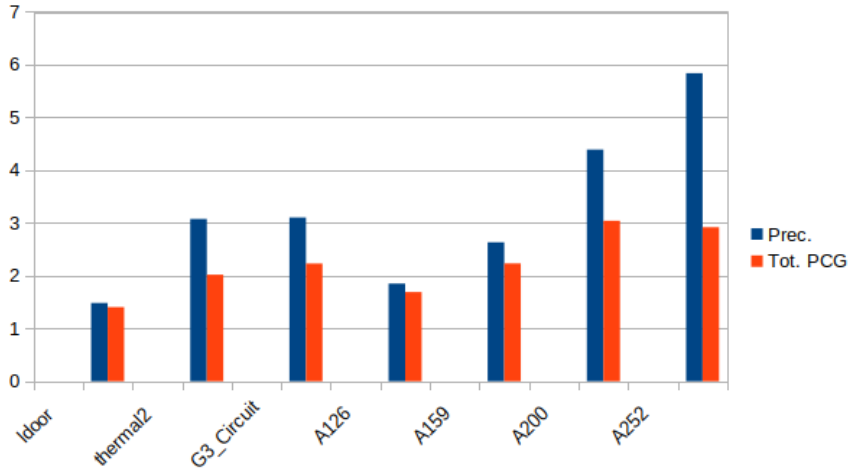

Figure 3: Acceleration factors for a scalable 3D PDE (matrix dimension for AXXX = XXX^3, 4 avg. nonzeros/row) in an Intel Core i3-3220 (2 cores @ 3.30 GHz / 16GB RAM) / Tesla K20 (2496 cores @ 0.70 GHz / 6GB RAM). ldoor, thermal2, G3 circuit are SPD problems from SuiteSparse Matrix collection.

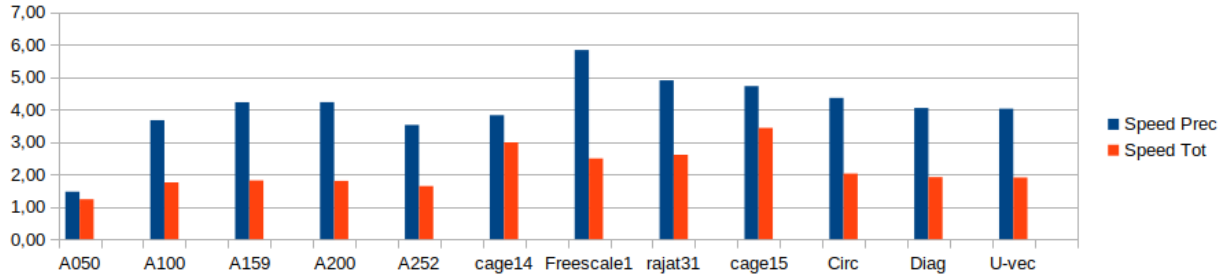

Figure 4: Acceleration factors for GMRES, obtained in an Intel Core i7-4770 (4 cores @ 3.40 GHz / 16GB RAM) connected to a Tesla K40c (2880 cores @ 0.75 GHz / 11GB RAM). Circ, Diag and Unit-vec are large 3D convection-diffusion problems (Dimension = 200^3). Freescale, cage14, cage15 and rajat31 are medium-large nonsymmetric problems from SSMC.

Given the modified-CSR (MCSR) storage layout adopted by ILUPACK for the LDL^T and LDU factors, and the native format handled by the cuSparse library (CSR), a layout reorganization is necessary before the corresponding kernel can be invoked. In the current implementation, this process is performed by the CPU, during the calculation of the preconditioner. 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, is executed only once for each level of the preconditioner, and it runs asynchronously with respect to the host CPU.

In the Symmetric Positive-Definite (SPD) case, the matrix-vector products that need to be computed during the application of the preconditioner, at each level, are of the form $x := Fv$ and $x := F^T v$. To avoid the use of the slow SpMvT operation provided by cuSparse, we allocated both F and F^T in the GPU, using a cuSparse routine to transpose the matrix in the device. We find this performance-storage trade-off acceptable, since the memory footprint of the symmetric solver does not exceed that of the general non-symmetric case. However, in the case of BiCG solver, the application of the preconditioner involves G and F as well as their respective transposes. Here, storing both transposes can exceed significantly the memory requirements of the non-symmetric versions of ILUPACK, as two additional matrices need to be stored. For this reason, our approach in this case only keeps G and F in the accelerator, and makes use of cuSparse SpMvT operation.

As it can be observed in Figure 3 the acceleration factors measured for a scalable 3D PDE, and benchmark problems from the Suite Sparse Matrix Collection (SSMC)[2], in an Intel Core i3-3220 CPU connected to a Nvidia Tesla K20 GPU, are between 1.5 and 6×.

In the case of GMRES, we managed to accelerate the application of the preconditioner by similar factors,

[2]http://faculty.cse.tamu.edu/davis/suitesparse.html

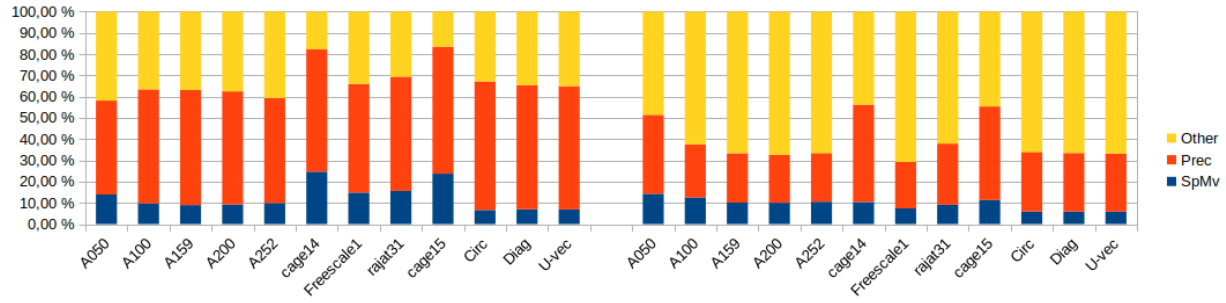

Figure 5: Fraction of the execution time of GMRES taken by the preconditioner and the MGSO procedure before and after accelerating the preconditioner with the GPU.

but other operations gain importance and limit the obtained accelerations to 3×. This situation can be observed in Figures 4 and 5. Specifically, after accelerating the preconditioner, the Modified-GS Orthogonalization (MGSO) involved in GMRES becomes the bottleneck of the solver.

For this reason, we developed a GPU-version for the orthogonalization procedure together with a hybrid implementation of GMRES that leverages the best type of architecture for each operation while, at the same time, limiting the data transferences. Since we already off-loaded the SpMv appearing in GMRES to the accelerator, the basis vectors of MGSO reside in the GPU. The output of the process is the current basis vector, orthogonalized with respect to the remaining vectors, and the coefficients of the current row of the Hessenberg matrix. This matrix is small, and the following application of rotations and triangular solve expose little parallelism, so it is natural to keep it in the CPU memory. The coefficients of the matrix are calculated serially via vector products with the basis vectors, so the GPU computes n flops for each coefficient that is transferred back to the CPU. A similar approach was studied in [20].

Table 1: Runtime and acceleration factors for the MGSO procedure and the enhanced GMRES solver in an Intel E5-2620v2 CPU (6 cores @ 2.10 GHz / 128 GB RAM) connected to a Tesla K40m (2880 cores @ 0.70 GHz / 12GB RAM).

