Design of a Task-Parallel Version of ILUPACK for Graphics Processors

José I. Aliaga[1], Ernesto Dufrechou[2](✉), Pablo Ezzatti[2], and Enrique S. Quintana-Ortí[1]

[1] Dep. de Ingeniería y Ciencia de la Computación, Universidad Jaime I, 12.701, Castellón, Spain
{aliaga,quintana}@icc.uji.es

[2] Instituto de Computación, Universidad de la República, 11.300, Montevideo, Uruguay
{edufrechou,pezzatti}@fing.edu.uy

Abstract. In many scientific and engineering applications, the solution of large sparse systems of equations is one of the most important stages. For this reason, many libraries have been developed among which ILUPACK stands out due to its efficient inverse-based multilevel preconditioner. Several parallel versions of ILUPACK have been proposed in the past. In particular, two task-parallel versions, for shared and distributed memory platforms, and a GPU accelerated data-parallel variant have been developed to solve symmetric positive definite linear systems. In this work we evaluate the combination of both previously covered approaches. Specifically, we leverage the computational power of one GPU (associated with the data-level parallelism) to accelerate each computation of the multicore (task-parallel) variant of ILUPACK. The performed experimental evaluation shows that our proposal can accelerate the multicore variant when the leaf tasks of the parallel solver offer an acceptable dimension.

Keywords: ILUPACK · Graphic processors · Multi-core processors · Sparse linear systems · High performance

1 Introduction

In several scientific applications, the solution of large sparse systems of equations arise as one of the most important stages. Some examples appear in circuit and device simulations, quantum physics, large-scale eigenvalue computations, nonlinear sparse equations, and all kind of applications that involve the discretization of partial differential equations (PDEs) [6].

ILUPACK[1] (incomplete LU decomposition PACKage) is a numerical package that contains highly efficient sparse linear systems solvers, and can handle large-scale application problems of up to millions of equations. The solvers are based

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

on Krylov subspace methods [9], preconditioned with an inverse-based multilevel incomplete LU (ILU) factorization, which keeps a unique control of the growth of the inverse triangular factors that determines its superior performance in many cases [7,10,11].

Despite the remarkable mathematical properties of ILUPACK's preconditioner, it has the disadvantage of a costly computation and application, in comparison with more simple ILU preconditioners like ILU0. In [4] and [5] we proposed the exploitation of task-level parallelism in ILUPACK, for shared and distributed memory platforms, focusing on symmetric positive definite systems (s.p.d.), by using the preconditioned Conjugate Gradient (PCG) method. More recently, in [1] we used graphics accelerators to exploit data-level parallelism in the application of ILUPACK's preconditioner without altering its mathematical and numerical semantics, by off-loading the computationally-intensive kernels to the device.

In this work we evaluate the combination of both previous approaches, i.e. shared memory and co-processor data parallelism. Specifically, we leverage the computational power of one GPU (associated with the data-level parallelism) to accelerate the individual tasks – i.e. the operations that compose the application of the multilevel preconditioner – of the multicore (task-parallel) variant of ILUPACK. The experimental evaluation shows that our proposal is able to accelerate the multicore variant when the leaf tasks of the parallel solver offer an acceptable dimension.

The rest of the paper is structured as follows. In Sect. 2 we review the s.p.d. solver integrated in ILUPACK and we offer a brief study about the application of both parallel techniques (task and data-level). This is followed by the detailed description of our proposal in Sect. 3, and the experimental evaluation in Sect. 4. Finally, Sect. 5 summarizes the main concluding remarks and offers a few lines of future work.

2 Overview of ILUPACK

Consider the linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is sparse, $b \in \mathbb{R}^n$, and $x \in \mathbb{R}^n$ the sought-after solution. ILUPACK integrates an "inverse-based approach" into the ILU factorization of matrix A, in order to obtain a multilevel preconditioner. In this paper, we only consider systems with A s.p.d., on which PCG [9] is applied. Although each iteration of the PGC also involves a sparse matrix-vector product (SpMV) and several vector operations, in the remaining part of this section we mainly focus on the computation and application of the preconditioner, which are by far the most challenging operations.

