Accelerating the task/data-parallel version of ILUPACK's BiCG in multi-CPU/GPU configurations

José I. Aliaga[a], Ernesto Dufrechou[b], Pablo Ezzatti[b], Enrique S. Quintana-Ortí[a]

[a]*Dpto. de Ingeniería y Ciencia de Computadores, Universitat Jaume I, Castellón, Spain.*
[b]*Instituto de Computación, Universidad de la República, Montevideo, Uruguay.*

Abstract

ILUPACK is a valuable tool for the solution of sparse linear systems via iterative Krylov subspace-based methods. Its relevance for the solution of real problems has motivated several efforts to enhance its performance on parallel machines. In this work we focus on exploiting the task-level parallelism derived from the structure of the BiCG method, in addition to the data-level parallelism of the internal matrix computations, with the goal of boosting the performance of a GPU (graphics processing unit) implementation of this solver. First, we revisit the use of dual-GPU systems to execute independent stages of the BiCG concurrently on both accelerators, while leveraging the extra memory space to improve the data access patterns. In addition, we extend our ideas to compute the BiCG method efficiently in multicore platforms with a single GPU. In this line, we study the possibilities offered by hybrid CPU-GPU computations, as well as a novel synchronization-free sparse triangular linear solver. The experimental results with the new solvers show important acceleration factors with respect to the previous data-parallel CPU and GPU versions.

Key words: Sparse linear systems, iterative Krylov-subspace methods, data parallelism, ILUPACK preconditioner, graphics processing units (GPUs)

1. Introduction

Solving large-scale sparse linear systems is a crucial task in a large number of engineering and scientific problems, such as the discretization of partial differential equations (PDEs) for quantum physics and circuit simulation [1]. In addition, this operation is frequently a computational bottleneck, demanding 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 is the use of preconditioned Krylov subspace-based methods. In particular, approximate matrix factorizations are a versatile and powerful solution to improve the convergence rate of the iterative solver [1], although their lack of robustness limits their applicability in some highly ill-conditioned scenarios. Among the efforts to solve these problems, ILUPACK[1] 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 for general as well as Hermitian positive definite/indefinite linear systems [3].

Unfortunately, the favorable numerical properties of ILUPACK's preconditioner in the context of an iterative Krylov subspace solvers are overshadowed by its expensive construction and application procedures. This high computational cost motivated the development 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 [4] we also introduced a version of ILUPACK's CG method, for s.p.d. systems, that exploits the data(-level) parallelism intrinsic to the most expensive kernels, off-loading their execution to a graphics processing unit (GPU). In [7], we adapted ILUPACK's solvers for general and symmetric indefinite linear systems to provide data-parallel implementations of GMRES, BiCG and SQMR [1] on GPUs.

This paper extends [8], where we exploited the intrinsic task parallelism of the BiCG algorithm to offload each of its two independent sequences to a different GPU, effectively avoiding the use of inefficient transposed operators. While the approach of our previous paper was straightforward given the presence of two accelerators, this extension deals with the exploitation of BiCG's task parallelism in platforms equipped with a multicore processor and one GPU. Our main objective is to determine the most efficient strategy to compute the BiCG method with ILUPACK preconditioner on this sort of hardware. With this aim, we explore the use of GPU streams and different strategies of concurrent CPU and GPU computations. Our main premise is that, in single-GPU contexts, the use of the multicore CPU could partially compensate the absence of a second GPU. The experimental analysis compares the performance of the different variants of the BiCG using a set of real problems extracted from the Suite Sparse collection of sparse matrices [9], and test problems of scalable size derived from the discretization of partial differential equations (PDEs), showing a notorious improvement in the concurrent computation with respect to the

Email addresses: `aliaga@icc.uji.es` (José I. Aliaga), `edufrechou@fing.edu.uy` (Ernesto Dufrechou), `pezzatti@fing.edu.uy` (Pablo Ezzatti), `quintana@icc.uji.es` (Enrique S. Quintana-Ortí)

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

original version.

The rest of the paper is structured as follows. In Section 2 we review the iterative solvers integrated into ILUPACK and the use of GPUs to accelerate them. Next, in Section 3, we describe our proposals, starting with a revision and extension of the ideas that support the exploitation of task parallelism in the BiCG method, and later, we describe the different techniques applied to harness the task and data(-level) parallelism of the BiCG method, both in dual- and single-GPU hardware platforms. Section 4 corresponds to the numerical evaluation of our proposal, while Section 5 summarizes the related work. Finally, a few remarks and some lines of future work close the paper in Section 6.

2. High Performance 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 includes software to calculate an inverse-based multilevel ILU preconditioner M, of dimension $n \times n$, which can be leveraged to accelerate the convergence of Krylov subspace-based iterative solvers. The package includes numerical methods for different matrix types, precisions and arithmetic, and has proved to be highly effective at reducing the number of iterations necessary for these methods to converge to an acceptable solution for many sparse linear systems [3, 10, 11].

The application of ILUPACK preconditioner is a recursive procedure that, for each level, requires two SpMV operations, solving two linear systems with coefficient matrix of the form LDU, and a few vector kernels (the reader is referred to [3] for a detailed explanation of both the computation and the application of the preconditioner). When using an iterative solver enhanced with the ILUPACK preconditioner, this procedure, which occurs (at least) once per iteration of the solver, is usually the most demanding task from the computational point of view. This has motivated the inclusion of parallel computing techniques in our previous work. In [7], we relied on the nVidia cuSparse library to perform the sparse triangular system solves (SpTrSV) and the SpMV in the GPU, since this library provides efficient implementations of the necessary kernels and supports the most common sparse matrix formats. The vector kernels, which are mainly diagonal matrix scalings and reorderings that gain mild importance only for highly sparse matrices of large dimension, were accelerated in our codes via *ad-hoc* Cuda kernels. 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 stored in the GPU, and this 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.