		Time CPU		Time GPU		Speed-up GPU	
Matrix	#Iter.	MGSO	Total	MGSO	Total	MGSO	Total
A200	6	0.57	4.27	0.17	1.61	3.45	2.65
A252	6	1.13	8.25	0.33	3.60	3.41	2.29
cage14	7	0.14	1.54	0.04	0.50	3.73	3.10
Freescale1	46	6.03	21.43	0.97	4.42	6.24	4.85
rajat31	4	0.22	2.23	0.05	0.76	4.14	2.93
cage15	7	0.48	5.54	0.13	1.63	3.77	3.39
circular	203	72.15	317.12	10.14	54.66	7.12	5.80
diagonal	241	86.63	377.17	13.15	68.56	6.59	5.50
unit-vector	252	91.43	394.08	13.20	68.68	6.93	5.74

Table 1 shows the results for the enhanced variant of GMRES, with speed-ups for the orthogonalization procedure that are between 3× and 7×. Combining this with the previous enhancements, we obtain speed-ups of up to 5.8× for the entire solver.

5 Leveraging task and data parallelism in BiCG

The BiCG method was first derived by Lanczos [22] in 1952 as a variation of the two-sided Lanczos algorithm to compute the eigenvalues of a non-symmetric matrix A. In a broad sense, it is based on maintaining two parallel recurrences, one for matrix A and the other for A^T, and imposing bi-conjugacy and bi-orthogonality conditions between the vectors of each recurrence. In Figure 6, we offer an algorithmic description of the method, detailing the corresponding computational kernel on the right column.

It follows from the algorithmic description of the BiCG, that contrary to most iterative linear solvers, in which there exist strict data dependencies that serialize the sequence of kernels appearing in the iteration, the two recurrences involved in the BiCG method are quasi-independent. Moreover, in the preconditioned version of the method, there is no data dependence between the application of the transposed and non-transposed

Operation	kernel
Initialize $x_0, r_0, q_0, p_0,$ $\ldots, s_0, \rho_0, \tau_0; k := 0$	
$A \rightarrow M$	Compute preconditioner
while $(\tau_k > \tau_{\max})$	
$\alpha_k := \rho_k/(q_k^T A p_k)$	SpMv + dot product
$x_k := x_k + \alpha_k p_k$	axpy
$r_k := r_k - \alpha_k A p_k$	axpy
$t_k := M^{-1} r_k$	Apply preconditioner
$z_k := M^{-T} A^T q_k$	SpMv + apply prec.
$s_{k+1} := s_k - \alpha_k z_k$	axpy
$\rho_{k+1} := (s_{k+1}^T r_k)/\rho_k$	dot product
$p_{k+1} := t_k + \rho_{k+1} p_k$	axpy
$q_{k+1} := s_{k+1} - \rho_{k+1} q_k$	axpy
$\tau_{k+1} :=\| r_k \|_2$	dot product
$k := k + 1$	
end while	

Figure 6: Algorithmic formulation of the preconditioned BiCG method.

preconditioner inside the iteration, exposing coarse-grain parallelism at the recurrence-level. Considering this context, we first rearranged the operations in the BiCG method so that these two sequences are isolated, making it possible to execute them concurrently. This idea is summarized in Figure 7, where we group the operations of the BiCG in three sets, namely Set A, Set B, and a third set that contains the rest of the operations. Each of the first two sets contains a single SpMv and the application of one of the preconditioners. Although Set A also contains a dot product and two axpy operations before the synchronization point, these kernels have almost negligible computational cost in general, and this distribution of the workload can be expected to be fairly well-balanced.

5.1 Variant for two GPUs

In systems equipped with two discrete graphics accelerators, the arrangement of the operations of the BiCG proposed in Figure 7 allows to execute each sequence in a different device, until reaching the synchronization point. This enables the exploitation of coarse-grain task-level parallelism, using the two GPUs to execute the operations of the solver that belong to different sets concurrently, in conjunction with the data-level parallelism of each operation, that is leveraged inside each accelerator.

In addition to the concurrent execution, the presence of two GPUs allows to avoid the use of the transposed version of the cuSparse SpMv routine. These operations are required because in the BiCG method both the transposed coefficient matrix A^T and the transposed preconditioner are involved in the calculations. In particular, the solver will perform an SpMv with the transposed matrix in each iteration, and the application of the transposed preconditioner will involve two transposed SpMv per level of the preconditioner in each iteration. In our baseline implementation, these transposed SpMVs are computed by calling the SpMvT kernel of cuSparse on the original non-transposed matrices. The execution times reached using this strategy show that the calls to SpMvT are, on average, 2–3× slower than those to SpMv, with one special case for which it becomes almost 7× slower. As a consequence, the application of the transposed preconditioner sometimes takes more than twice the time of its non-transposed counterpart.

As each GPU (1 and 2) has its own separate memory, we maintain the non-transposed operands in GPU 1 and the transposed ones in GPU 2, using the faster version of cuSparse SpMv in both devices. It should be noted that we are not wasting memory in this case, since using the transposed SpMv routine in the second GPU would also imply the storage of the original matrix in the device, at the same memory cost.

The copy of the data to both devices is performed concurrently with the construction of the preconditioner. In particular, the asynchronous transference of one level takes place while the next one is being computed by the incomplete factorization procedure.

Regarding the concurrent execution in both devices, we use two CPU threads that concurrently enqueue work to each accelerator to ensure the correct overlapping of the two execution streams.

Operation	kernel	
Initialize $x_0, r_0, q_0, p_0,$ $\ldots, s_0, \rho_0, \tau_0; k := 0$ $A \rightarrow M$ **while** $(\tau_k > \tau_{\max})$	Compute preconditioner	
$\alpha_k := \rho_k/(q_k^T A p_k)$ $x_k := x_k + \alpha_k p_k$ $r_k := r_k - \alpha_k A p_k$ $t_k := M^{-1} r_k$	SpMv + dot product axpy axpy Apply preconditioner	Set A
$z_k := M^{-T} A^T q_k$	SpMv + apply prec.	Set B
synchronization $s_{k+1} := s_k - \alpha_k z_k$ $\rho_{k+1} := (s_{k+1}^T r_k)/\rho_k$ $p_{k+1} := t_k + \rho_{k+1} p_k$ $q_{k+1} := s_{k+1} - \rho_{k+1} q_k$ $\tau_{k+1} :=\| r_k \|_2$ $k := k + 1$ **end while**	 axpy dot product axpy axpy dot product	

Figure 7: Algorithmic re-formulation of the preconditioned BiCG method. The steps have been rearranged so that the two sequences that compose the method can be isolated.