2.1 Sequential (and Data Parallel) ILUPACK

Computation of the Preconditioner. This operation of ILUPACK relies on the Crout variant of the incomplete Cholesky (IC) factorization, yielding the approximation $A \approx L\Sigma L^T$, with $L \in \mathbb{R}^{n\times n}$ sparse lower triangular and

$\Sigma \in \mathbb{R}^{n \times n}$ diagonal. Before the factorization commences, a scaling and a reordering (defined respectively by $P, D \in \mathbb{R}^{n \times n}$) are applied to A in order to improve the numerical properties as well as reduce the fill-in in L. After these initial transforms, the factorization operates on $\hat{A} = P^T DADP$. At each step of the Crout variant, the "current" column of $\hat{A}$ is initially updated with respect to the previous columns of the triangular factor L, and the current column of L is then computed. An estimation of the norm of the inverse of L, with the new column appended, is obtained next. If this estimation is below a predefined threshold κ, the new column is accepted into the factor; otherwise the updates are reversed, and the corresponding row and column of $\hat{A}$ are moved to the bottom-right corner of the matrix. This process is graphically depicted in Fig. 1. Once $\hat{A}$ is completely processed in this manner, the trailing block only contains rejected pivots, and a partial IC factorization of a permuted matrix is computed:

$$\hat{P}^T \hat{A} \hat{P} \equiv \begin{bmatrix} B & F^T \\ F & C \end{bmatrix} = \begin{bmatrix} L_B & 0 \\ L_F & I \end{bmatrix} \begin{bmatrix} D_B & 0 \\ 0 & S_c \end{bmatrix} \begin{bmatrix} L_B^T & L_F^T \\ 0 & I \end{bmatrix} + E. \tag{1}$$

Here, $\|L_B^{-1}\| \lessapprox \kappa$ and E contains the elements dropped during the IC factorization. Restarting the process with $A = S_c$, we obtain a multilevel approach.

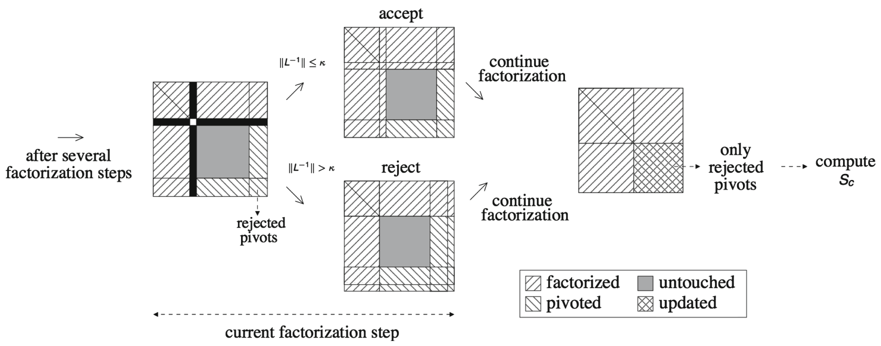

Fig. 1. A step of the Crout variant of the preconditioner computation.

Application of the Preconditioner. The application of the preconditioner in the PCG algorithm consists in the solution of the linear system $z := M^{-1}r$, where M is the preconditioner and r is the current residual. From (1), the preconditioner can be recursively defined, at level l, as

$$M_l = D^{-1} P \hat{P} \begin{bmatrix} L_B & 0 \\ L_F & I \end{bmatrix} \begin{bmatrix} D_B & 0 \\ 0 & M_{l+1} \end{bmatrix} \begin{bmatrix} L_B^T & L_F^T \\ 0 & I \end{bmatrix} \hat{P}^T P^T D^{-1}, \tag{2}$$

where $M_0 = M$. Operating properly on the vectors,

$$\hat{P}^T P^T D^{-1} z = \hat{z} = \begin{bmatrix} \hat{z}_B \\ \hat{z}_C \end{bmatrix}, \ \hat{P}^T P^T D r = \hat{r} = \begin{bmatrix} \hat{r}_B \\ \hat{r}_C \end{bmatrix}, \tag{3}$$

and applying $L_F = F L_B^{-T} D_B^{-1}$ (derived from (1)), we can expose the following computations to be performed at each level of the preconditioner [1]:

$$\begin{array}{lll} \text{Before:} & \hat{r} := \hat{P}^T P^T D r, & \text{Solve } L_B D_B L_B^T s_B = \hat{r}_B \text{ for } s_B, \\ & t_B := F s_B, & y_C := \hat{r}_B - t_B, \\ \text{Recursive step:} & \text{Solve } M_{l+1} \hat{z}_C = y_C \text{ for } \hat{z}_C, & \\ \text{After:} & \hat{t}_B := F^T \hat{z}_C, & \text{Solve } L_B D_B L_B^T \hat{s}_B = \hat{t}_B \text{ for } \hat{s}_B, \\ & \hat{z}_B := s_B - \hat{s}_B, & z := D P \hat{P} \hat{z}. \end{array} \tag{4}$$

To conclude this subsection, we emphasize that the data-parallel version of ILUPACK proceeds exactly in the same manner as the sequential implementation and, therefore, preserves the semantics of a serial execution.