Operation	kernel
Initialize $x_0, r_0, q_0, p_0, \ldots, s_0, \rho_0, \tau_0; k := 0$	
$A \to M$	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
$\tau_{k+1} := \parallel r_k \parallel_2$	dot product
$k := k + 1$	
end while	

Figure 1: Algorithmic formulation of the preconditioned BiCG method.

3. Task-data parallel BiCG in ILUPACK

While our previous work introduced optimizations to the preconditioner aimed to leverage the data-level parallelism in all of the Krylov subspace solvers bundled with ILUPACK, the focus of this article is in exploiting the task-level parallelism offered by the BiCG method.

The BiCG method was first derived by Lanczos [12] 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 1, 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 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 2, 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.

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	Set A
$x_k := x_k + \alpha_k p_k$	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.	Set B
synchronization		
$s_{k+1} := s_k - \alpha_k z_k$	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	
$\tau_{k+1} :=\parallel r_k \parallel_2$	dot product	
$k := k + 1$		
end while		

Figure 2: 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.

In the remainder of the section we describe the different strategies designed to take advantage of this task-level parallelism. Concretely, we first revisit the proposal of a task- and data-level parallel implementation of the BiCG method in ILUPACK tailored for servers equipped with two GPUs, presented in [8]. Later, we describe several variants in order to exploit the task-level parallelism of BiCG in single-GPU servers, compensating the lack of a second GPU with an efficient use of the hardware resources.

3.1. *Variant for dual-GPU systems,* GPU_×2

In systems equipped with two discrete graphics accelerators, the arrangement of the operations of the BiCG proposed in Figure 2 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 previous implementations, these transposed SpMVs were computed by calling the kernel in cuSparse on the original non-transposed matrices setting a parameter so that the library performs the transposed operation. The execution times reached using the accelerated data-parallel version BiCG from [7] show that the calls to SpMV that operate with A^T are, on average, 2–3× slower than those working with the non-transposed matrix, 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.

In summary, when the hardware platform contains at least two devices, we can exploit the extended computational power and memory capacity obtained from the second GPU to overlap the execution of the two recurrences of the BiCG, using the non-transposed version of the SpMV at no additional memory cost. The initial data transferences to both devices are performed asynchronously overlapped with the construction of the preconditioner. See [8] for more details.

3.2. *Variant with two streams in a single GPU,* Cusp_2str

The results obtained in [7] indicate that, when there is only one GPU available in the server, using cuSparse to accelerate the most data-parallel stages of BiCG is, in general, convenient. However, the variant in [7] did not take advantage of the coarse-grain parallelism of the BiCG method, and thus there is room for improvement.

Perhaps the most straightforward strategy consists in exploiting the concurrent GPU execution offered by *CUDA Streams*. Therefore, the first accelerated version for single-GPU platforms that we propose, performs the operations that corresponded to Set A in Figure 2 in the first stream, while the operations corresponding to Set B are performed in the second stream. No operations are left to the default stream, and we use cuBlas *device pointer mode* [13] to avoid unnecessary synchronizations due to scalar parameters being transferred to and retrieved from the GPU.

Regarding the computation of $A^T v$ on the GPU, we use the transposed variant of the SpMV routine of cuSparse. An alternative to avoid the use of the transposed SpMV operation of cuSparse, is to store the transposed operands in the accelerator, and operate with them directly. This has the obvious effect of almost duplicating the memory footprint of the method and reducing the size of the problems that can be solved, and thus is not a satisfactory solution.

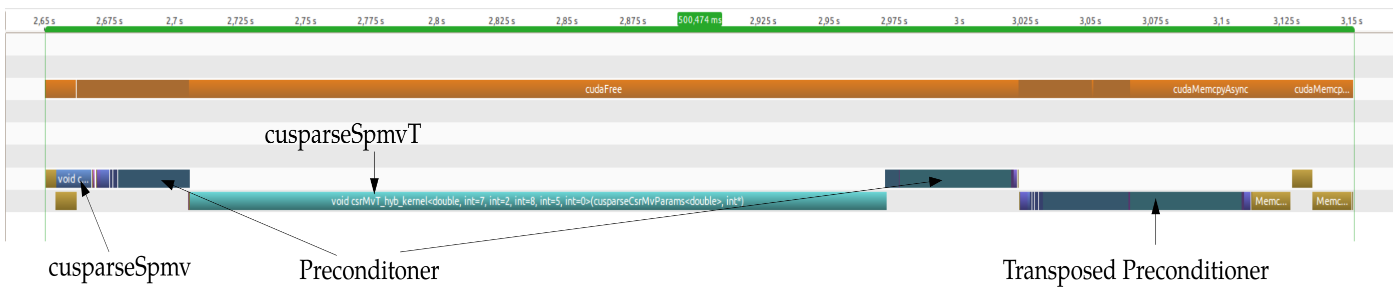

Figure 3: Timeline of the execution of the Cusp_2str version to solve the *cage15* test case in the experimental platform. Extracted with nVidiaVisual Profiler.