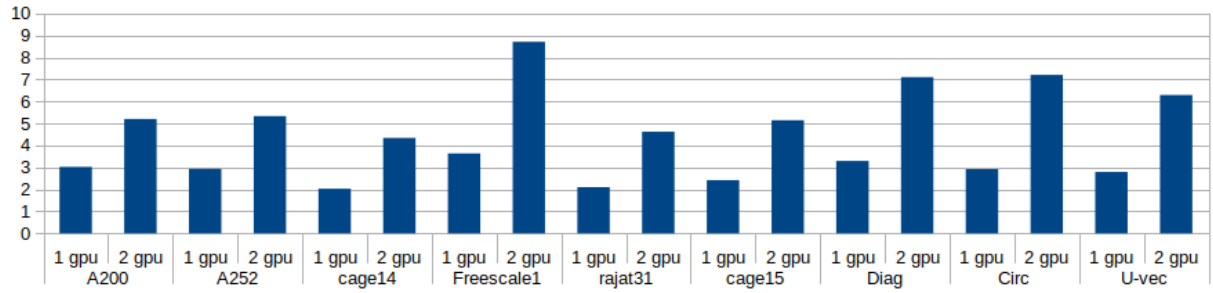

Figure 8: Results for the Adv_BICG_2GPU variants vs. multi-thread MKL in an Intel E5-2620v2 CPU (6 cores @ 2.10 GHz / 128 GB RAM) with 2 Tesla K40m GPUs (2880 cores @ 0.75 GHz / 12GB RAM).

5.2 Leveraging task-parallelism in the GPU using streams

In systems equipped with only one GPU, exploiting the task-level parallelism present in the BiCG method demands more elaborate solutions to obtain significant improvements.

Since the GPU offers the possibility of overlapping the execution of several kernels using streams, it is reasonable to offload the Set A and Set B blocks in Figure 7 to different GPU streams in the same device. However, the achieved overlapping will depend on the resources that one kernel leaves available to the others [23]. If one of the kernels is sufficient to fully occupy the GPU, none overlapping will occur. To study the overlapping that can be achieved between the two sequences using streams, we first evaluate the concurrent execution of two triangular system solvers, which is the main component of the application of the preconditioner.

Table 2: Runtime (in milliseconds) of the solution of two independent triangular linear systems using one and two GPU streams on a GTX 1080 Ti GPU.

Matrix	1 stream	2 streams
cage14	5.19	5.18
cage15	19.90	19.87
Freescale1	5.37	5.34
A050	0.56	0.53
A100	5.33	5.29
A200	62.19	62.07
A252	115.65	115.45

Table 2 shows that, for the tested matrices, almost no overlapping could be achieved. Even if the matrices employed are relatively small (as in the case of matrix A050) the overlapping obtained is negligible. This suggest that the use of GPU streams will not deliver any significant improvement in the parallelization of the BiCG solver.

5.3 Concurrent CPU-GPU variant

Since the use of GPU streams does not seem to offer any advantage with respect to the baseline data-parallel variant in single-GPU systems, an interesting alternative to enable task-level parallelism is to perform computations in the multicore CPU concurrently with computations on the GPU. In this sense, the Adv_BICG_Hyb_Cusp variant employs the multicore CPU to perform the transposed SpMv of BiCG using the multi-threaded MKL library. The rest of the computations are performed in the GPU using cuSparse and cuBlas libraries.

Figure 9 presents the time-line for the Adv_BICG_Hyb_Cusp version, extracted with Nvidia Visual Profiler. The time line consists of four horizontal lines divided into several bars corresponding to the duration of different tasks. The first line corresponds to the execution of CUDA API calls in the CPU thread (kernel launches, parameter setup, memory transferences, etc.); the second line also corresponds to the main CPU thread, and we use it to display the duration of the transposed SpMv in the CPU (yellow bar) and to aggregate the CUDA API calls that correspond to the application of the transposed preconditioner. The two inferior lines correspond to each one of the two GPU streams.

The figure shows that the transposed SpMv considerably overlaps with the application of the preconditioner in the GPU. Additionally, from the analysis of the orange bars in the first line, it follows that an important part of the CPU time is devoted to the processing of CUDA API calls, which we refer to as "kernel launch overhead". This overhead is mainly due to the solution of four triangular linear systems at each level of the ILUPACK preconditioner.

The solution of these operations relies on the routine `cusparseDcsrsv_solve`, of cuSparse library, whose implementation is based on the so-called level-set strategy [24], and is described in [25]. This strategy consists of two main stages. In the analysis phase, the triangular sparse matrix is processed to determine sets of independent rows called *levels*. In the solution phase, the solver launches a GPU kernel to process each of these levels, processing the rows that belong to each level in parallel. The number of levels that derive from the analysis can vary largely according to the nonzero pattern of each triangular matrix, and for matrices of considerable size the variation is usually in the order of hundreds or even a few thousands. In such cases, the overhead due to launching the kernels that correspond to each level can become significant [26].

Figure 9 illustrates how the launching of these kernels delays the start of the multi-threaded transposed SpMv on the CPU.

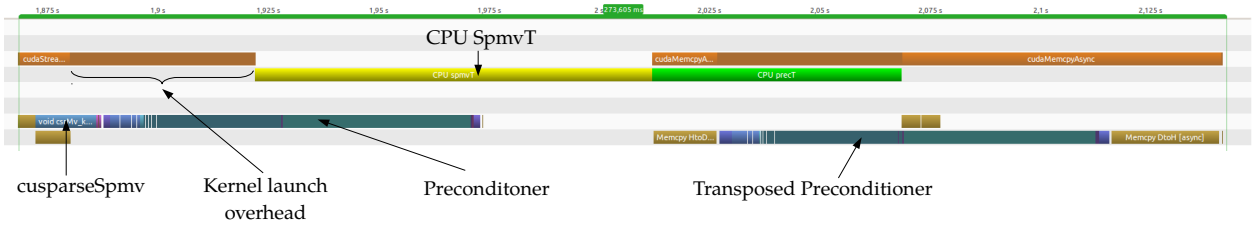

Figure 9: Time-line of the execution of the ADV_BICG_HYB_CUSP version to solve the *cage15* test case in the experimental platform. Extracted with NVIDIA Visual Profiler.

6 Self-scheduled sparse triangular solvers

As Figure 2 shows, the solution of sparse triangular linear systems is the operation that concentrates most of the computation time during the application of the preconditioner. Hence, improving the performance of this kernel can have a significant impact on the performance of ILUPACK.

The parallel algorithms for the solution of this operation can be classified in two main contrasting categories. Apart from two-stage methods based on the level-set paradigm, which was briefly described in Section 5.3, there are one-stage methods that rely on a *self-scheduled* pool of tasks, where some of the tasks wait until the data necessary to perform their computations is made available by other tasks[3].

In the previous sections, we relied on the solver bundled with the CUSPARSE library to perform this operation. This implementation, based on the two-stage paradigm, is publicly available, and it is constantly revised and adapted to the latest NVIDIA GPU architectures. For several years, this has been the reference implementation for this operation on GPUs.

Recently, Liu et al. [34] proposed a new GPU solver for sparse triangular systems, for matrices stored in the CSC format, based on the *self-scheduled* strategy. The method relies on a lightweight analysis phase to determine the number of dependencies of each unknown (which is not trivial in the CSC format), and the use of atomic operations to manage the synchronization of the warps inside the GPU kernel. This new synchronization mechanism avoids the constant synchronization with the CPU that CUSPARSE suffers from.

In [35], we proposed a triangular solver for matrices stored in the CSR format based on Liu's approach. We developed several routines for this operation, outperforming CUSPARSE for a varied set of sparse matrices. Later, in [36], we applied a similar idea to the level-set preprocessing stage used by CUSPARSE, showing remarkable performance gains.

Our baseline algorithm for the parallel solution of a sparse triangular linear system advances row-wise, such that each *warp* processes one row and each thread is assigned to one nonzero element. The threads must actively-wait on the dependency represented by its nonzero entry until all the dependencies of the warp are satisfied, instead of relying on a fixed schedule of kernel launches determined by the level-set analysis of the sparse matrix, as in the CUSPARSE approach. The built-in warp-scheduling mechanism of the GPU prevents the warps from entering in dead-lock.

