Aliaga, JI.; Dufrechou, E.; Ezzatti, P.; Quintana-Orti, ES. (2019). An efficient GPU version of the preconditioned GMRES method. The Journal of Supercomputing. 75(3):1455-1469. https://doi.org/10.1007/s11227-018-2658-1

Journal of Supercomputing manuscript No.
(will be inserted by the editor)

An efficient GPU version of the preconditioned GMRES method

José I. Aliaga · Ernesto Dufrechou · Pablo Ezzatti · Enrique S. Quintana-Ortí

Received: date / Accepted: date

Abstract In a large number of scientific applications, the solution of sparse linear systems is the stage that concentrates most of the computational effort. This situation has motivated the study and development of several iterative solvers, among which preconditioned Krylov subspace methods occupy a place of privilege. In a previous effort, we developed a GPU-aware version of the GMRES method included in ILUPACK, a package of solvers distinguished by its inverse-based multilevel ILU preconditioner. In this work we study the performance of our previous proposal and integrate several enhancements in order to mitigate its principal bottlenecks. The numerical evaluation shows that our novel proposal can reach important runtime reductions.

Keywords GPUs, GMRES, sparse triangular solver, MGSO

1 Introduction

The solution of sparse linear systems of large dimension is a challenging task, as well as the main performance bottleneck for a myriad of scientific and engineering applications, such as the numerical solution of partial differential equations (PDEs), and the simulation of circuits or graph theory problems [14].

This situation has motivated the study and development of several solvers. When the dimension of the systems is moderate, direct solvers perform reasonably well, and are usually the default choice. However, this type of methods face difficulties when handling large unstructured sparse systems, because of their outstanding memory requirements. In this cases, iterative methods,

J. I. Aliaga · E. S. Quintana-Ortí
Dep. de Ingeniería y Ciencia de la Computación, Universidad Jaime I, Castellón, Spain.
E-mail: {aliaga,quintana}@icc.uji.es

E. Dufrechou · P. Ezzatti
Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay.
E-mail: {edufrechou,pezzatti}@fing.edu.uy

among which preconditioned Krylov subspace methods occupy a place of privilege, become an attractive option. In this context, ILUPACK[1] is a remarkable software package that comprises several iterative sparse linear system solvers, in conjunction with an inverse-based multilevel incomplete LU (ILU) preconditioning technique [7]. Although there is not a best preconditioner for all cases, the robustness of this technique is especially remarkable in comparison with other ILU preconditioners families [6,15] due to its control over the growth of the inverse approximate factors. Unfortunately, these desirable numerical properties are counteracted by the elevated computational cost that the preconditioner implies. This situation has motivated the development of parallel versions of ILUPACK in the past [1,3,4], designed for shared and distributed memory environments. These implementations offer important values of speedup and scalability, but are only able to address symmetric and positive definite (SPD) matrices.

Later, in [1] we designed a GPU-aware version of ILUPACK's Conjugate Gradient solver, able to exploit the computational power offered by the GPUs, and in [2], we extend these ideas with the aim of accelerating the solution of general and symmetric indefinite linear systems using the ILUPACK BiCG, SQMR, and GMRES implementations.

In this work, we enhance our previous proposal of a massively-parallel variant of the GMRES method, by attacking the main computational bottlenecks of our past developments. The study of this version reveals that, after the GPU-acceleration of the SpMV of the GMRES and the preconditioner routine, the more demanding stages from the computational perspective are the solution of sparse triangular linear systems (SpTrSV) on the GPU and the calculation in CPU of the modified Gram-Schmidt with selective re-orthogonalization (MGSO) procedure [11]. In order to overcome these limitations, we substitute the cuSparse routine that we have used so far for the SpTrSV, by our own implementation, based on a recently developed approach that avoids the constant synchronization with the CPU that cuSparse suffers from. We also design and integrate an accelerated GPU-version of the MGSO procedure that balances the costs related with the host-device communication. The experimental evaluation compares the new GPU-enabled version of the solver with our original data-parallel proposal, over a set of real linear systems taken from the Suite Sparse Matrix Collection[2]. The results show that the novel developments can outperform the original ones in values of up to 10× for each stage, which translates into a runtime reduction of between 1.1× and 3.1× for the whole solver.

The rest of the paper is structured as follows. Section 2 revisits the main concepts of the iterative solvers integrated in ILUPACK and the general strategies followed to accelerate them with the use of GPUs. In Section 3, we present the performance study of our previous data-parallel variant of the preconditioned GMRES method of ILUPACK. Next, Section 4 describes the main

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

[2] formerly known as University of Florida (Sparse) Matrix Collection

aspects of our current proposal. This is followed by the experimental evaluation in Section 5. Finally, a few remarks and some lines of future work are summarized in Section 6.