As the resources of the GPU must be shared between all streams, the overlapping of kernels in different streams can be modest in those cases where one of the kernels fully utilizes the accelerator. In such scenario, the performance of the solver will be almost identical to the variant that uses only one stream. This can be appreciated in the timeline of Figure 3, corresponding to an execution of this variant for test case *cage15* on our experimental platform, traced using the nVidiaVisual Profiler (see Section 4 for details on the experimental setup).

To summarize, Cusp_2str computes the operations corresponding to the Set A and Set B blocks in Figure 2 in one device, assigning one GPU stream to each block. The GPU computation of the SpMVs and sparse triangular linear systems are performed using cuSparse library, the vector operations of BiCG are computed using cuBlas, and the remaining vector operations inside the application of the preconditioner are implemented using *ad-hoc* GPU kernels.

3.3. Hybrid CPU-GPU variant, Hyb_Cusp

The timeline in Figure 3 reveals that there is almost no overlapping between the operations in Set A and Set B. This indicates that the kernels of the cuSparse library tend to occupy the multiprocessors of the GPU so that there is no room to execute two such kernels in parallel. Therefore, it is interesting to study the use of the multicore CPU to achieve the overlapping of the two sequences.

Given the important difference between the performance on the GPU of the transposed and non-transposed SpMV routines, one possibility is to leverage the multicore CPU to perform the transposed SpMV. We still use cuSparse to compute the application of the transposed preconditioner, as our previous results indicate that we can still obtain an important acceleration for this routine.

The proposed strategy aims to overlap an important part of the operations corresponding to Set A, which execute on the GPU, with the transposed SpMV on the CPU.

Additionally, the use of different GPU-streams for Set A and Set B is maintained so that the computations of one set can be overlapped with the communications of the other.

In summary, Hyb_Cusp 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, as in Cusp_2str.

In Figure 4 we present the timeline for the Hyb_Cusp version, extracted with nVidiaVisual Profiler using the same configuration parameters as in Figure 3. In the timeline, four horizontal lines can be clearly distinguished, each one divided into several bars that correspond 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.

At least two aspects are worth noting. First, the figure shows that the yellow bar of the transposed SpMV considerably overlaps with the blue bar which corresponds to the application of the preconditioner in the GPU. Second, 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 [14], and is described in [15]. In a broad sense, this technique conducts an analysis of the triangular sparse matrix to determine sets of independent rows called *levels*. The triangular solver then 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 sparsity 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 [16].

Figure 4 illustrates how the launching of these kernels delays the start of the multi-threaded transposed SpMV on the CPU.

3.4. Hybrid variant with enhanced task-parallelism, Hyb_SyncFree

The situation described in the previous section suggests that, in order to enhance the concurrent execution between the CPU and the GPU, it is important to reduce the overhead derived from the many kernel launches implied by the synchronization

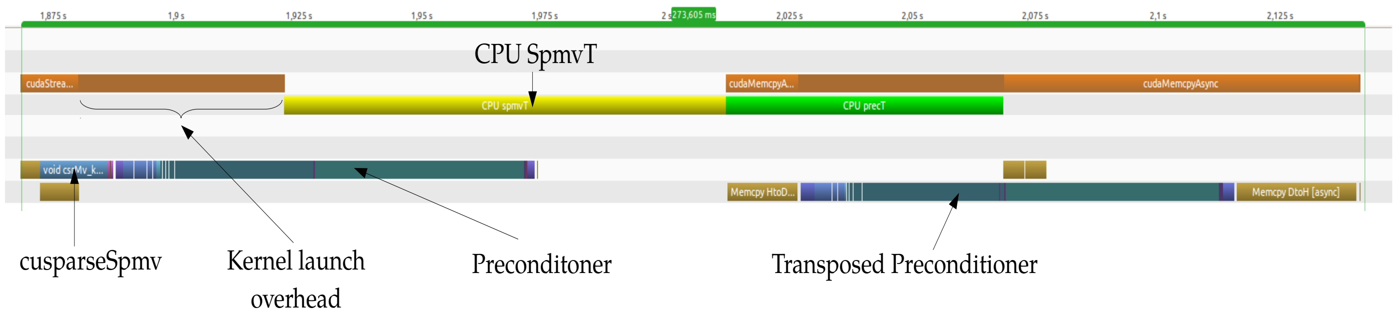

Figure 4: Timeline of the execution of the Hyb_Cusp version to solve the *cage15* test case in the experimental platform. Extracted with nVidiaVisual Profiler.

between levels in the cuSparse `cusparseDcsrsv_solve` routine.

Recent research has dealt with this problem by introducing a self-scheduled strategy that effectively avoids the synchronization with the CPU [17]. In [18] we followed these ideas to develop a synchronization-free GPU routine to solve triangular linear systems for matrices in the CSR format. Although this technique does not always improve the runtime attained by cuSparse for the triangular systems involved in ILUPACK, it can favour the overlapping of operations between CPU and GPU.

As the cost of launching the kernels involved in this routine is completely negligible, replacing the triangular solver of cuSparse by our synchronization-free routine in the non-transposed stream can enable a better overlapping of these operations with the transposed SpMV in the CPU.

In a nutshell, the Hyb_SyncFree 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 5 presents the timeline corresponding to the execution of the Hyb_SyncFree 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.

4. Experimental Evaluation

In this section we perform the analysis of the experimental results obtained from the execution of the different variants discussed in Section 3. In addition to the four variants presented in that section (GPU_×2, Cusp_2str, Hyb_Cusp and Hyb_SyncFree), we include other two reference versions in the experimental evaluation:

- Cpu_mkl performs all the computations on the multicore processor. The two SpMV of the BiCG are performed using the multi-threaded variant of the MKL library while the triangular solvers are computed with an optimized sequential code. This variant does not exploit task-level parallelism.
- Cusp_1str computes the main operation of BiCG method in the GPU employing only one stream and the cuSparse library. This version is the same as that presented in [7].