Unlike CUSPARSE and Liu's solution, our solver can run without any previous analysis phase. However, our strategy faces some limitations when the sparse matrix presents only few nonzero elements per row. Concretely, as each warp (of 32 threads) processes one row, and each thread processes one nonzero, there can be a significant waste of threads (and cores). Moreover, the thread-blocks that are executing concurrently on the multiprocessors of the GPU at a given time (*active thread-blocks*) is limited. Inactive thread-blocks have to wait until the active ones finish their execution before starting their own. To avoid deadlocks, the rows need to be processed in ascending index order, which means that a warp can be ready to execute (have no dependencies) but nevertheless belongs to an inactive thread-block. This situation can be avoided by processing the rows in one of the possible orders that derive from the level-set analysis. The use of the analysis information to produce a fixed execution schedule can also be of benefit on the cases where the sparse matrices present nonzero patterns that imply a high grade of dependencies between their rows.

In [37] we developed new routines that, after performing a level-set analysis of the sparse matrix, leverages this information to overcome some of the above difficulties. In addition to this, the SF_MULTIROW routine partitions the warp into equally-sized *sub-warps* such that each *sub-warp* processes either 32 rows of 1 element, 16 rows of 2 elements, 8 rows of 3 or 4 elements, 4 rows of 5, 6, 7 or 8 elements, and so on. These rows must correspond to the same level-set, since they are processed in parallel. The rationale of this strategy is to mitigate the waste of resources, and consequent performance downscale, experienced for rows with few nonzero elements. Moreover, the number of rows that are processed concurrently by active thread-blocks also increases. The procedure is described in Figure 10.

[3]Diagonal inverse-based methods [27, 28, 29, 30], and iterative approaches [31, 32, 33], are two alternative categories can be considered. These are interesting strategies but face important difficulties or are not general enough to be widely applicable.

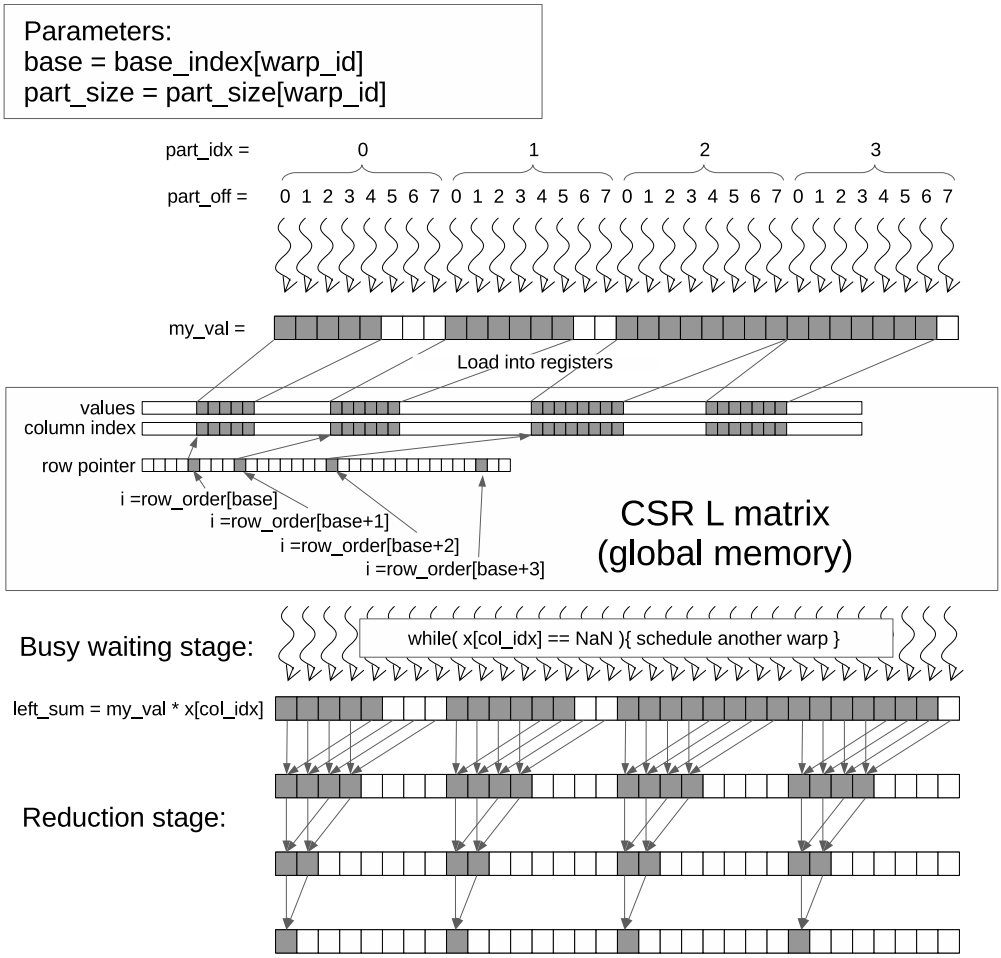

Figure 10: Life-cycle of a warp in the *multirow* variant of the algorithm. The analysis-phase produces three vectors: `row_order`, with the row indexes of the matrices partially ordered according to their level-set and number of nonzero element, `base_index`, which stores the first index in `row_order` to be processed by each warp, and `part_size` which stores the size of the partitions of each warp. With this information, the warp loads the corresponding coefficients from global memory, waits until the dependencies are ready, performs the multiplication, and then each partition performs a parallel summation to obtain the corresponding unknown.

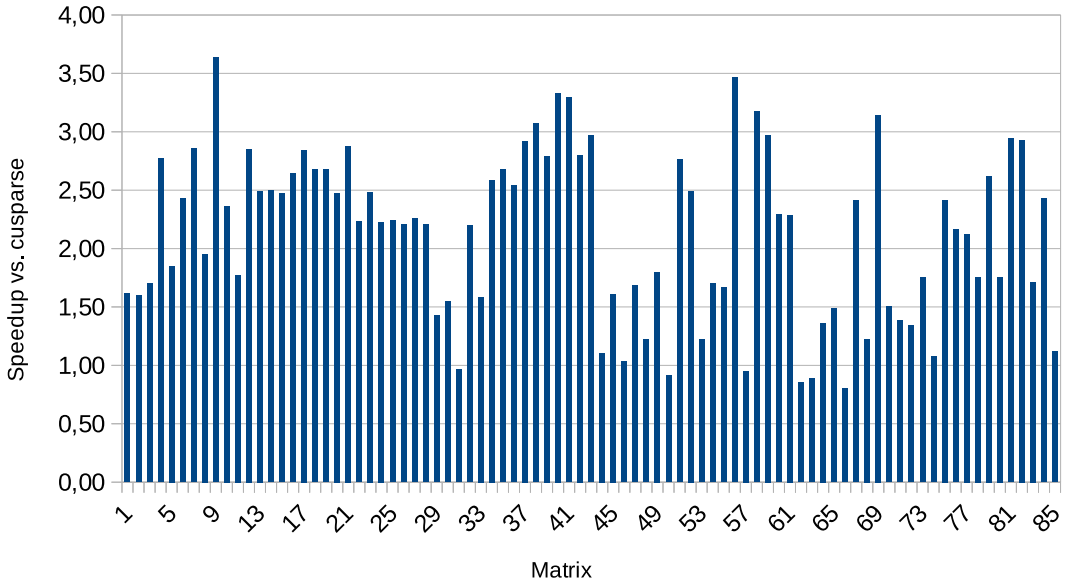

Figure 11: The picture shows the speedup obtained using synchronization free triangular solver for the preconditioner application relative to the preconditioner application using cuSparse for 86 matrices of the Suite Sparse matrix collection.