2 Accelerated Solution of Sparse Linear Systems with ILUPACK

ILUPACK is a software package that comprises several methods for the numerical solution of sparse linear systems. It includes a comprehensive set of routines that handle different matrix types, precisions, and arithmetic, covering Hermitian positive definite/indefinite and general real and complex matrices.

The most remarkable feature of this package is its inverse-based multilevel ILU preconditioner, which can be applied to accelerate the convergence of Krylov subspace-based iterative solvers. However, despite the advantageous numerical properties, the high complexity of this type of preconditioner makes its application, which occurs (at least) once per iteration of the solver, the most demanding task from the computational point of view.

2.1 Computation of the preconditioner

The computation of the ILUPACK preconditioner, for a system $Ax = b$ where $A \in \mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^n$, is organized in a recursive manner. First, the matrix A is scaled by a diagonal matrix $\tilde{D} \in \mathbb{R}^{n\times n}$ and reordered by a fill-reducing permutation $\tilde{P} \in \mathbb{R}^{n\times n}$. Then, an incomplete factorization is performed on the resulting sparse matrix. This factorization procedure is based on the Crout variant of the LU decomposition and, broadly speaking, applies a test on the current row and column that determines how their factorization will affect the norm of the inverse factors. If this norm exceeds a certain threshold, the factorization of the current row and column is postponed, sending the rows and columns that fail the test to the bottom right corner of the matrix. After all the rows and columns have been processed, the factorization proceeds recursively on the the bottom-right $k \times k$ submatrix that contains the rejected k pivots, until this matrix is void or sufficiently dense to be handled by a dense solver.

If $\hat{P} \in \mathbb{R}^{n\times n}$ is a matrix that records the permutations of rows and columns that preserve the norms of the inverse factors, at level l, the multilevel preconditioner can be recursively expressed as

$$M_l = \begin{pmatrix} B & F \\ G & C \end{pmatrix} \approx \tilde{D}\tilde{P}\hat{P} \begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} \hat{P}^T \tilde{P}^T \tilde{D}^{-1}, \qquad (1)$$

where L_B and U_B^T are unit lower triangular, D_B is diagonal, and M_{l+1} stands for the preconditioner computed at level $l+1$. A more detailed explanation of the factorization procedure can be found in [7].

2.2 Application of the preconditioner during the iterative solve

Now consider the computation of $z := M_l^{-1} r$, which is the application of the preconditioner on the residual vector r at level l. This is equivalent to the solution of the following system of linear equations:

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} \hat{P}^T \tilde{P}^T \tilde{D}^{-1} z = \hat{P}^T \tilde{P}^T \tilde{D}^{-1} r. \tag{2}$$

After computing $\hat{r} := \hat{P}^T \tilde{P}^T (\tilde{D} r)$, and partitioning the vectors accordingly, the linear system in (2), can be solved for $w (= \hat{P}^T \tilde{P}^T \tilde{D}^{-1} z)$ in three steps,

$$\begin{pmatrix} L_B & 0 \\ L_G & I \end{pmatrix} \begin{pmatrix} y_B \\ y_C \end{pmatrix} = \begin{pmatrix} \hat{r}_B \\ \hat{r}_C \end{pmatrix}, \tag{3}$$

$$\begin{pmatrix} D_B & 0 \\ 0 & M_{l+1} \end{pmatrix} \begin{pmatrix} x_B \\ x_C \end{pmatrix} = \begin{pmatrix} y_B \\ y_C \end{pmatrix}, \tag{4}$$

$$\begin{pmatrix} U_B & U_F \\ 0 & I \end{pmatrix} \begin{pmatrix} w_B \\ w_C \end{pmatrix} = \begin{pmatrix} x_B \\ x_C \end{pmatrix} \tag{5}$$

where the second system implies the recursive stage.

This leads to the following expressions for y, x, and w:

$$L_B y_B = \hat{r}_B \ , \ y_C := \hat{r}_C - L_G y_B. \tag{6}$$

$$x_B := D_B^{-1} y_B \ , \ x_C := M_{l+1}^{-1} y_C. \tag{7}$$

$$w_C := x_C \ , \ U_B w_B = x_B - U_F w_C. \tag{8}$$

In the recursion base step, M_{l+1} is void and only x_B has to be computed. Once w is obtained z is easily computed as $z := \tilde{D}(\tilde{P}(\hat{P} w))$.