2.2 Task Parallel ILUPACK

Following, we summarize the main ideas behind the task parallel version of ILUPACK. A more detailed explanation can be found in [4].

Computation of the Preconditioner. The task parallel version of this procedure employs Nested Dissection [9] to extract parallelism. To illustrate this, consider a ND partition, defined by a permutation $\bar{P} \in \mathbb{R}^{n \times n}$, such that

$$\bar{P}^T A \bar{P} = \left[\begin{array}{cc|c} A_{00} & 0 & A_{02} \\ 0 & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right]. \tag{5}$$

Computing a partial IC factorizations of the two leading blocks, A_{00} and A_{11}, yields the following partial approximation of $\bar{P}^T A \bar{P}$

$$\left[\begin{array}{cc|c} L_{00} & 0 & 0 \\ 0 & L_{11} & 0 \\ \hline L_{20} & L_{21} & I \end{array}\right] \left[\begin{array}{cc|c} D_{00} & 0 & 0 \\ 0 & D_{11} & 0 \\ \hline 0 & 0 & S_{22} \end{array}\right] \left[\begin{array}{cc|c} L_{00}^T & 0 & L_{20}^T \\ 0 & L_{11}^T & L_{21}^T \\ \hline 0 & 0 & I \end{array}\right] + E_{01},$$

where

$$S_{22} = A_{22} - (L_{20} D_{00} L_{20}^T) - (L_{21} D_{11} L_{21}^T) + E_2, \tag{6}$$

is the approximate Schur complement. By recursively proceeding in the same manner with S_{22}, the IC factorization of $\bar{P}^T A \bar{P}$ is eventually completed.

The block structure in (5) allows the permuted matrix to be decoupled into two submatrices, so that the IC factorizations of the leading block of both submatrices can be processed concurrently, with

$$A_{22} = A_{22}^0 + A_{22}^1, \left\{ \begin{array}{l} \left[\begin{array}{c|c} A_{00} & A_{02} \\ \hline A_{20} & A_{22}^0 \end{array}\right] = \left[\begin{array}{c|c} L_{00} & 0 \\ \hline L_{20} & I \end{array}\right] \left[\begin{array}{c|c} D_{00} & 0 \\ \hline 0 & S_{22}^0 \end{array}\right] \left[\begin{array}{c|c} L_{00}^T & L_{20}^T \\ \hline 0 & I \end{array}\right] + E_0 \\ \\ \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22}^1 \end{array}\right] = \left[\begin{array}{c|c} L_{11} & 0 \\ \hline L_{21} & I \end{array}\right] \left[\begin{array}{c|c} D_{11} & 0 \\ \hline 0 & S_{22}^1 \end{array}\right] \left[\begin{array}{c|c} L_{11}^T & L_{21}^T \\ \hline 0 & I \end{array}\right] + E_1, \end{array} \right. \tag{7}$$

and

$$S_{22}^0 = A_{22}^0 - \left(L_{20} D_{00} L_{20}^T\right) + E_2^0; \quad S_{22}^1 = A_{22}^1 - \left(L_{21} D_{11} L_{21}^T\right) + E_2^1.$$

Once the two systems are computed, S_{22} can be constructed given that

$$E_2 \approx E_2^0 + E_2^1 \ \rightarrow\ S_{22} \approx S_{22}^0 + S_{22}^1. \tag{8}$$

To further increase the amount of task-parallelism, one could find a permutation analogous to $\bar{P}$ for the two leading blocks following the ND scheme. For example, a block structure similar to (5) would yield the following decoupled matrices:

$$\left[\begin{array}{cccc|cc|c} A_{00} & 0 & 0 & 0 & A_{04} & 0 & A_{06} \\ 0 & A_{11} & 0 & 0 & A_{14} & 0 & A_{16} \\ 0 & 0 & A_{22} & 0 & 0 & A_{25} & A_{26} \\ 0 & 0 & 0 & A_{33} & 0 & A_{35} & A_{36} \\ \hline A_{40} & A_{41} & 0 & 0 & A_{44} & 0 & A_{46} \\ 0 & 0 & A_{52} & A_{53} & 0 & A_{55} & A_{56} \\ \hline A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} \end{array}\right] \rightarrow \begin{array}{cc} \bar{A}_{00} = \left[\begin{array}{c|c|c} A_{00} & A_{04} & A_{06} \\ \hline A_{40} & A_{44}^0 & A_{46}^0 \\ A_{60} & A_{64}^0 & A_{66}^0 \end{array}\right] & \bar{A}_{11} = \left[\begin{array}{c|c|c} A_{11} & A_{14} & A_{16} \\ \hline A_{41} & A_{44}^1 & A_{46}^1 \\ A_{61} & A_{64}^1 & A_{66}^1 \end{array}\right] \\ \bar{A}_{22} = \left[\begin{array}{c|c|c} A_{22} & A_{25} & A_{26} \\ \hline A_{52} & A_{55}^2 & A_{56}^2 \\ A_{62} & A_{65}^2 & A_{66}^2 \end{array}\right] & \bar{A}_{33} = \left[\begin{array}{c|c|c} A_{33} & A_{35} & A_{36} \\ \hline A_{53} & A_{55}^3 & A_{56}^3 \\ A_{63} & A_{65}^3 & A_{66}^3 \end{array}\right] \end{array} \tag{9}$$

Figure 2 illustrates the dependency tree for the factorization of the diagonal blocks in (9). The edges of the preconditioner *directed acyclic graph* (DAG) define the dependencies between the diagonal blocks (tasks), which dictate the order in which these blocks of the matrix have to be processed.

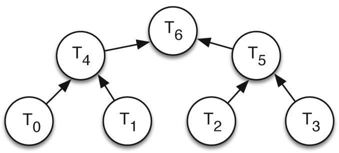

Fig. 2. Dependency tree of the diagonal blocks. Task T_j is associated with block A_{jj}. The leaf tasks are associated with the subgraphs of the leading block of the ND, while inner tasks are associated to separators.

Thus, the task-parallel version of ILUPACK partitions the original matrix into a number of decoupled blocks, and then delivers a partial IC factorization during the computation of (7), 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; see Fig. 3. Although the recursive definition of the preconditioner, shown in (2), 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 [4].

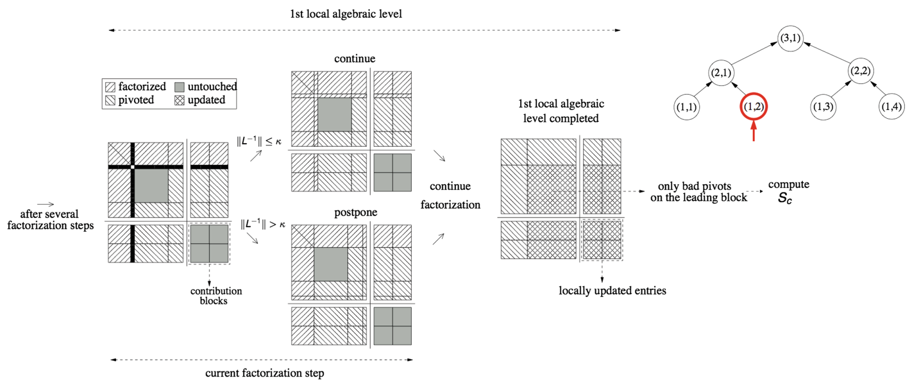

Fig. 3. A step of the Crout variant of the parallel preconditioner computations.

Application of the Preconditioner. As the definition of the recursion is maintained, the operations to apply the preconditioner, in (4), 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 PCG: once from bottom to top and a second time from top to bottom (with dependencies/arrows reversed in the DAG).

3 Proposal

In this section we present our strategy to enable GPU acceleration in the multicore version of ILUPACK. We analize two different approaches. The first one entirely off–loads the leaf tasks of the preconditioner application phase to the GPU, while the second one uses a threshold to use the GPU only when there is enough work to take advantage of the accelerator.

Our solution is designed for multicore platforms equipped with one GPU, using different streams to queue work that belongs to different tasks, but the idea is easily extensible to a multi-GPU context.

3.1 All Leafs in GPU, GPU$_{all}$

The task-parallel version of ILUPACK is based on a ND, that results in a *task tree* where only leaf tasks perform an important amount of work. Inner tasks correspond to the separator subgraphs in the ND process, and hence have much less work than their leaf counterparts. For this resason we only consider leaf tasks from here on.