Before analyzing the results we first describe the hardware and software platform employed in the experiments, as well as the test cases we used in the evaluation.

4.1. 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 128 GB of DDR3 RAM memory. The platform also contained two NVIDIA “Kepler” K40m GPUs, each with 2,880 CUDA cores and 12 GB of GDDR5 RAM.

The CPU codes were compiled with gcc 4.8.5 using the `-O3` flag, we used version 2017 (update 3) of the Intel MKL library. The GPU compiler and the cuSparse library were those in version 9.2 of the CUDA Toolkit.

4.2. Test Cases

Laplace. 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 benchmark s.p.d. linear systems of order $n \approx$ 8M and 16M, respectively; see Table 1 for details.

SSMC. We selected a variety of large-scale matrices from the SSMC benchmark collection[2]; see Table 1.

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:

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

[2]Suite Sparse Matrix Collection: at https://sparse.tamu.edu/

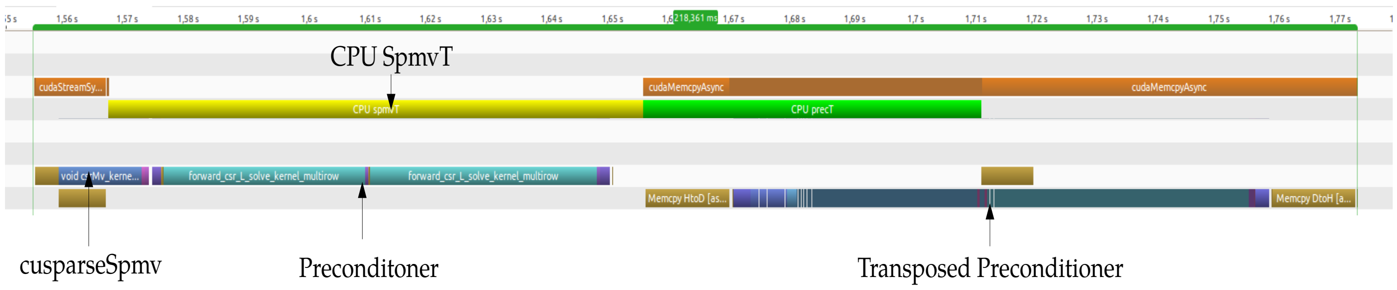

Figure 5: Timeline of the execution of the Hyb_SyncFree version to solve the *cage15* test case in the experimental platform. Extracted with nVidiaVisual Profiler.

Table 1: Main features of the matrices used in the experiments: dimension n and number of nonzeros nnz.

Group	Matrix	n	nnz	nnz/n
Laplace	A200	8,000,000	31,880,000	3.99
	A252	16,003,008	63,821,520	3.99
SSMC	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

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 1. For the diffusion part $-\varepsilon\Delta u$ we use a seven-point-stencil. The convective part $b * u$ is discretized using up-wind differences.

4.3. Experimental results

The results obtained for the six variants are displayed in Tables 2 and 3. We focus the analysis on the data corresponding to the accelerated part of the BiCG because the variants differ from each other on these computations. The runtimes of the remaining of the stages are similar.

The runtimes in this section are computed as the wall-time elapsed between two points in the program, and thus include the time taken by all the CPU and GPU activity. In the tables, Total is referred to the time-to-solution of the BiCG and includes the time taken by the transference of the right-hand side vector and the solution vector to and from the GPU, respectively, but does not include the time taken by the transference of the coefficient matrix A and the preconditioner, as these transferences are performed asynchronously during the construction of the preconditioner.

First, we can observe that the Cusp_1str variant improves the MKL-based version, offering speed-ups between 2.03 and 3.63×. Additionally, these results are aligned with our previous work, where the highest values of acceleration are attained for the solution of the most demanding test cases.

Moreover, the results show that the GPU_×2 variant produces a sensible reduction of the execution time for all cases. Naturally, the acceleration values tend to be higher for the cases

Table 2: Number of iterations and runtime (in seconds) obtained by the GPU_×2 variant. The times are disaggregated in the time corresponding to the accelerated part of the method (dominated by the SpMV and the application of the preconditioner) and the part corresponding to other operations. The results of the Cpu_mkl and Cusp_1str variants are displayed for comparison, and we compute the speedup of each version respect to Cpu_mkl.

Matrix	Routine	# It.	Accel. part	Other Ops.	Total	Speed vs. Cpu_mkl
A200	Cpu_mkl	12	6.03	0.73	6.88	
	Cusp_1str	12	1.46	0.71	2.28	3.02
	GPU_×2	12	0.58	0.62	1.32	5.20
A252	Cpu_mkl	12	12.13	1.34	13.69	
	Cusp_1str	12	3.00	1.42	4.67	2.93
	GPU_×2	12	1.16	1.17	2.57	5.33
cage14	Cpu_mkl	14	2.10	0.18	2.35	
	Cusp_1str	14	0.96	0.14	1.16	2.03
	GPU_×2	14	0.34	0.14	0.54	4.34
Freescale1	Cpu_mkl	442	150.91	8.23	159.21	
	Cusp_1str	442	36.70	7.10	43.88	3.63
	GPU_×2	442	11.06	7.15	18.29	8.71
rajat31	Cpu_mkl	12	3.42	0.42	3.93	
	Cusp_1str	12	1.35	0.45	1.87	2.10
	GPU_×2	12	0.41	0.37	0.85	4.62
cage15	Cpu_mkl	16	9.78	0.68	10.69	
	Cusp_1str	16	3.72	0.49	4.42	2.42
	GPU_×2	16	1.34	0.50	2.08	5.14
diagonal	Cpu_mkl	170	294.13	7.56	301.86	
	Cusp_1str	170	84.35	6.26	91.77	3.29
	GPU_×2	170	35.75	6.63	42.53	7.10
circular	Cpu_mkl	158	242.45	6.35	248.95	
	Cusp_1str	158	79.35	5.88	85.38	2.92
	GPU_×2	158	28.18	6.24	34.57	7.20
unit-vector	Cpu_mkl	170	260.81	6.83	267.81	
	Cusp_1str	170	89.37	6.31	95.83	2.79
	GPU_×2	170	36.01	6.44	42.60	6.29