To save memory, ILUPACK keeps only the blocks G and F, discarding L_G and U_F (which suffer from the factorization fill-in) after the preconditioner is computed. Thus, (6) is changed as:

$$L_G = G U_B^{-1} D_B^{-1} \Rightarrow y_C := \hat{r}_C - G U_B^{-1} D_B^{-1} y_B = \hat{r}_C - G U_B^{-1} D_B^{-1} L_B^{-1} \hat{r}_B, \tag{9}$$

while the expressions related to (8) are modified as

$$U_F = D_B^{-1} L_B^{-1} F \Rightarrow U_B w_B = D_B^{-1} y_B - D_B^{-1} L_B^{-1} F w_C. \tag{10}$$

This yields the final expressions

$$\begin{array}{ll} L_B D_B U_B s_B = \hat{r}_B, & y_C := \hat{r}_C - G s_B, \\ L_B D_B U_B \hat{s}_B = F w_C & w_B := s_B - \hat{s}_B \end{array} \tag{11}$$

In summary, each level of the application of the preconditioner involves, two SpMV, solving two LDU linear systems, and three other types of operations: diagonal scalings, vector permutations, and vector updates of the form $x := a - b$.

2.3 Data-Parallel Variants of ILUPACK

As mentioned previously, although ILUPACK has demonstrated its effectiveness to improve the convergence rate of Krylov subspace-based iterative methods [6,7,15], in several cases the higher complexity in relation to other ILU-based alternatives can make it impractical from a computational perspective. To remedy this situation, we are interested in the exploitation of the data-level parallelism of the operations that compose the application of the preconditioner, and those that integrate the different solvers, using GPUs. In this work, we focus our efforts on developing an efficient GPU-aware implementation of ILUPACK's GMRES method for general systems.

Before we introduce our new contributions, it is necessary to revisit the data-parallel implementations of ILUPACK that were introduced previously in [2]. For many of the systems that we evaluated in [2], the computational cost required to apply the preconditioner was dominated by the sparse triangular system solves (SpTrSV) and the SpMV appearing in equation (11). We relied on the nVidia cuSparse library to perform these operations in the GPU, since it is a publicly available and well accomplished library that offers kernels for these two operations. The vector updates that appear in the preconditioner gain only mild importance for highly sparse matrices of large dimension, and were accelerated in our codes via *ad-hoc* CUDA kernels.

ILUPACK represents the multilevel preconditioner as a linked list that contains the information computed at each level, such as pointers to the factors of B, the G and F blocks, and the permutation vectors that represent the matrices $\tilde{D}$, $\tilde{P}$, and $\hat{P}$ of Section 2.1.

The factorization of B is stored in a modified CSR (MCSR) format [10], were the columns of the strict lower part of L_B and the rows of the strict upper part of U_B are interlaced in the same CSR data structure, while the diagonal matrix D_B is stored (inverted) as a separate vector.

Using cuSparse requires to adapt the sparse storage layouts employed by ILUPACK to those supported by cuSparse. We therefore separate the MCSR data structure into lower and upper factors, stored by rows in the conventional CSR format. This transformation is done only once, during the calculation of each level of the preconditioner, and occurs entirely in the CPU. Afterwards, these factors are transferred to the GPU, where the triangular systems involved in the preconditioner application are solved later by means of two consecutive calls to the routine `cusparseDcsrsv_solve`. This routine requires a previous analysis phase, which determines a parallel execution schedule based on the dependencies between the equations of the linear system. This stage is executed only once for each level of the preconditioner, and runs asynchronously with respect to the host CPU.

In order to compute the SpMV in the GPU, G and F are also transferred to the device during the computation of the preconditioner. These matrices need no reorganization, as they are stored in plain CSR format.

Operation	kernel
$A \to M$	Compute preconditioner
Initialize $x_0, r_0, q_0, p_0, \ldots$	
$k := 0$, $\beta := \|r_0\|$, $\xi_1 := (1, 0, \ldots, 0)^T$	
while $(k < m)$	
$z_{k+1} := M^{-1}(Aq_k)$	SpMV + apply preconditioner
$[H(:, k+1), q_{k+1}] := \text{MGSO}(Q_k, z_{k+1})$	
$k := k + 1$	
end while	
$y_k := \arg\min_y \|\beta\xi_1 - H_k y\|$	
$x_k := x_0 + Q_k y_k$	
$\tau_k := \|b - Ax_k\|_2$	
if $(\tau_k > \tau_{\max})$ **then**	
$x_0 := x_k$	
restart GMRES	
end if	

Fig. 1 Algorithmic formulation of the preconditioned GMRES (m) method. The threshold $\tau_{\max}$ is an upper bound on the residual for the computed approximation to the solution.