The leaf tasks are independent from each other and can be executed concurrently provided sufficient threads were available. Therefore, we associate each of these tasks with a different GPU stream. Also, each task has its own data structure, both in CPU and GPU memory, containing the part of the multilevel preconditioner relevant to it, together with private CPU and GPU buffers. At the beginning of the application, where these buffers are allocated, our GPU-enabled versions make this memory non-pageable in order to perform asynchronous memory transferences between the CPU and the GPU.

For the GPU$_{all}$ version of the preconditioner application, the computation on each node of the DAG is based on the data-parallel version presented in [1]. It proceeds as described in Sect. 2.1, with the difference that, in this case, the forward and backward substitution are separated and spread upon the levels of the task-tree. Now, entering or leaving the recursive step in Eq. (2) 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 transfered back to the CPU buffers before entering the recursive step. This communication can be broken down into several asynchronous transfers from the device to pinned host memory, given the nature of the multilevel forward substitution. Furthermore, it can be overlapped almost entirely with other computations. 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, on which 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 to the CPU, making asynchronous transfers for each algebraic level of the preconditioner.

The computational cost of the preconditioner application corresponds mostly to two types of operations, the solution of $(LD^{\frac{1}{2}})$ and $(D^{\frac{1}{2}}L^T)$ linear systems and SpMVs. The rest of the operations involve vector scalings, reorderings, and substractions, which have relatively lower cost. We employ the CUSPARSE library kernels for the first two operations, while the lower cost operations (i.e. a diagonal scalings, vector permutations and a vector updates) are performed by *ad-hoc* kernels. The optimal block size for this kernels was determined experimentally, and was set to 512 threads.

This version aims to accelerate the computations involved by the leaf tasks while keeping a low communication cost, relying on the results obtained for the GPU acceleration of the serial version, and the streaming capabilities offered by the new GPU architectures. However, this version has serious drawbacks. The division of the work in various leaf tasks reduces the size of each independent linear system, and the multilevel ILU-factorization of the preconditioner produces levels of even smaller dimension. This can have a strong negative impact on the performance of massively parallel codes [8], and specifically on the CUSPARSE library kernels. It should be noted that the amount of data-parallelism available in the sparse triangular linear systems is severely reduced, leading to a poor performance of the whole solver. Additionally, the work assigned to the CPU in this variant is really minor, impeding the concurrent use of both devices.

3.2 Threshold Based Version, GPU$_{thres}$

In order to deal with the disadvantages of the previous version, we propose a threshold-based strategy, that computes the algebraic levels in the GPU until certain granularity, and the remaining levels in the CPU. This aims to produce two effects. On one hand, allowing the smaller and highly data-dependent levels to be computed on the CPU while the first levels, of larger dimension and higher data-parallelism, run on the GPU, implies that each operation is performed in the most convenient device. On the other hand, this strategy also improves the concurrent execution of both devices, increasing the overlap of the CPU and GPU sections of the code.

Regarding data transfer, in this approach the working buffer has to be brought to the CPU memory at some point of the forward substitution phase, and it has to be transferred to the GPU before the backward substitution of the upper triangular system ends. Moreover, these transfers are synchronous with respect to the current task or GPU stream, since the application of one algebraic level of the multilevel precondioner cannot commence until the results from the previous level are available.

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

4 Numerical Evaluation

In this section we summarize the experiments carried out to evaluate the performance of the proposal. Our primary goal is to assess the use of the GPU in the task-parallel version of ILUPACK. In order to do so, we compare our two GPU-accelerated versions with the original task-parallel ILUPACK, which exploits shared-memory parallelism via the OpenMP interface. All experiments reported were obtained using IEEE double-precision arithmetic.

4.1 Experimental Setup

The performance evaluation was carried out in a server equipped with an Intel(R) Xeon(R) CPU E5-2620 v2 of six physical cores, running at 2.10 GHz, with 132 GB of DDR3 RAM memory. The platform also features a Tesla K40m GPU (of the Kepler generation) with 2,880 CUDA Cores and 12 GB of GDDR5 RAM.

The CPU code was compiled with the Intel(R) Parallel Studio 2016 (update 3) using the `-O3` flag. The GPU compiler and the CUSPARSE Library correspond to version 6.5 of the CUDA Toolkit.

The benchmark utilized for the test is a s.p.d. case of scalable size derived from a finite difference discretization of the 3D Laplace problem. We generated 4 instances of different dimension; see Table 1. In the linear systems, the right-hand side vector b was initialized to $A(1,1,\ldots,1)^T$, and the preconditioned CG iteration was started with the initial guess $x_0 \equiv 0$. For the tests, the parameter that controls the convergence of the iterative process in ILUPACK, *restol*, was set to 10^8. The drop tolerance, and the bound to the condition number of the inverse factors, which control ILUPACK's multilevel incomplete factorization process, where set to 0.01 and 5 respectively.