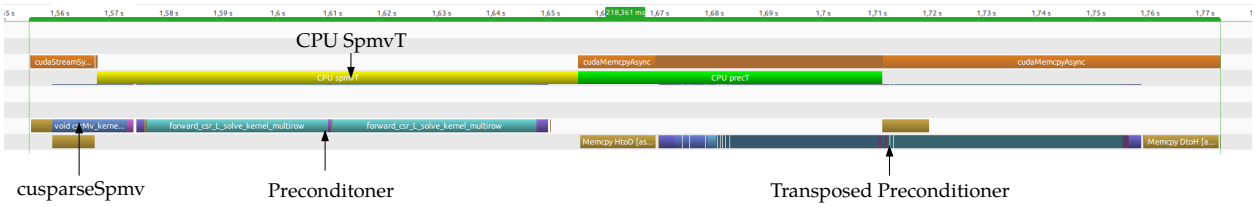

Figure 12: Time-line of the execution of the Adv_BICG_Hyb_SF version to solve the *cage15* test case in the experimental platform. Extracted with Nvidia Visual Profiler.

To assess the impact of our method in the context of ILUPACK we compare the runtimes of one application of the preconditioner using cuSparse solvers and one application of the preconditioner using the sf_Multirow solver. At this point, we remark that the analysis phase required by the sf_Multirow solver is, in general, much faster than that of cuSparse. However, we only consider the solution phase for this evaluation, since the analysis is performed only once during the construction of the preconditioner.

Figure 11 shows the speedup obtained by the preconditioner accelerated with the sf_Multirow solver with respect to the cuSparse counterpart. The experiment was performed on all the matrices of the SuiteSparse matrix collection with dimension n such that $100,000 < n < 1,000,000$ and nnz such that $1,000,000 < nnz < 10,000,000$. This yields 94 sparse matrices, but 8 of them failed during the construction of the preconditioner (with drop tolerance of 10^{-2} and $\kappa = 5$) because they were severely ill-conditioned.

The results for the remaining 86 matrices shows that the use of the sf_Multirow solver manages to accelerate the preconditioner in the vast majority of the tested instances. The greatest acceleration (of 3.62×) was obtained for matrix *cage12*, while the worst result (of 0.80×) was obtained for matrix *tmt_sym*. The harmonic mean of the speedups was 1.85×.

6.1 Hybrid variant of BiCG with enhanced task-parallelism, Adv_BICG_Hyb_SF

The Adv_BICG_Hyb_SF version replaces the triangular solver of the routine that applies the non-transposed preconditioner by our new synchronization-free routine. It employs the MKL library to perform the transposed SpMv on the CPU, and cuSparse to compute the application of the transposed preconditioner on the GPU.

Figure 12 presents the time-line corresponding to the execution of the Adv_BICG_Hyb_SF variant for the test case *cage15*. In this figure, the light-blue bars correspond to our synchronization-free routine. It can be observed that the delay in the execution of the transposed SpMv is significantly reduced, and that this operation overlaps almost completely with the execution of Set A in the GPU.

Figure 13 compares the speedups achieved by the baseline, Adv_BICG_Hyb_SF and Adv_BICG_2GPU

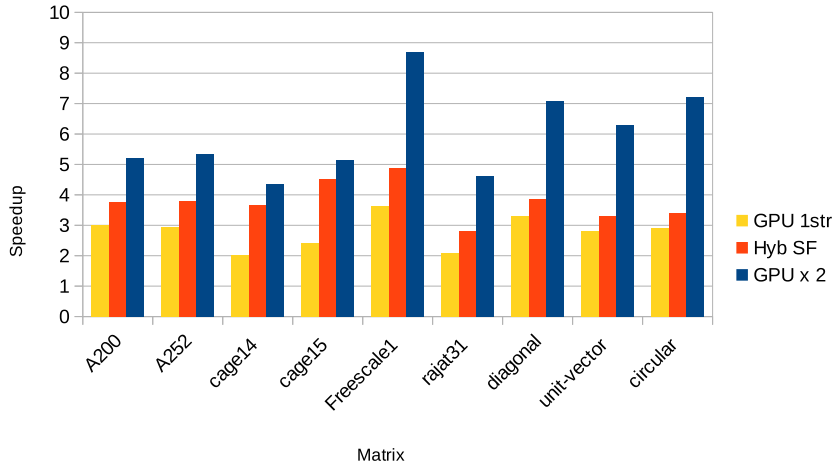

Figure 13: Acceleration of baseline, Adv_BICG_Hyb_SF and Adv_BICG_2GPU variants vs. multi-thread MKL in an Intel E5-2620v2 CPU (6 cores @ 2.10 GHz / 128 GB RAM) with a Tesla K40m GPU (2880 cores @ 0.75 GHz / 12GB RAM).

variants respect to the CPU version based on the multi-thread MKL, showing that Adv_BICG_Hyb_SF improves the speedups of the baseline version, with a performance that is comparable to that of Adv_BICG_2GPU in some cases.

7 Task+data parallel variants

Encouraged by the results obtained by the data-parallel variants, we explored the GPU acceleration of the task-parallel version of ILUPACK. We focused on the preconditioner application and developed variants for shared and distributed systems with GPUs.

We followed two different approaches. First, we entirely offloaded the leaf tasks of the preconditioner application phase to the GPU. Later, we used a threshold aiming to offload computations to the GPU only when there is enough work to take advantage of the accelerator.

Since the inner tasks in the tree correspond to separator sets with much fewer nodes than leaf sets, only leaf tasks perform an important amount of work. The leaf tasks are also independent from each other.

Each task has its own data structure, which contains the part of the multilevel preconditioner relevant to it and private CPU and GPU buffers. We associate each of these tasks with a different GPU stream, following a round-robin strategy to assign tasks to GPUs on nodes with multiple devices.

The computation on each node of the DAG is based on the data-parallel version presented in [38]. It proceeds as described in Section 2.1, with the difference that, in this case, the forward and backward substitution are separated and distributed among the levels of the task-tree. Therefore, entering or leaving the recursive step in Equation (5) sometimes implies moving to a different level in the task tree hierarchy. In these cases, the residual r_{k+1} has to be transferred to the GPU at the beginning of the forward substitution phase, and the intermediate result has to be retrieved back into the CPU buffers before entering the recursive step. Once the inner tasks compute the recursive steps, the backward substitution proceeds from top to bottom until finally reaching the leaf tasks again, where the z_{k+1} vector has to be transferred to the GPU. There the last steps of the calculation of the preconditioned residual $z_{k+1} := M^{-1}r_{k+1}$ are performed. Upon completion, the preconditioned residual z_{k+1} is retrieved back into the CPU.

The experiments showed that offloading the leaf tasks entirely to the accelerator can degrade the performance, as the SpTrSV of the smaller levels have little parallelism. In order to deal with this, we propose a threshold-based strategy that computes the algebraic levels in the GPU until certain granularity, and then moves the computation of the SpTrSV on the remaining levels to the CPU. This allows to perform each operation in the most convenient device, as even in smaller levels some parallelism can still be leveraged for the SpMv.