Table 3: Number of iterations and runtime (in seconds) obtained by our single-GPU variants. The times are disaggregated in the time corresponding to the accelerated part of the method (dominated by the SpMV and the application of the preconditioner) and the part corresponding to other operations. We compute the speedup of each version respect to Cpu_mkl.

Matrix	Routine	# It.	Accel. part	Other Ops.	Total	Speed vs. Cpu_mkl
A200	Cusp_2str	12	1.40	0.64	2.19	3.14
	Hyb_Cusp	12	1.12	0.64	1.88	3.67
	Hyb_SyncFree	12	1.07	0.64	1.82	3.77
A252	Cusp_2str	12	2.94	1.41	4.60	2.98
	Hyb_Cusp	12	2.13	1.24	3.59	3.81
	Hyb_SyncFree	12	2.12	1.26	3.61	3.79
cage14	Cusp_2str	14	0.94	0.13	1.16	2.03
	Hyb_Cusp	14	0.50	0.14	0.70	3.35
	Hyb_SyncFree	14	0.44	0.13	0.64	3.67
Freescale1	Cusp_2str	442	35.42	6.74	42.23	3.77
	Hyb_Cusp	442	28.70	8.95	37.73	4.22
	Hyb_SyncFree	442	25.75	6.80	32.62	4.88
rajat31	Cusp_2str	12	1.34	0.49	1.92	2.05
	Hyb_Cusp	12	1.02	0.49	1.60	2.46
	Hyb_SyncFree	12	0.96	0.37	1.40	2.80
cage15	Cusp_2str	16	3.65	0.53	4.40	2.43
	Hyb_Cusp	16	2.07	0.53	2.86	3.74
	Hyb_SyncFree	16	1.65	0.50	2.36	4.53
diagonal	Cusp_2str	170	86.35	6.80	93.30	3.24
	Hyb_Cusp	170	78.75	8.62	87.51	3.45
	Hyb_SyncFree	170	70.85	7.33	78.33	3.85
circular	Cusp_2str	158	79.25	6.24	85.63	2.91
	Hyb_Cusp	158	72.94	10.48	83.57	2.98
	Hyb_SyncFree	158	65.43	7.58	73.17	3.40
unit-vector	Cusp_2str	170	89.35	6.99	96.48	2.78
	Hyb_Cusp	170	79.81	6.46	86.42	3.10
	Hyb_SyncFree	170	72.05	8.68	80.89	3.31

where the impact of the operations that correspond to Set A and Set B on the total runtime is larger. If we consider only this stage, the acceleration achieved by the dual-GPU variant, with respect to the CPU version, is of up to 14×. Nevertheless, the cost of the unaccelerated parts of the solver significantly affect the performance in some cases. Overall, for some of the problem instances the execution time of the iterative solve is reduced by a factor in the range 4–9×, with respect to the multicore-based version. On the other hand, it is especially remarkable that with our strategy the speed-up associated to doubling the many-core devices overcomes the linear evolution for the largest cases. Note that the dual-GPU version of BiCG outperforms the Cusp_1str variant by a factor of up to 2.5× for the whole method yielding a superlinear speed-up.

The first proposed task-data parallel variant over only one GPU, Cusp_2str, offers a similar performance as the data-level parallel scheme that exploits only one GPU. The differences in runtimes between both variants are negligible for all the addressed test cases. Thus, the usage of GPU streams in this context has little effect on the performance of the solver. This result confirms the preliminary idea that the Cusp_2str version is not able to compute both tasks (both streams) in parallel, as it can be observed in the timelines extracted with nVidia Visual Profiler, for the *cage15* case in Figure 3.

Regarding the hybrid GPU-CPU variants, i.e. Hyb_Cusp version, the results show that off-loading the transposed SpMV to the multicore can yield important benefits. In our experiments, the improvements with respect to Cusp_2str variant (or GPU) range from 3% to almost 65%. A number of factors influence the relation between the performance of Cusp_2str and Hyb_Cusp variants. For example, as illustrated in Figure 4 for matrix *cage15*, a large fraction of the performance improvement is due to the reduction of the time taken by the transposed SpMV. In Cusp_2str, this takes more than half the runtime of the iteration, and the multicore implementation in Hyb_Cusp is able to both reduce this time in half and allow the overlapping of part of the application of the preconditioner. A different situation is observed for matrix *A200*, where the cuSparse routine for the transposed SpMV outperforms the MKL counterpart. In this case though, the difference between the runtime of both routines is less critical, and a performance improvement is obtained regardless. This is due to the almost perfect overlapping of the application of the preconditioner with the transposed SpMV.