Table 1. Matrices employed in the experimental evaluation.

Matrix	Dimension n	nnz	nnz/n
A126	2,000,376	7,953,876	3.98
A159	4,019,679	16,002,873	3.98
A171	5,000,211	19,913,121	3.98
A200	8,000,000	31,880,000	3.99
A252	16,003,008	63,821,520	3.99

4.2 Results

Each test instance was executed with 2 and 4 CPU threads with $f = 2$ and $f = 4$ respectively. The parameter f is related with the construction of the task tree. The algorithm that forms this tree relies on an heuristic estimation of the computational cost of each leaf task and divides a leaf into two whenever its correspondent subgraph has more edges than the number of edges of the whole graph divided by f. The parameter f is chosen so that, in general, there are more leaf tasks than processors. In [2,3] the authors recomend choosing a value between p and $2p$, where p is the number of processors.

Table 2 summarizes the structure of the multilevel preconditioner and the linear systems corresponding to leaf tasks that were generated using the aforementioned parameters. For each one of the tested matrices, the table presents the number of leaf tasks that resulted from the task tree construction for $f = 2$ and $f = 4$, and next to it shows the average dimension of the algebraic levels of the corresponding multilevel preconditioner, the average number of nonzeros, and the average row density of the levels, with their respective standard deviation. It can be easily observed that a higher value of f results in more leaf tasks of smaller dimension. Regarding the algebraic levels of the factorization, the table shows how the average dimension of the involved matrices decreases from one level to the next. It is important to notice how, in the second algebraic level, the submatrices already become about one third smaller in dimension, and have five times more non zero elements on each row. In other words, the subproblems become dramatically smaller and less sparse with each level of the factorization, causing that, in this case, only the first algebraic level is attractive for GPU acceleration.

Table 3 shows the results obtained for the original shared-memory version and the two GPU-enabled ones for the matrices of the Laplace problem. In the table, the total runtime of PCG, as well as the time spent on the preconditioner application stage and the SpMV are presented. The table also shows the number of iterations taken to converge to the desired residual tolerance, and the final relative residual error attained, which is calculated as

$$\mathcal{R}(x^*) := \frac{||b - Ax^*||_2}{||x^*||_2},$$

where x^* stands for the computed solution.

Table 2. Number of leaf tasks and average structure of each algebraic level of the preconditioner using $f = 2$ and $f = 4$. To represent the structure of the levels, the average dimension, the number of non-zeros and the rate of non-zeros per row is presented, toghether with the respective standard deviations.

Matrix	# th./f	# leaves	Level	Avg. n	$\sigma(n)$	Avg. nnz	$\sigma(nnz)$	$\frac{nnz}{n}$	$\sigma(\frac{nnz}{n})$
A159	2	3	0	1,006,831	345,798	6,193,794	2,183,862	6.1	0.1
			1	317,362	113,151	9,682,114	3,486,401	30.5	0.2
			2	2,875	736	10,099	2,014	3.6	0.6
	4	6	0	502,108	159,044	3,116,629	1,005,408	6.2	0.1
			1	156,048	50,647	4,685,500	1,537,905	30.0	0.2
			2	1,251	437	4,095	1,754	3.2	0.4
A171	2	2	0	1,881,030	16,604	11,421,390	123,868	6.1	0.1
			1	598,152	1,384	18,490,583	154,695	30.9	0.2
			2	6,304	984	23,444	7,247	3.7	0.6
	4	4	0	937,998	6,011	5,764,461	71,397	6.1	0.1
			1	294,702	1,310	8,967,985	72,180	30.4	0.2
			2	2,845	506	10,885	3,768	3.7	0.7
A200	2	3	0	2,003,212	795,192	12,207,556	4,592,834	6.1	0.1
			1	636,895	253,696	19,665,756	8,089,987	30.8	0.4
			2	6,466	3,316	23,189	12,912	3.5	0.2
	4	7	0	856,365	186,595	5,283,746	1,141,907	6.2	0.1
			1	268,523	595,53	8,155,375	1,842,559	30.3	0.2
			2	2,449	525	8,552	2,032	3.5	0.4
A252	2	3	0	4,004,955	1,694,044	24,271,087	9,856,575	6.1	0.1
			1	1,283,180	543,882	39,965,828	17,408,294	31.0	0.3
			2	14,762	7,162	57,168	28,744	3.8	0.1
	4	6	0	1,998,470	494,294	12,196,071	3,070,313	6.1	0.1
			1	635,612	159,942	19,603,140	4,936,718	30.8	0.1
			2	6,523	1,429	23,807	5,758	3.6	0.3

First, it should be noted that there are no significant differences, from the perspective of accuracy, between the task-parallel CPU variant and the GPU-enabled ones. Specifically, the three versions reach the same number of iterations and final relative residual error for each case, see Table 3.