3 Performance study of the GPU-based preconditioned GMRES

In previous work [2], we accelerated ILUPACK's implementation of GMRES (which we describe in Figure 1) by off-loading the application of the preconditioner and the SpMV operations that appear in each iteration of the solver to the hardware accelerator. In the sequential version of ILUPACK these were the most time-consuming operations of the method. However, once we accelerated these two tasks, other operations of gained significant importance, especially for some problems of large dimension that show a slow convergence rate.

In this section we study the computational performance of our previous data-parallel version of the GMRES method to determine the performance bottlenecks that arise after the acceleration of the SpMV and the preconditioner routine, and how they affect the overall performance of the solver.

We first describe the test cases and the hardware platforms utilized in the numerical evaluation. Later, we summarize the experimental results obtained from the execution of our previous effort.

3.1 Experimental Setup

The experimental platforms used in this work are two servers equipped with quad-core CPUs and nVidia GPUs of different characteristics. On the one hand, the GTX TitanX, of the Maxwell architecture family, is a graphic card well-suited for scientific computing applications, with 12 GB of memory and a superior double-precision performance, while the GTX 1080Ti is a recent and powerful gaming GPU, of the Pascal generation, that presents a higher number of cores, clock speed and memory bandwidth than the TitanX. Some specifications of each platform are summarized in Table 1.

Table 1 Description of the platform used in the experimental evaluation.

		MAXWELL	PASCAL
GPU	Model	GTX TitanX	GTX 1080 Ti
	Arch.	Maxwell	Pascal
	Cores	3072	3584
	Freq. (GHz)	1.08	1.58
	RAM (GB)	12	11
	Max. bw. (GB/s)	336.5	484
CPU	Model	Intel Core i7-6700	Intel Core i7-4770
	Cores	4	4
	Freq. (GHz)	3.40	3.40
	Cache (MB)	8	8
	RAM (GB)	64	16
	Max. bw. (GB/s)	34.1	25.6
Software	C compiler	GCC 4.8.5	GCC 7.3.1
	CUDA Toolkit	8.0	9.2

3.1.1 Test cases

We selected a set of medium to large-scale matrices from the Suite Sparse benchmark collection to numerically evaluate our enhancements. The choice of matrices intends to be sufficiently varied regarding the origin of the problem, its dimension, and its degree of sparsity. The details of the selected problems are displayed in Table 2.

Table 2 Matrices used in the experiments: dimension n and number of nonzeros nnz.

Matrix	n	nnz	nnz/n
apache2	715176	2766523	3.9
ASIC_320ks	321671	1827807	5.7
atmosmodd	1270432	8814880	6.9
cage13	445315	7479343	16.8
ecology2	999999	2997995	3.0
G3_circuit	1585478	4623152	2.9
lung2	109460	492564	4.5
parabolic_fem	525825	2100225	4.0
ship_003	121728	4103881	33.7
thermomech_dK	204316	2846228	13.9

3.2 Analysis of the baseline implementation

In Table 3 we summarize the runtimes taken by the original sequential CPU version of ILUPACK's GMRES method and the GPU data-parallel version described previously in platform MAXWELL. We detail the fraction of the total runtime implied by the SPMV operation of the iterative solver, the application of the preconditioner and the MGSO procedure.

Table 3 Evaluation of GMRES for selected cases with expensive re-orthogonalization: Number of iterations for convergence; execution time (in sec.) of each stage of the GMRES solver and cost (in %) relative to the total time; and relative residual of the solution. In the GPU variant, the SpMV and the application of the preconditioner (identified with the label "Appl. M^{-1}") proceed in the GPU, while MGSO and other operations are performed by the CPU.