In this variant we determine the threshold value experimentally. Our on-going work aims to identify the best algorithmic threshold from a model that captures the algorithm's performance.

Figure 14 shows the results obtained on 4 nodes equipped with 2 GPUs each. It can be observed that the execution time of the all-CPU variant can be improved by a factor of up to 2× by assigning the tasks to the GPUs. The experiments also reveal scaling properties that suggest that it is possible to address large-scale

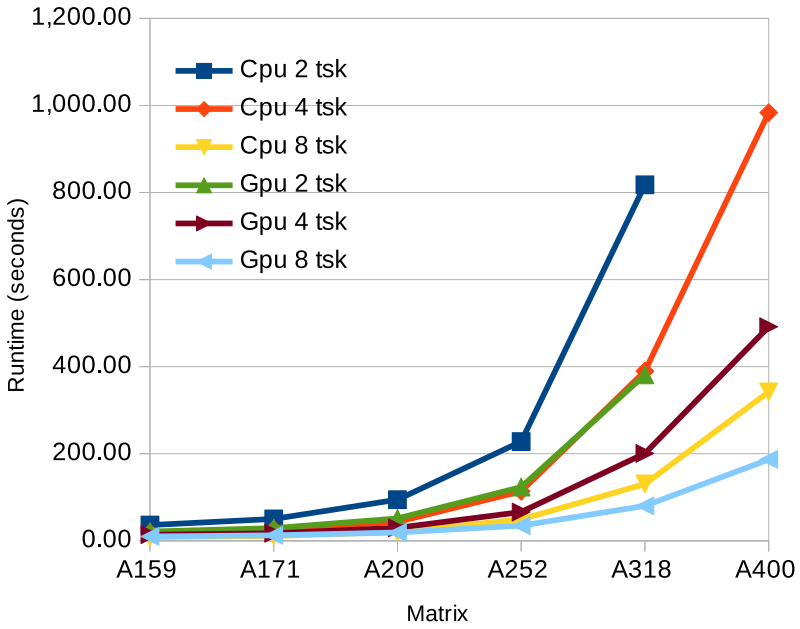

Figure 14: Runtimes for the GPU-aware distributed memory variant with 2, 4 and 8 leaf tasks (1 GPU per task) on 4 nodes with an Intel E5-2620v2 CPU (6 cores @ 2.10 GHz / 128 GB RAM) / 2×Tesla K40m (2880 cores @ 0.75 GHz / 12GB RAM)

problems efficiently.

8 Variant of BiCGStab for low-power devices

As an alternative for reducing the power consumption of large clusters, new systems that include unconventional devices have been proposed. In particular, it is now common to encounter energy efficient hardware such as GPUs and low-power ARM processors as part of hardware platforms intended for scientific computing.

In [39], we made a first step towards exploiting the use of energy efficient hardware to compute the ILUPACK solvers. Specifically, we adapted the SPD linear system solver of ILUPACK for the NVIDIA Jetson TX1 platform, based on an NVIDIA Tegra X1 processor, and evaluated its performance in problems that are unable to fully leverage the capabilities of high end GPUs.

Here we extend this work in two aspects. First, we consider our GPU-variant of the BiCGStab for ILUPACK [40], instead of the SPD solver, exploring new capabilities of the Jetson board for our adaptation. Second, we evaluate this proposal on a Jetson AGX Xavier, the latest and more powerful of this family of compute platforms.

The Jetson AGX Xavier integrates a 512-core Volta GPU (2× more cores than the TX1) with an 8-core ARM v8.2 64-bit CPU, 32GB of unified 256-Bit LPDDR4x memory (shared between the CPU and the integrated GPU) with a peak bandwidth of 137GB/s (+5× more bandwidth than the TX1), on a 100 × 87 mm board. The maximum power consumption can be configured to 10, 15 or 30W. We set it to 30W for the experiments in this work.

Our original implementation of the BiCGStab solver offloads the entire solver to the GPU. Hence, 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. The rest of the computations occur in the GPU. The matrix-vector multiplications and solution of triangular linear systems are performed using the cuSparse library, while vector operations rely on cuBlas and our own kernels.

This implementation is capable of running on the Jetson board without any modification. However, the unified memory of the Jetson provides the opportunity of executing each operation in the most convenient device without the overhead implied by the data transfers through the PCIe bus.

In this sense, Table 3 shows the harmonic mean of the relation between the performance of the CPU and the GPU for application of the first four levels of the multilevel preconditioner. The benchmark used is the same as in Section 6. The results clearly shows that the GPU is unable to deliver high performance in the case of higher-numbered levels (which present smaller matrices). This is primarily due to the performance of

Table 3: Harmonic mean of the ratio between the runtime taken by the CPU and the GPU to compute the first 4th levels of the multilevel non-symmetric preconditioner, for 86 matrices of the SuiteSparse matrix collection.

Level	1st	2nd	3rd	4th
T_{CPU}/T_{GPU}	2.07	0.57	0.26	0.12

the sparse triangular solver, which requires a considerable amount of work and parallelism to be able to hide the performance restrictions implied by the data dependencies. It is then reasonable to solve the systems corresponding to the first level of the preconditioner in the integrated GPU, and the rest in the ARM CPU.

Although the physical memory is shared between the CPU and the GPU, the Jetson presents four types of memory (Device Memory, Pageable Host Memory, Pinned Memory and Unified Memory), which differ on their caching and accessing policies[4]. Hence, it is important to select the right type of memory for each buffer in order to maximize the performance.

Since we are still interested on leveraging the GPU for the matrix-vector product and other operations of the BiCGStab solver, we keep the principal matrix on the Device Memory. We do the same for all the vectors implied in the solver but the residual and preconditioned residuals, which are the input and output vectors of the routine that performs application of the multilevel preconditioner. These vectors are kept in the Unified Memory space, since they will be shared between the CPU and the GPU. Some internal buffers of the preconditioner routine must be moved to the Unified Memory as well. Regarding the preconditioner, the first level is stored in the Device Memory, while the rest of the levels are stored in Pageable Host memory. Inside the routine that applies the preconditioner, the *cudaStreamAttachMemAsync* call is used accordingly before changing the device for the solution of a system, to aid the prefetching policy.

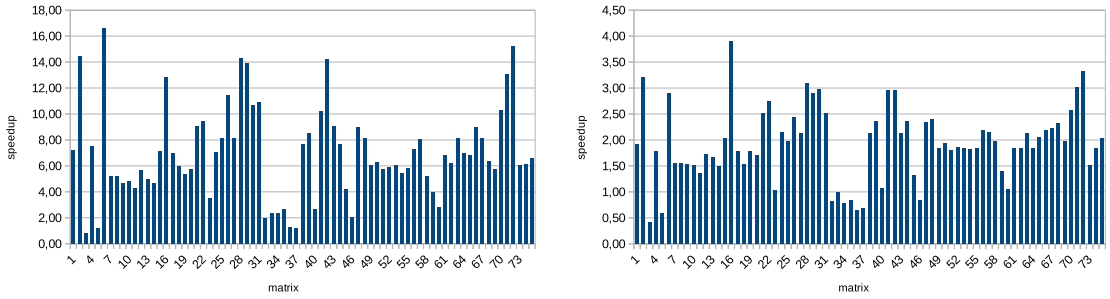

Figure 15: Comparison between the acceleration obtained for the baseline GPU variant of the BiCGStab on a discrete GTX 1080 Ti accelerator (left), and on the integrated GPU of the Jetson (right), with respect to the ARM v8 processor of the Jetson.