From the perspective of performance it can be observed that, on one hand, GPU$_{all}$ only outperforms the multi-core version for the largest matrices (A252) and in the context of 2 CPU threads. This result was expected, as the GPU requires large volumes of computations to really leverage the device and hide the overhead due to memory transfer. On the other hand, GPU$_{thres}$ is able to accelerate the multi-core counterpart for all covered cases, see Fig. 4. This result reveals the potential benefit that arise from overlapping computations on both devices. Hence, even in cases where the involved matrices presented modest dimension, this version outperforms the highly tuned multi-core version. Additionally, the benefits related with the use of the GPU are similar for all matrices of each configuration, though the percentage of improvement is a bit higher for the smaller cases. This behavior is not typical for GPU-based solvers and one possible explanation is that the smaller cases are near to the optimal point (from the threshold perspective) while the largest cases are almost able to

Table 3. Runtime (in seconds) of the three task-parallel variants.

# threads	Matrix	Version	Iters	Total SpMV	Total prec	Total PCG	$\mathcal{R}(x^*)$
2	A159	CPU$_{omp}$	88	2.30	29.55	32.86	1.39E-08
		GPU$_{all}$			44.33	47.46	
		GPU$_{thres}$			20.46	23.83	
	A171	CPU$_{omp}$	97	3.07	39.43	43.87	1.52E-08
		GPU$_{all}$			48.02	52.36	
		GPU$_{thres}$			30.62	35.19	
	A200	CPU$_{omp}$	107	5.83	71.58	79.98	2.45E-08
		GPU$_{all}$			84.37	92.61	
		GPU$_{thres}$			47.73	56.26	
	A252	CPU$_{omp}$	131	13.86	175.66	195.67	3.23E-08
		GPU$_{all}$			153.48	173.62	
		GPU$_{thres}$			120.19	140.50	
4	A159	CPU$_{omp}$	88	1.30	22.72	24.55	9.96E-09
		GPU$_{all}$			44.82	46.40	
		GPU$_{thres}$			15.21	17.15	
	A171	CPU$_{omp}$	95	1.58	22.43	24.76	2.20E-08
		GPU$_{all}$			57.84	59.78	
		GPU$_{thres}$			17.50	19.87	
	A200	CPU$_{omp}$	108	3.13	40.34	45.03	1.06E-08
		GPU$_{all}$			108.37	112.41	
		GPU$_{thres}$			33.80	38.60	
	A252	CPU$_{omp}$	130	8.25	104.21	116.37	2.16E-08
		GPU$_{all}$			193.19	204.74	
		GPU$_{thres}$			90.05	104.60	

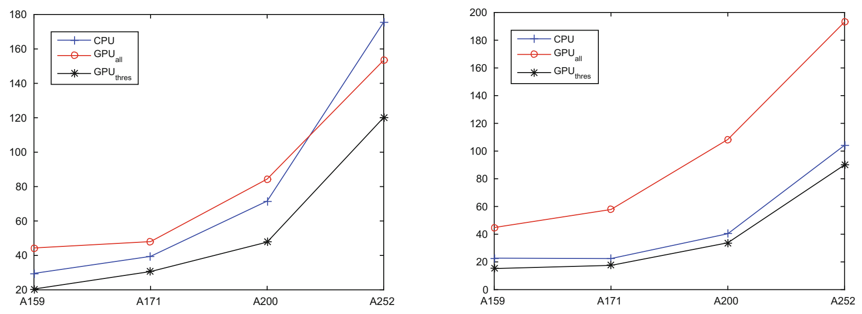

Fig. 4. Execution time (in seconds) of preconditioner application for the three task parallel variants, using two (left) and four (right) CPU threads. CPU version is the blue line with crosses. GPU$_{all}$ version is the red line with circles. GPU$_{thres}$ is the black line with stars. (Color figure online)

Table 4. Runtime (in seconds) of GPU$_{thres}$ adjusting the threshold to compute 1 and 2 levels in the GPU.