On the other hand, the results also reveal that the use of our synchronization-free routine can contribute with an additional performance improvement by enabling a higher degree of overlapping between operations. This will depend mostly on the number of level-sets of the incomplete factors, as additional levels imply the launching of more kernels, and augment the corresponding overhead. For instance, the analysis of the triangular factors generated for matrix *A200* yields only 40 levels, whereas for *cage15* it generates 616. Thus, it is not surprising that the difference between the runtimes of Hyb_Cusp and Hyb_SyncFree for matrix *A200* is minimal, while for *cage15* the performance gain is relevant.

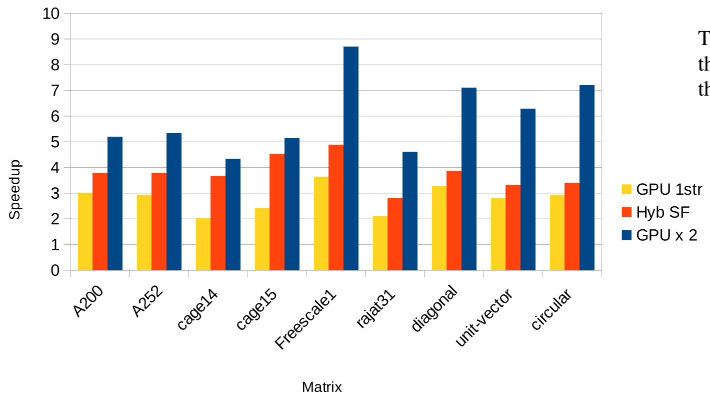

Figure 6: Comparison of the speed-up obtained (respect to Cpu_mkl) between Cusp_1str, Hyb_SyncFree and GPU_×2 variants.

Table 4: Runtime (in seconds) of version Hyb_SyncFree for the application of the transposed preconditioner and the percentage of the total runtime taken by this stage.

Matrix	Levs.	App. Transp. Prec.	% TP
A200	1	0.45	25%
A252	1	0.71	25%
cage14	1	0.25	39%
Freescale1	3	14.86	46%
rajat31	3	0.64	46%
cage15	1	0.87	37%
diagonal	3	52.90	61%
circular	3	47.28	65%
unit-vector	3	52.80	65%

4.4. *Characterizing the performance of the different variants*

As we have proposed several strategies to compute the BiCG method of ILUPACK in different hardware platforms, it is important to discuss a characterization of the variants in order to estimate which is the best method for a given matrix and hardware platform. Next, we analyze the factors that affect the performance of each variant, knowing that the distinct implementations require different hardware configurations. Furthermore, a complete analysis should consider other costs associated with the use of these platforms, and not only the execution time.

Since Hyb_SyncFree is the single-GPU variant that delivers the best performance for the majority of cases, and GPU_×2 is clearly the best performing version, we are interested in analyzing the performance gap between these two solution. In Figure 6, which presents a graphical comparison of the Cusp_1str, GPU_×2 and Hyb_SyncFree variants, it can be observed that the differences between GPU_×2 and Hyb_SyncFree are not always significant. In fact, with a more careful study, we can group the problems into four clusters based on the difference in performance between both versions:

1. *circular*, *unit*, *diagonal*: with differences near to 2.0×.
2. *Freescale1*, *rajat31*: with differences around 1.7×.
3. *A252*, *A200*: with differences in the order of 1.4×.
4. *cage14*, *cage15*: showing a gap of 1.2×.

The analysis of these results reveals that these differences are mainly related with the cost of the application of the transposed preconditioner. It should be noted that this operation lies in the critical path of version Hyb_SyncFree, as there it is performed individually at the end of the computation. whereas, in GPU_×2 this operation is overlapped with other computations. Table 4 summarizes the runtime in version Hyb_SyncFree for the application of the transposed preconditioner and the percentage of the total runtime taken by this stage.

To understand why the fraction of the runtime represented by this stage is so different in each case, it is important to note that the number of levels of the ILUPACK preconditioner is related with the inverse-based multilevel ILU factorization process and the characteristics of each particular matrix. This is described, from a mathematical perspective, in [3].

When the preconditioner presents more than one level, the application process requires the computation of two SpMV operations at each level in order to operate with the Schur complement of the factorization of the level. In the transposed preconditioner, the SpMV kernel works with the respective transposes of this Schur complement and, as we stated previously, the transpose SpMV kernel strongly deteriorates the performance of the entire process. In our test cases, the matrices with a factorization that consists of only one level are *cage14*, *cage15*, *A200* and *A252*; while the matrices with more than one level are *circular*, *unit*, *diagonal*, *Freescale1* and *rajat31*.

Considering the previous discussion, it is evident that the number of levels in the preconditioner, and especially whether or not the transposed SpMV has to be computed as part of the application of the preconditioner, is the factor that fundamentally determines the differences between the Hyb_SyncFree and GPU_×2 variants. It is worth noting that the number of levels of the preconditioner is known before the solver starts, so it can be used to select between the different computation strategies presented earlier.

A second aspect that affects the difference between the runtime of both variants (although in a lower degree), is the performance of the synchronization-free GPU routine to solve the triangular linear systems. Naturally, this has a stronger effect when the preconditioner consists of only one level and no SpMV has to be computed.