Figure 15 shows a comparison of the acceleration obtained by the baseline GPU variant of the BiCGStab on the integrated GPU of the Jetson and on a discrete GTX 1080 Ti accelerator, with respect to the ARM processor of the Jetson. It is evident that the acceleration obtained is significantly lower in the Jetson (3× lower on average). However, it must be observed that the problem is highly memory-bound, and the theoretical bandwidth of the GTX 1080 Ti is of 484 GB/s (3.5× higher than the theoretical bandwidth of the Jetson). In this sense, the results are completely aligned with the theory. Moreover, it is important to note that the total power dissipation of the GTX 1080 Ti is 250W (8× higher than the Jetson).

Regarding the selection of the most convenient memory buffer and device to perform each operation, Figure 16 shows that a sensible advantage can be obtained in some cases, specially for preconditioners that present many small levels. It is then interesting to develop a model that allows predicting which will be the most convenient device to perform each operation in this sort of platforms.

9 Concluding Remarks

The undeniable relevance of sparse linear systems in many areas of science and engineering, as well as the rapid scaling in the size of the problems we are witnessing nowadays, make necessary to adapt numerical software to correctly exploit modern computational platforms. In this sense, hardware accelerators such as GPUs have become an ubiquitous and powerful parallel architecture, and making an efficient use of these devices is of utmost importance.

[4]More information can be found in: https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html

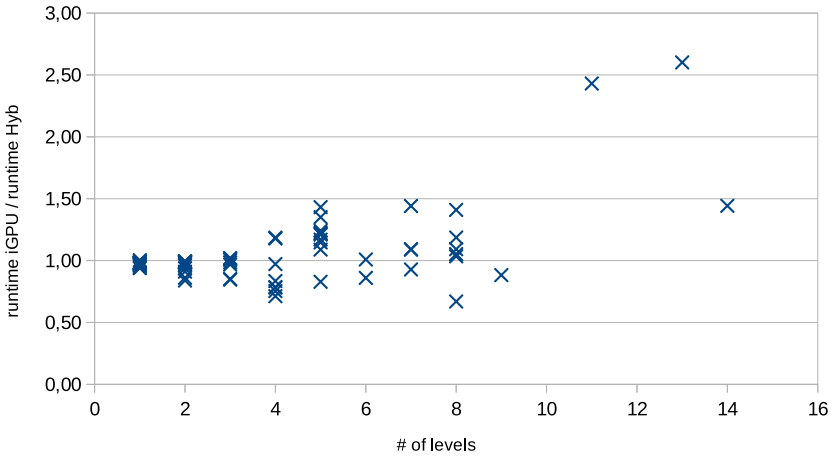

Figure 16: Ratio between the runtime to compute the BiCGStab taken by the baseline GPU variant and the Hybrid (Unified Memory) variant, according to the total number of levels of the preconditioner.

ILUPACK is a package that bundles several of the most widely used Krylov subspace solvers with a sophisticated multilevel "inverse-based" ILU preconditioner, that stands out due to its remarkable numerical properties.

Prior to this dissertation, there were no variants of ILUPACK capable of exploiting data-parallelism. Previous task-parallel implementations of ILUPACK [6, 7, 5] have shown good performance and scalability in many scenarios, and remain a valuable tool to this day, but also face some limitations.

In this thesis we have made a comprehensive study of this tool, in order to enable its efficient execution in platforms equipped with hardware accelerators. This study involves the development of several GPU-aware implementations of its preconditioner and most important solvers. In this sense, our new data-parallel implementations are able to improve the performance of the original preconditioner without significantly affecting its numerical properties.

In the context of accelerating the triangular linear systems, which are the computational bottleneck of the ILUPACK preconditioner, we proposed a synchronization-free algorithm for the solution of this operation [35], based in a recently proposed GPU computation paradigm. We also developed an algorithm for the level-set analysis of the sparse matrices based on the same strategy [36]. Our solvers present performance results that are, in general, better than those of cuSparse. In [41] we show how our new solvers are able to significantly accelerate the application of ILUPACK preconditioner, while on [42] we show how this paradigm favors the overlapping of the SpTrSV with other operations.

In addition to our data-parallel variants of the sequential version of ILUPACK, we studied the exploitation of GPU-enabled platforms for the two task-parallel implementations available at the moment. Specifically, we enabled GPU computations for the shared-memory and distributed-memory implementations of ILUPACK.

During the process of the thesis we have detected some directions in which this work can be extended. First, the computation of ILUPACK preconditioner is a complex and time consuming process and its parallel execution has not been studied for non-SPD problems. The main reasons for this are that the computation of the preconditioner is performed only once for each matrix, and it is therefore more interesting to accelerate its application, which is likely to be performed even hundreds of times for each matrix. However, there are use cases in which the computation time of the preconditioner completely overshadows the reduction in the iteration count of the solvers. The study of this computation process and its parallelization is an interesting line of future work.

Another interesting line of future work is the solution of sparse triangular linear systems. Our recent advances on a synchronization-free approach for this operation motivate the future development of GPU synchronization-free *LDU*-system solvers specially designed for ILUPACK data types and characteristic nonzero patterns.

In the context of our acceleration of the task-parallel variants of the ILUPACK preconditioner, we are interested on the development of an automatic mechanism to determine the most convenient device for each operation.

Finally, we plan to study the exploitation of the new features presented by the latest GPU architectures, such as Tensor Cores or the group synchronization mechanisms introduced in CUDA 9.0. These developments are too recent to be covered in this thesis, so they will be addressed as part of future work.

Acknowledgments

Special thanks to the advisors of the thesis, Dr. Pablo Ezzatti (Universidad de la República, Uruguay) and Dr. Enrique Quintana-Ortí (Universitat Politècnica de València, Spain). I would also like to acknowledge the support from the Programa de Desarrollo de las Ciencias Básicas (PEDECIBA, Uruguay).

References

[1] P. Blanchard and E. Brüning, *Mathematical Methods in Physics: Distributions, Hilbert Space Operators, Variational Methods, and Applications in Quantum Physics.* Birkhäuser, 2015, vol. 69.

[2] A. C. Chiang, *Fundamental methods of mathematical economics.* Aukland (New Zealand) McGraw-Hill, 1984.

[3] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 4th ed. Baltimore: The Johns Hopkins Univ. Press, 2012.