			SpMV		Appl. M^{-1}		MGSO		Total
Matrix	Dev.	#It.	Time	%	Time	%	Time	%	time
apache2	CPU	5	0.016	12.33	0.072	54.54	0.026	19.33	0.132
	GPU	5	0.007	10.24	0.020	28.39	0.025	35.79	0.070
ASIC_320ks	CPU	2	0.006	20.41	0.015	49.28	0.002	6.05	0.030
	GPU	2	0.002	10.46	0.005	32.63	0.002	10.75	0.016
atmosmodd	CPU	46	0.388	5.52	5.003	71.12	1.561	22.19	7.035
	GPU	46	0.130	3.17	2.338	56.82	1.564	38.01	4.115
cage13	CPU	7	0.058	26.13	0.122	55.39	0.029	13.08	0.221
	GPU	7	0.008	7.71	0.063	57.74	0.026	24.12	0.108
ecology2	CPU	5	0.020	10.84	0.097	52.59	0.042	22.59	0.184
	GPU	5	0.009	9.43	0.025	25.92	0.037	38.70	0.096
G3_circuit	CPU	5	0.034	11.18	0.165	53.74	0.067	21.74	0.307
	GPU	5	0.014	9.20	0.042	26.70	0.060	38.67	0.156
lung2	CPU	4	0.002	13.56	0.011	58.49	0.002	13.15	0.018
	GPU	4	0.001	6.95	0.011	63.87	0.002	13.29	0.017
parabolic_fem	CPU	3	0.008	13.64	0.032	53.63	0.007	12.09	0.060
	GPU	3	0.003	10.88	0.008	25.07	0.008	24.46	0.031
ship_003	CPU	5	0.021	29.94	0.041	60.23	0.004	5.56	0.069
	GPU	5	0.003	7.17	0.026	73.80	0.004	10.49	0.035
thermomech_dK	CPU	500	1,630	11.08	10,312	70.07	2,697	18.33	14,717
	GPU	500	0.308	2.21	10.856	77.91	2.692	19.32	13.934

Considering the obtained results, we can see that the GPU variant offers runtime reductions from 1.1× to up to 2.0×. Regarding the individual execution of each stage, the SpMV operation diminishes its computational importance in the GPU variant, which lies below 10.88%, while the preconditioner routine, also reduces its importance in general. As a consequence, the MGSO procedure strongly increases its impact on the total runtime, moving from up to 22.59% to a range between 10.5% and 38.7%.

With these results we can conclude that the candidate stages to be accelerated are the application of the preconditioner and the MGSO procedure, which concentrate near to 90% of the execution time. In our previous effort the MGSO procedure was not ported to the accelerator, and thus this line of work is straightforward. In the case of the preconditioner routine, the main stages are already off-loaded to be performed in the GPU. Specifically, the SpMV over the rectangular G and F matrices, and the solution of sparse triangular linear solvers which is, without doubt, the most computationally demanding stage of the whole procedure, are already computed in the accelerator using kernels from the cuSparse library.

4 Extensions of the Data-Parallel GMRES of ILUPACK

In this section we describe the main aspects of the techniques employed to accelerate the GPU variant of the GMRES method in ILUPACK. First, we detail

Operation	kernel
Initialize $\omega := \|z_k\|^2, \tau := 0.99, i := 1$	K0. DOT product
while $(i <= k)$	Loop over all the preceding basis vectors
$H(i,k+1) := q_i^T z_k$	K1. DOT product
$z_k := z_k - H(i,k+1)q_i$	K2. AXPY
if $(\|H(i,k+1)\|^2 > \omega\tau)$ **then**	
$\beta := q_i^T z_k$	K3. DOT product
$H(i,k+1) := H(i,k+1) + \beta$	
$z_k := z_k - \beta q_i$	K4. AXPY
end if	
$\omega := \omega - \|H(i,k+1)\|^2$	
if $(\omega < 0)$ **then**	
$\omega := 0,$	
end if	
$i := i + 1$	
end while	
$q_{k+1} := z_k / \|z_k\|$	K5. SCAL
$H_{k,k+1} := \|z_k\|$	

Fig. 2 Algorithmic formulation of MGSO. The method performs a re-orthogonalization if the cosine between the two vectors is greater than τ, which is set to 0.99 in our case.

our efforts to accelerate the MGSO procedure using the GPU, to then proceed to describe the new SpTrSV routine that we employ seeking to enhance the performance of the preconditioner.

4.1 GMRES with Accelerated Data-Parallel MGSO

As we stated in the previous Section one of the major constraints of our original proposal was the important execution time related with non-parallelized operations, such as the MGSO procedure described in Figure 2. In this effort, we incorporate a GPU-based routine for the orthogonalization following preliminary ideas of [12,5]. In particular, we propose a version of the modified Gram-Schmidt procedure that follows a hybrid paradigm, trying to take profit from the best hardware architecture for each operation, but paying special attention to the minimization of the data interchange between the CPU and GPU memory spaces.