In the selected test cases, the synchronization-free solver outperforms the cuSparse `cusparseDcsrsv_solve` routine by a factor of approximately 3× for matrices *cage14* and *cage15*, but both triangular solvers offer similar performances for matrices *A200* and *A252*. When there is a significant advantage of synchronization-free solver, the extra cost in Hyb_SyncFree implied by the impossibility of overlapping the transposed preconditioner with other computations can be partially compensated by the use of the faster solver. However, two aspects should be noted. First, determining which triangular solver will perform better for a given matrix is still an open question. Second, if a prediction tool was available, it would be easy to modify the GPU_×2 version to employ the synchronization-free solver for the adequate matrices as well.

Table 5: Performance gap between the the GPU_×2 and Hyb_SyncFree variants for 4 non-symmetric matrices of the Suite Sparse collection.

Matrix	*n*	*nnz*	Levs.	% TP	Gap GPU_×2
G2_circuit	150,102	726,674	1	34%	1.20
G3_circuit	1,585,478	7,660,826	1	22%	1.36
atmosmodd	1,270,432	8,814,880	3	69%	1.50
Transport	1,602,111	23,487,281	3	65%	1.99

Finally, in order to provide a better numerical evaluation of the previous ideas, we include 4 additional non-symmetric matrices from the Suite Sparse collection. Table 5 offers the description of their main properties, including the dimension of each matrix (*n*), the number of non-zero coefficients (*nnz*), the number of levels of the preconditioner (Levels), the percentage of the total runtime taken by the transposed preconditioner in the Hyb_SyncFree variant (% TP), and the relation between the speed-ups of GPU_×2 and Hyb_SyncFree variants (gap GPU_×2). As we can observe, the results shown in Table 5 are aligned with our previous conjectures.

5. Related work

A number of research efforts have reported important benefits for the solution of sparse linear systems of equations using the BiCG method on multi-core and many-core platforms. However, many of these address the non-preconditioned version of the method, where the sparse matrix-vector product (SpMV) is the main bottleneck.

In [19] a variant of the two-sided Lanczos iteration that is equivalent to the BiCG in exact arithmetic is proposed. The new method reduces the global synchronization points of the original BiCG reporting performance improvements on a parallel machine with 120 processors. The implementation uses a domain-decomposition approach to distribute the work among the processors and performs the direct and transposed matrix-vector products concurrently.

Based on this work, in [20] Yang et al. propose an Improved BiCG algorithm for large-scale sparse matrices on distributed memory platforms. The authors follow the ideas in [21] for the distribution of the data among the processors as well as for the communication schemes, efficiently overlapping communication and computation.

In 2009, Jost et al. present a multi-algorithm GPU solver for matrices that contain few-diagonals [22]. The set of solvers includes an implementation of the (non-preconditioned) BiCG method for single-core and for GPUs, using a format similar to DIA to store the sparse matrices. In 2010, N. Garcia [23] presents a preconditioned BiCG to solve the power flow problem using GPUs. The implementation uses a row-based sparse format to store the Jacobian of an outer Newton-Raphson iteration. In 2012, Ortega et al. proposed an implementation of the BiCG for GPUs that focuses on improving the performance of the sparse matrix-vector products on complex matrices. In this work, both A and A^T are explicitly stored in the GPU. The authors compare the performance of the BiCG using the standard CSR SpMV routine of cuSparse, and a routine for the ELLR-T format [24]. They show that the ELLR-T implementation outperforms the CSR-based code on a C2050 GPU, as well as an MKL-based implementation running on an eight-core processor. Their experimental analysis reveals that the poor performance of the inner products of the BiCG strongly degrades the overall performance of the method for some matrices, even though they only represent a small fraction of the workload. The authors address this problem on [25], where they perform the fusion of the level-1 BLAS routines of the BiCG to improve their performance. This optimization improves the performance of the method by up to 30%.

6. Concluding Remarks and Future Work

ILUPACK offers implementations of a variety of Krylov-based numerical methods for the solution of sparse linear systems, and its advanced ILU-based preconditioner has achieved remarkable results in scenarios where other general purpose preconditioners tend to fail. Unfortunately, its higher complexity makes its application costly, motivating the inclusion of high performance computing techniques to mitigate this issue.

Data parallel versions, including a GPU-enabled implementation of ILUPACK's BiCG solver, have been proposed in some of our previous work. Here, we have revisited and extended these previous efforts to offer a variant of ILUPACK's BiCG routine capable of efficiently exploiting the parallel processing power of dual-GPU platforms.

In addition, we have extended the work analyzing the benefits offered by the exploitation of task parallelism, when a single GPU is only available. In this context, we have evaluated the use of GPU streams in order to increase the concurrency, and the use of the multicore CPU to allow further overlapping of independent operations. Finally, we take advantage of a recently proposed synchronization-free solver for sparse triangular linear systems to accelerate the execution on a multicore CPU, since this kind of methods diminishes the CPU overhead incurred by launching cuSparse kernels. The experimental evaluation shows that we can reach fair runtime reductions in single-GPU platforms, despite of the processing and memory limitations.

The evaluation also exposes that, in some cases, after the acceleration effort using GPUs, the performance bottleneck is shifted to the cost of data transference. As part of future work, we plan to address this issue, and evaluate the behaviour of this solver on platforms with nvLink. We will also analyze the impact of recent GPU architectures in these codes, in particular, the new thread scheduling capabilities of the Volta generation and the novel tensor cores.

Acknowledgments

J. I. Aliaga and E. S. Quintana-Ortí were supported by project TIN2017-82972-R of the MINECO and FEDER. E. Dufrechou and P. Ezzatti were supported by *Programa de Desarrollo de las Ciencias Básicas* (PEDECIBA), Uruguay.

References

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

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

[3] M. Bollhöfer, Y. Saad, Multilevel preconditioners constructed from inverse–based ILUs, SIAM J. Sci. Comput. 27 (5) (2006) 1627–1650.