# threads	Matrix	GPU$_{1lev}$	GPU$_{2lev}$	GPU$_{all}$
2	A159	23.83	43.79	44.33
	A171	35.19	47.52	48.02
	A200	56.26	84.16	84.37
	A252	140.50	153.79	153.48
4	A159	17.15	44.54	44.82
	A171	19.87	57.21	57.84
	A200	38.60	108.70	108.37
	A252	104.60	185.72	193.19

compute 2 levels in GPU. This can be noticed in Table 4, were we add a variant that computes the first 2 levels on the accelerator. As the multilevel factorization generates only 3 levels, with the third one very small with respect to the other two, it is not surprising that the runtimes of this version are almost equivalent to those of GPU$_{all}$. The table shows how the penalty of computing the second level in the GPU decreases as the problem dimension grows.

Finally, GPU$_{thres}$ also offers higher performance improvements for the 2-threads case than for its 4-threads counterpart.

5 Final Remarks and Future Work

In this work we have extended the task-parallel version of ILUPACK so that leaf tasks can exploit the data-parallelism of the operations that compose the application of the multilevel preconditioner, i.e. SpMV and the solution of triangular linear systems, along with some minor vector operations. We presented two different GPU versions, one that computes the entire leafs in the accelerator (GPU$_{all}$) and an alternative that employs a threshold to determine if a given algebraic level of the preconditioner presents enough granularity to take advantage of the GPU (GPU$_{thres}$). Both variants are executed on a single GPU, asigning a GPU stream to each independent leaf task.

The experimental evaluation shows that the division of the workload in smaller tasks makes difficult the extraction of enough data-parallelism to fully occupy the hardware accelerator, and this results in poor performance for GPU$_{all}$. However, GPU$_{thres}$ is able to execute each operation in the most convenient device while mantaining a moderate communication cost, outperforming the original multicore version for all the tested instances.

As part of future work we plan to advance towards the GPU acceleration of the distributed task-parallel version of ILUPACK. An intermediate step of this process involves the study of integrating a multi-GPU scenario in the current task parallel versions. Additionally, we plan to develop a mathematical model for the GPU-offload threshold, which was determined empirically in the present work.

Acknowledgments. The researchers from the *Universidad Jaime I* were supported by the CICYT project TIN2014-53495R of The researchers from *UdelaR* were supported by PEDECIBA and CAP-UdelaR Grant.

References

1. Aliaga, J.I., Bollhöfer, M., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S.: Leveraging data-parallelism in ILUPACK using graphics processors. In: 2014 IEEE 13th International Symposium on Parallel and Distributed Computing, pp. 119–126. IEEE (2014)
2. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Parallelization of multilevel preconditioners constructed from inverse-based ILUs on shared-memory multiprocessors. Parallel Comput. Archit. Algorithms Appl. **38**, 287–294 (2007)
3. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Design, tuning and evaluation of parallel multilevel ILU preconditioners. In: Palma, J.M.L.M., Amestoy, P.R., Daydé, M., Mattoso, M., Lopes, J.C. (eds.) VECPAR 2008. LNCS, vol. 5336, pp. 314–327. Springer, Heidelberg (2008). doi:10.1007/978-3-540-92859-1_28
4. 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 Comput. **37**(3), 183–202 (2011)
5. Aliaga, J.I., Bollhöfer, M., Martín, A.F., Quintana-Ortí, E.S.: Parallelization of multilevel ILU preconditioners on distributed-memory multiprocessors. In: Jónasson, K. (ed.) PARA 2010. LNCS, vol. 7133, pp. 162–172. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28151-8_16
6. Barrett, R., Berry, M.W., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Van der Vorst, H.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, vol. 43. SIAM, New Delhi (1994)
7. George, T., Gupta, A., Sarin, V.: An empirical analysis of the performance of preconditioners for SPD systems. ACM Trans. Math. Softw. **38**(4), 24:1–24:30 (2012)
8. Kirk, D.B., Hwu, W.W.: Programming Massively Parallel Processors: A Hands-on Approach. Newnes, Boston (2012)
9. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM Publications, New Delhi (2003)
10. Schenk, O., Wächter, A., Weiser, M.: Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. SIAM J. Sci. Comput. **31**(2), 939–960 (2009)
11. Schenk, O., Bollhöfer, M., Römer, R.A.: On large scale diagonalization techniques for the anderson model of localization. SIAM Rev. **50**, 91–112 (2008)