Specifically, we leverage the cuBlas library for the GPU computation the DOT products (i.e. operations K1 and K3) and vector updates (i.e. operations K2, K4 and K5). Considering that the SpMV of the iterative method is computed into the GPU, the basis vector required by the MGSO procedure are already in the accelerator memory. The process transforms the current basis vector such that it becomes orthogonal to the ones computed in previous iterations, returning also the coefficients of the current row of the Hessenberg matrix. As this matrix is small, and the remaining operations of the solver expose little parallelism and computational intensity, it is natural to perform these operations in the CPU. Because the coefficients of this matrix are the

result of DOT products with the basis vectors of size n, the GPU computes n flops for each coefficient that is transferred back to the CPU.

4.2 Incorporating a new triangular system solver

As we stated previously, the solutions of LDU-systems in (11) is without question the most computationally demanding step in the preconditioner application. Therefore, the performance of the preconditioner will be closely related to that of the SpTrSV routine used for the LDU-systems.

Until now, we have relied on the cuSparse library for the GPU execution of this operation. This library is publicly available, and it is constantly revised and adapted to the latest nVidia GPU architectures. However, in recent works we have proposed a new approach to solve sparse triangular systems in the GPU [8,9], developing several routines for this operation, and outperforming cuSparse for a varied set of sparse matrices.

The strategy proposed in [13,9] is based on avoiding the constant synchronization with the CPU that cuSparse suffers from. The procedure advances row-wise, such that each *warp* processes one row and each thread is assigned to one nonzero element. The threads must busy-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. Besides solving the linear system faster than cuSparse for several of the evaluated cases, this strategy has the advantage of being *self-scheduled*, which means that it does not need a previous analysis phase.

Unfortunately there are some cases where our new strategy falls short. Specifically, when the sparse matrix presents only few nonzero elements per row, as each warp processes one row and each thread 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 belong 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 this effort we exploit a new solution routine that, after performing a level-set analysis of the sparse matrix, leverages this information to overcome some of the above difficulties. The general idea is to partition the warp such that each one 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. This intends to alleviate the waste of resources when the rows have few nonzero elements,

while augmenting the number of rows being processed at a given time by the active thread-blocks. It is important that the rows processed by one warp do not have dependencies between them, so they all should belong to the same level.

The level-set analysis of the sparse matrix produces a `row_order` array that groups together the indexes of the rows of each level. To achieve a distribution as that described above, we further order this array so that the rows of 16 or more elements are placed after those that have between 8 and 16, which in turn must appear after those that have between 4 and 8, and so on. After that, we post-process the ordered array to determine the size of the partition, and the start position in `row_order` of the range of row indices to be processed by each warp.

In the solution stage, after obtaining the position of the corresponding nonzero entry of the sparse matrix, given by the base index in the `row_order array` plus the lane of the thread modulo the size of the partition, the threads will enter in the busy-waiting stage, where they fetch the corresponding entry of the solution vector, which is initialized with an invalid value (*NaN*), until they obtain a valid floating-point number.

After all values are obtained, the entire warp proceeds to the *product stage*, which comprises the multiplication of the nonzero coefficient by the recently obtained unknown, adding the result in a local register.

Finally, the values in the local registers of each partition of the warp are reduced in parallel using warp shuffle operations, and the first thread of each partition writes the final value of its corresponding unknown to the solution vector.

5 Numerical evaluation of the new variants

Tables 4 and 5 summarize the results of the experiments performed on platforms Maxwell and Pascal, respectively. In both tables, the total time taken by the iterative solver is disaggregated to show the part of the runtime corresponding to the most important stages of the solver, the SpMV, the application of the preconditioner, and the MGSO routine. For each of the matrices, the first line corresponds to the runtime of the version presented at [2], which performs the MGSO stage in the CPU and relies on the cuSparse library to compute the SpTrSV. The second line corresponds to our new proposal, which includes the GPU implementation of the MGSO procedure and incorporates the synchronization-free routine for the SpTrSV.

The results show that the inclusion of the new SpTrSV routine is highly beneficial for most of the evaluated problems, reducing the runtime of the application of the preconditioner by up to 3×. The acceleration factors are slightly better in the Maxwell platform than in Pascal but maintain certain correlation in the results for each matrix. Since each of the platforms presents a different version of the cuSparse library, it is difficult to determine the