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

[5] J. I. Aliaga, M. Bollhöfer, A. F. Martín, E. S. Quintana-Ortí, Exploiting thread-level parallelism in the iterative solution of sparse linear systems, Parallel Computing 37 (3) (2011) 183–202.

[6] J. I. Aliaga, M. Bollhöfer, A. F. Martín, E. S. Quintana-Ortí, Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors, in: K. Jónasson (Ed.), Applied Parallel and Scientific Computing, LNCS, Vol. 7133 of Lecture Notes in Computer Science, 2012, pp. 162–172.

[7] J. I. Aliaga, M. Bollhöfer, E. Dufrechou, P. Ezzatti, E. S. Quintana-Ortí, A data-parallel ilupack for sparse general and symmetric indefinite linear systems, in: F. Desprez, P.-F. Dutot, C. Kaklamanis, L. Marchal, K. Molitorisz, L. Ricci, V. Scarano, M. A. Vega-Rodríguez, A. L. Varbanescu, S. Hunold, S. L. Scott, S. Lankes, J. Weidendorfer (Eds.), Euro-Par 2016: Parallel Processing Workshops, Springer International Publishing, Cham, 2017, pp. 121–133.

[8] J. I. Aliaga, M. Bollhöfer, E. Dufrechou, P. Ezzatti, E. S. Quintana-Ortí, Extending ilupack with a task-parallel version of bicg for dual-gpu servers, in: Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'18, ACM, New York, NY, USA, 2018, pp. 71–78. doi:10.1145/3178442.3178450.
URL `http://doi.acm.org/10.1145/3178442.3178450`

[9] T. A. Davis, Y. Hu, The university of florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1:1–1:25. doi:10.1145/2049662.2049663.
URL `http://doi.acm.org/10.1145/2049662.2049663`

[10] O. Schenk, A. Wächter, M. Weiser, Inertia Revealing Preconditioning For Large-Scale Nonconvex Constrained Optimization, SIAM J. Scientific Computing 31 (2) (2008) 939–960.

[11] M. Bollhöfer, M. J. Grote, O. Schenk, Algebraic multilevel preconditioner for the helmholtz equation in heterogeneous media, SIAM Journal on Scientific Computing 31 (5) (2009) 3781–3805.

[12] C. Lanczos, Solution of systems of linear equations by minimized iterations, J. Res. Nat. Bur. Standards 49 (1) (1952) 33–53.

[13] C. Nvidia, Cublas library, NVIDIA Corporation, Santa Clara, California 15 (27) (2008) 31.

[14] E. Anderson, Y. Saad, Solving sparse triangular linear systems on parallel computers, International Journal of High Speed Computing 1 (01) (1989) 73–95.

[15] 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 1.

[16] D. Erguiz, E. Dufrechou, P. Ezzatti, Assessing sparse triangular linear system solvers on gpus, in: 2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), Vol. 00, 2017, pp. 37–42. doi:10.1109/SBAC-PADW.2017.15.
URL `doi.ieeecomputersociety.org/10.1109/SBAC-PADW.2017.15`

[17] W. Liu, A. Li, J. Hogg, I. S. Duff, B. Vinter, A synchronization-free algorithm for parallel sparse triangular solves, in: European Conference on Parallel Processing, Springer, 2016, pp. 617–630.

[18] E. Dufrechou, P. Ezzatti, Solving sparse triangular linear systems in modern gpus: A synchronization-free algorithm, in: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), Vol. 00, 2018, pp. 196–203. doi:10.1109/PDP2018.2018.00034.
URL `doi.ieeecomputersociety.org/10.1109/PDP2018.2018.00034`

[19] H. M. Bücker, M. Sauren, Reducing Global Synchronization in the Biconjugate Gradient Method, Springer US, Boston, MA, 1999, pp. 63–76.

[20] L. T. Yang, R. P. Brent, The improved bicg method for large and sparse linear systems on parallel distributed memory architectures, in: Proceedings of the 16th International Parallel and Distributed Processing Symposium, IPDPS '02, IEEE Computer Society, Washington, DC, USA, 2002, pp. 315–.
URL `http://dl.acm.org/citation.cfm?id=645610.661567`

[21] A. Basermann, B. Reichel, C. Schelthoff, Preconditioned cg methods for sparse matrices on massively parallel machines, Parallel Computing 23 (1997) 381–398.

[22] T. Jost, S. Contassot-Vivier, S. Vialle, An efficient multi-algorithms sparse linear solver for GPUs, in: ParCo2009, Frédéric Desprez, Lyon, France, 2009.
URL `https://hal.inria.fr/inria-00430520`

[23] N. Garcia, Parallel power flow solutions using a biconjugate gradient algorithm and a newton method: A gpu-based approach, in: IEEE PES General Meeting, 2010, pp. 1–4. doi:10.1109/PES.2010.5589682.

[24] F. Vázquez, G. Ortega, J. J. Fernández, E. M. Garzón, Improving the performance of the sparse matrix vector product with gpus, in: 2010 10th IEEE International Conference on Computer and Information Technology, 2010, pp. 1146–1151. doi:10.1109/CIT.2010.208.

[25] S. Tabik, G. Ortega, E. M. Garzón, Performance evaluation of kernel fusion blas routines on the gpu: iterative solvers as case study, The Journal of Supercomputing 70 (2) (2014) 577–587. doi:10.1007/s11227-014-1102-4.
URL `https://doi.org/10.1007/s11227-014-1102-4`