[4] M. Bollhöfer and Y. Saad, "Multilevel preconditioners constructed from inverse-based ILUs," *SIAM Journal on Scientific Computing*, vol. 27, no. 5, pp. 1627–1650, 2006. [Online]. Available: https://doi.org/10.1137/040608374

[5] 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. [Online]. Available: https://doi.org/10.1016/j.parco.2015.12.004

[6] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819110001432

[7] J. I. Aliaga, M. Bollhöfer, A. F. Martín, and E. S. Quintana-Ortí, "Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors," in *Applied Parallel and Scientific Computing, LNCS*. Springer, Berlin, Heidelberg, 2012, vol. 7133, pp. 162–172. [Online]. Available: https://doi.org/10.1007/978-3-642-28151-8_16

[8] 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*, 2016, pp. 121–133. [Online]. Available: https://doi.org/10.1007/978-3-319-58943-5_10

[9] Y. Saad, *Iterative methods for sparse linear systems*, 2nd ed. Society for Industrial and Applied Mathematics, 2003. [Online]. Available: http://gen.lib.rus.ec/book/index.php?md5=FF966E070560946D91C44B50C1D92492

[10] L. Buatois, G. Caumon, and B. Lévy, "Concurrent number cruncher: An efficient sparse linear solver on the gpu," in *High Performance Computing and Communications*, R. Perrott, B. M. Chapman, J. Subhlok, R. F. de Mello, and L. T. Yang, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 358–371. [Online]. Available: htpps://www.doi.org/10.1007/978-3-540-75444-2_37

[11] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 18:1–18:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654078

[12] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2012, version 0.3.0. [Online]. Available: http://cusp-library.googlecode.com

[13] M. Ament, G. Knittel, D. Weiskopf, and W. Straßer, "A parallel preconditioned conjugate gradient solver for the poisson problem on a multi-gpu platform," in *PDP '10: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Networkbased Processing*, 2010, pp. 583–592. [Online]. Available: https://doi.org/10.1109/PDP.2010.51

[14] H. Sudan, H. Klie, R. Li, and Y. Saad, "High performance manycore solvers for reservoir simulation," in *12th European conference on the mathematics of oil recovery*, 2010.

[15] R. Gupta, M. B. van Gijzen, and C. Vuik, "3d bubbly flow simulation on the gpu - iterative solution of a linear system using sub-domain and level-set deflation," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013, pp. 359–366.

[16] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," NVIDIA corporation, NVIDIA white paper, 2011.

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

[18] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013. [Online]. Available: http://dx.doi.org/10.1007/s11227-012-0825-3

[19] D. Lukarski, H. Anzt, S. Tomov, and J. Dongarra, "Multi-elimination ilu preconditioners on gpus," in *IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, 2014.

[20] K. He, S. X. Tan, H. Zhao, X.-X. Liu, H. Wang, and G. Shi, "Parallel GMRES solver for fast analysis of large linear dynamic systems on GPU platforms," *Integration, the VLSI Journal*, vol. 52, pp. 10 – 22, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016792601500084X

[21] C. D. Team, *Cusparse Library v9.0*, NVIDIA Corporation, 2018.

[22] C. Lanczos, "Solution of systems of linear equations by minimized iterations," *J. Res. Nat. Bur. Standards*, vol. 49, no. 1, pp. 33–53, 1952.

[23] P. Carvalho, R. A. Q. Cruz, L. M. de A. Drummond, C. Bentes, E. Clua, E. Cataldo, and L. A. J. Marzulo, "Kernel concurrency opportunities based on GPU benchmarks characterization," *Cluster Computing*, vol. 23, no. 1, pp. 177–188, 2020. [Online]. Available: https://doi.org/10.1007/s10586-018-02901-1

[24] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *International Journal of High Speed Computing*, vol. 1, no. 01, pp. 73–95, 1989.

[25] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011*, vol. 1, 2011.

[26] D. Erguiz, E. Dufrechou, and P. Ezzatti, "Assessing sparse triangular linear system solvers on gpus," in *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, Oct 2017, pp. 37–42. [Online]. Available: https://10.1109/SBAC-PADW.2017.15

[27] F. Alvarado, D. Yu, and R. Betancourt, "Ordering schemes for partitioned sparse inverses," in *SIAM Symposium on Sparse Matrices, Salishan Lodge, Gleneden Beach, Oregon*, 1989.

[28] F. Alvarado and R. Schreiber, "Fast parallel solution of sparse triangular systems," in *13th IMACS World Congress on Computation and Applied Mathematics, Dublin*, 1991.

[29] A. Pothen and F. L. Alvarado, "A fast reordering algorithm for parallel sparse triangular solution," *SIAM journal on scientific and statistical computing*, vol. 13, no. 2, pp. 645–653, 1992.

[30] F. L. Alvarado and R. Schreiber, "Optimal parallel solution of sparse triangular systems," *SIAM Journal on Scientific Computing*, vol. 14, no. 2, pp. 446–460, 1993. [Online]. Available: https://doi.org/10.1137/0914027

[31] H. Anzt, E. Chow, and J. Dongarra, *Iterative Sparse Triangular Solves for Preconditioning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 650–661. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48096-0_50

[32] A. C. Van Duin, "Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices," *SIAM journal on matrix analysis and applications*, vol. 20, no. 4, pp. 987–1006, 1999. [Online]. Available: https://doi.org/10.1137/S0895479897317788

[33] H. A. van der Vorst, "A vectorizable variant of some iccg methods," *SIAM Journal on Scientific and Statistical Computing*, vol. 3, no. 3, pp. 350–356, 1982. [Online]. Available: https://doi.org/10.1137/0903021

[34] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter, "Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, 2017. [Online]. Available: https://doi.org/10.1002/cpe.4244

[35] E. Dufrechou and P. Ezzatti, "Solving sparse triangular linear systems in modern gpus: A synchronization-free algorithm," in *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*, 2018, pp. 196–203. [Online]. Available: https://doi.org/10.1109/PDP2018.2018.00034

[36] E. Dufrechou and P. Ezzatti, "A new GPU algorithm to compute a level set-based analysis for the parallel solution of sparse triangular systems," in *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*, 2018, pp. 920–929. [Online]. Available: https://doi.org/10.1109/IPDPS.2018.00101

[37] E. Dufrechou and P. Ezzatti, "Using analysis information in the synchronization-free GPU solution of sparse triangular systems," *Concurr. Comput. Pract. Exp.*, vol. 32, no. 10, 2020. [Online]. Available: https://doi.org/10.1002/cpe.5499

[38] J. I. Aliaga, M. Bollhöfer, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, "Leveraging data-parallelism in ILUPACK using graphics processors," in *IEEE 13th International Symposium on Parallel and Distributed Computing, ISPDC 2014, Marseille, France, June 24-27, 2014*, 2014, pp. 119–126. [Online]. Available: https://doi.org/10.1109/ISPDC.2014.19

[39] J. I. Aliaga, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, "Evaluating the NVIDIA tegra processor as a low-power alternative for sparse GPU computations," in *High Performance Computing - 4th Latin American Conference, CARLA 2017, Buenos Aires, Argentina, and Colonia del Sacramento, Uruguay, September 20-22, 2017, Revised Selected Papers*, ser. Communications in Computer and Information Science, E. E. Mocskos and S. Nesmachnow, Eds., vol. 796. Springer, 2017, pp. 111–122. [Online]. Available: https://doi.org/10.1007/978-3-319-73353-1_8

[40] J. I. Aliaga, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, "Extending ILUPACK with a GPU version of the bicgstab method," in *XLIV Latin American Computer Conference, CLEI 2018, São Paulo, Brazil, October 1-5, 2018*. IEEE, 2018, pp. 728–734. [Online]. Available: https://doi.org/10.1109/CLEI.2018.00092

[41] J. I. Aliaga, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, "An efficient gpu version of the preconditioned gmres method," *The Journal of Supercomputing*, Oct 2018. [Online]. Available: https://doi.org/10.1007/s11227-018-2658-1

[42] J. I. Aliaga, E. Dufrechou, P. Ezzatti, and E. S. Quintana-Ortí, "Accelerating the task/data-parallel version of ilupack's bicg in multi-cpu/gpu configurations," *Parallel Computing*, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819118301777