Table 4 Experimental results in platform MAXWELL

matriz	#It.	spmv	prec		msgo		other	Total	
			time	accel.	time	accel.		time	accel
apache2	5	0.007 0.007	0.020 0.018	1.10	0.025 0.005	4.98	0.018 0.018	0.070 0.048	1.46
ASIC_320ks	2	0.002 0.002	0.005 0.003	1.74	0.002 0.001	2.29	0.007 0.007	0.016 0.013	1.25
atmosmodd	46	0.130 0.131	2.338 1.064	2.20	1.564 0.177	8.82	0.082 0.082	4.115 1.454	2.83
cage13	7	0.008 0.008	0.063 0.027	2.36	0.026 0.006	4.74	0.011 0.011	0.108 0.052	2.09
ecology2	5	0.009 0.009	0.025 0.026	0.96	0.037 0.006	5.93	0.025 0.025	0.096 0.067	1.45
G3_circuit	5	0.014 0.014	0.042 0.043	0.97	0.060 0.009	6.63	0.040 0.040	0.156 0.106	1.47
lung2	4	0.001 0.001	0.011 0.005	2.11	0.002 0.002	1.45	0.003 0.003	0.017 0.011	1.60
parabolic_fem	3	0.003 0.003	0.008 0.007	1.16	0.008 0.002	3.93	0.012 0.012	0.031 0.024	1.28
ship_003	5	0.003 0.003	0.026 0.010	2.69	0.004 0.002	1.89	0.003 0.003	0.035 0.017	2.05
thermomech_dK	500	0.308 0.311	10.856 3.650	2.97	2.692 0.537	5.01	0.078 0.078	13.934 4.576	3.05

Table 5 Experimental results in platform PASCAL

matriz	#It.	spmv	prec		msgo		other	Total	
			time	accel.	time	accel.		time	accel
apache2	5	0,008 0,008	0,017 0,017	0,96	0,025 0,004	5,96	0,049 0,049	0,098 0,078	1,26
ASIC_320ks	2	0,002 0,002	0,004 0,003	1,33	0,001 0,001	2,43	0,021 0,021	0,027 0,026	1,06
atmosmodd	46	0,176 0,169	1,507 0,861	1,75	1,779 0,173	10,30	0,164 0,165	3,625 1,368	2,65
cage13	7	0,010 0,010	0,047 0,023	1,99	0,022 0,005	4,29	0,030 0,031	0,108 0,069	1,58
ecology2	5	0,010 0,010	0,023 0,025	0,93	0,038 0,005	6,84	0,069 0,068	0,140 0,109	1,29
G3_circuit	5	0,016 0,016	0,036 0,040	0,90	0,066 0,008	8,19	0,110 0,109	0,229 0,174	1,32
lung2	4	0,001 0,001	0,007 0,004	1,64	0,002 0,001	1,67	0,007 0,007	0,017 0,014	1,24
parabolic_fem	3	0,003 0,003	0,006 0,006	1,00	0,007 0,002	4,24	0,034 0,034	0,051 0,046	1,11
ship_003	5	0,002 0,002	0,015 0,007	2,23	0,003 0,001	2,16	0,008 0,008	0,028 0,019	1,51
thermomech_dK	500	0,367 0,366	7,015 2,429	2,89	2,102 0,487	4,32	0,100 0,099	9,584 3,381	2,83

origin of this difference. In general, both versions of the preconditioner routine take less time in the PASCAL platform than they do in MAXWELL.

Regarding the acceleration of the MGSO routine, the results exhibit a notable reduction of the execution time, with acceleration factors that range from 1.5× to 10×. The results for this stage are also slightly better in the PASCAL platform, improving the MAXWELL times by 21% on average.

The combined effect of the two enhancements translates to an average performance improvement of 85% on platform MAXWELL (reaching speed-ups of up to 3×), and of 58% on platform PASCAL (obtaining a maximum acceleration of 2.83×). This performance improvement is remarkable, considering that the preconditioner routine of the baseline version is already data-parallel.

The analysis of the data also allows to confirm that the two factors that most affect the acceleration, specially of the MGSO routine, are the dimension of the problem and the number of iterations that the solver takes to converge to the desired tolerance. The influence of the dimension can be observed by considering the speedup of the MGSO stage for the 4 matrices that take 5 iterations to converge (*apache2*, *ecology2*, *G3_circuit* and *ship_003*), while the influence of the number of iterations is best appreciated when considering the *G3_circuit* and *atmosmodd* problems, which are of comparable dimension but differ greatly in the amount of iterations.

To understand this behaviour, it should be remarked that the basis vectors involved in the Arnoldi process of GMRES need to be transferred back to the CPU to perform the least squares solution prior to the restart. This means that n floating point values are transferred from the GPU to the CPU after every call to MGSO. However, the workload of this routine at a given iteration k of GMRES is proportional to $n \times \hat{k}$, where $\hat{k}$ is the number of vectors involved in the orthogonalization, which is equivalent to k modulo the restart parameter of GMRES. The ratio between the volume of transferences and the amount of computation in MGSO for a given iteration k is therefore $\hat{k}$, and improves with each iteration until the restart. As a consequence, the transference overhead is more significant when the solver converges in only a few steps and does not approach the restart point of GMRES, which is set at 30 iterations in our experiments. In future work we plan to avoid this communication by transferring the basis vectors asynchronously, such that this transference is overlapped with the computation of the next basis vector, or performing the last step of the GMRES on the accelerator.

6 Concluding Remarks and Future Work

After the GPU acceleration of the routine that applies the ILUPACK preconditioner on its implementation of the GMRES method, the Modified Grahm-Schmidt process of the GMRES gained importance to become, in some cases, the computational bottleneck of the solver.

In this work, we have enhanced our previous implementation by proposing a GPU-aware MGSO routine. As the acceleration of this stage shifts the most important fraction of computational effort back to the application of the preconditioner, we have also evaluated the use of a new approach to solve sparse triangular linear systems in the context of this operation. The new routine differs from the CUSPARSE implementation in that it uses a recently proposed synchronization-free strategy. It also introduces several enhancements in rela-

tion to previous proposals, such as the concurrent solution of several equations in each GPU warp.

The experimental results show fair acceleration factors in both stages, reducing the runtime of the MGSO by up to 10×, and reaching accelerations of up to 3× for the routine that applies the preconditioner. The combined application of both enhancements produces an average performance gain of 85% on platform MAXWELL (reaching speed-ups of up to 3×), and of 58% on platform PASCAL (obtaining a maximum acceleration of 2.83×).

As future work we plan to develop a strategy that minimizes the transference overhead that limit the performance of some particular cases. We are also interested in evaluating the behavior of our new version of this solver in GPUs of the Volta generations.

References

1. Aliaga, J.I., Badia, R.M., Barreda, M., Bollhöfer, M., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Exploiting task and data parallelism in ILUPACK's preconditioned CG solver on NUMA architectures and many-core accelerators. Parallel Computing **54**, 97–107 (2016)
2. Aliaga, J.I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: A data-parallel ILUPACK for sparse general and symmetric indefinite linear systems. In: Lecture Notes in Computer Science, 14th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms – HeteroPar'16. Springer (2016)
3. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Exploiting thread-level parallelism in the iterative solution of sparse linear systems. Parallel Computing **37**(3), 183–202 (2011)
4. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors. In: Applied Parallel and Scientific Computing, LNCS, vol. 7133, pp. 162–172 (2012)
5. Aliaga, J.I., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Accelerating a preconditioned GMRES method in massivelly parallel processors. In: CMMSE 2018 : Proceedings of the 18th International Conference on Mathematical Methods in Science and Engineering (2018) (2018)
6. Bollhöfer, M., Grote, M.J., Schenk, O.: Algebraic multilevel preconditioner for the helmholtz equation in heterogeneous media. SIAM Journal on Scientific Computing **31**(5), 3781–3805 (2009)
7. Bollhöfer, M., Saad, Y.: Multilevel preconditioners constructed from inverse–based ILUs. SIAM J. Sci. Comput. **27**(5), 1627–1650 (2006)
8. Dufrechou, E., Ezzatti, P.: 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, Canada, 2018. IEEE Computer Society (2018)
9. Dufrechou, E., Ezzatti, P.: 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), pp. 196–203 (2018). DOI 10.1109/PDP2018.2018.00034
10. Eijkhout, V.: LAPACK working note 50: Distributed sparse data structures for linear algebra operations. Tech. rep., Knoxville, TN, USA (1992)
11. Golub, G.H., Van Loan, C.F.: Matrix Computations (2013)
12. He, K., Tan, S.X.D., Zhao, H., Liu, X.X., Wang, H., Shi, G.: Parallel GMRES solver for fast analysis of large linear dynamic systems on GPU platforms. Integration, the VLSI Journal **52**, 10 – 22 (2016). URL `http://www.sciencedirect.com/science/article/pii/S016792601500084X`

13. Liu, W., Li, A., Hogg, J.D., Duff, I.S., Vinter, B.: Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. Concurrency and Computation: Practice and Experience **29**(21) (2017)
14. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM, Philadelphia, PA, USA (2003)
15. Schenk, O., Wächter, A., Weiser, M.: Inertia Revealing Preconditioning For Large-Scale Nonconvex Constrained Optimization. SIAM J. Scientific Computing **31**(2), 939–960 (2008)