SIAM J. SCI. COMPUT.
Vol. 38, No. 1, pp. C53–C72

A FACTORED SPARSE APPROXIMATE INVERSE PRECONDITIONED CONJUGATE GRADIENT SOLVER ON GRAPHICS PROCESSING UNITS*

MASSIMO BERNASCHI†, MAURO BISSON†, CARLO FANTOZZI‡, AND CARLO JANNA§

Abstract. Graphics Processing Units (GPUs) exhibit significantly higher peak performance than conventional CPUs. However, in general only highly parallel algorithms can exploit their potential. In this scenario, the iterative solution to sparse linear systems of equations could be carried out quite efficiently on a GPU as it requires only matrix-by-vector products, dot products, and vector updates. However, to be really effective, any iterative solver needs to be properly preconditioned and this represents a major bottleneck for a successful GPU implementation. Due to its inherent parallelism, the factored sparse approximate inverse (FSAI) preconditioner represents an optimal candidate for the conjugate gradient–like solution of sparse linear systems. However, its GPU implementation requires a nontrivial recasting of multiple computational steps. We present our GPU version of the FSAI preconditioner along with a set of results that show how a noticeable speedup with respect to a highly tuned CPU counterpart is obtained.

Key words. preconditioning, approximate inverses, parallel computing, iterative methods

AMS subject classifications. 65F08, 65F10, 65F50, 65Y05, 65Y10, 65Y20

DOI. 10.1137/15M1027826

1. Introduction.

The solution of large and sparse linear systems of equations is a central issue in many scientific and engineering applications. For instance, several simulation codes, both in academy and industry, use finite difference or finite element schemes to solve PDE problems, with the solution of the resulting linear system of equations representing one of the most expensive tasks in terms of both memory and CPU time. The size of the systems that need to be solved is rapidly growing as scientists and engineers require increasingly accurate results, with the use of parallel computers becoming quite mandatory in several application domains. For the above reasons, the development of parallel linear solvers is a very active research field with several contributions coming from both numerical analysis and computer science. While most of the research focused so far on algorithms suitable for traditional CPUs, it is by now apparent that the higher peak performance of graphics processing units (GPUs) should be better exploited. As a matter of fact, several works [10, 29, 35] showed that conjugate gradient–like (CG) algorithms based on the sparse matrix-by-vector (SpMV) product can obtain noticeable speedup on a GPU with respect to a CPU. Since the SpMV kernel is usually bandwidth limited, its performance strongly depends on the storage format, which in turn is strictly related to the number and distribution of nonzero elements of the matrix. There is no gen-

*Submitted to the journal's Software and High-Performance Computing section June 25, 2015; accepted for publication (in revised form) November 6, 2015; published electronically January 26, 2016. This study has been supported by the ISCRA project "PARPREC: Parallel Preconditioners for Advanced Engineering Applications."

http://www.siam.org/journals/sisc/38-1/M102782.html

†Institute for Computing Applications, CNR, Roma, Italy (m.bernaschi@iac.cnr.it, mauro.bis@gmail.com).

‡Department of Information Engineering, University of Padova, Via Gradenigo 6/b, 35131 Padova, Italy (carlo.fantozzi@unipd.it).

§Department ICEA, University of Padova, Via Trieste 63, 35121 Padova, Italy (carlo.janna@unipd.it).

eral rule; usually for matrices exhibiting a regular nonzero structure, the ELLPACK format gives good performance, while for more irregular ones compressed sparse row (CSR) or hybrid formats are more suited [3]. Some effort has been recently made in developing more general formats [24, 25]. However, the effectiveness of any iterative solver strongly depends on the availability of a suitable preconditioner, that is, an approximation of A^{-1} which has to be relatively cheap to compute and apply to a vector. The development of effective preconditioners on GPUs is not at all straightforward. A wide review of the possible alternatives for GPUs can be found in [27]. Incomplete LU factorization (ILU) is a popular class of algebraic preconditioners usually providing good performance in a wide variety of problems. However, both their computation and application to a vector are sequential in nature and difficult to port on GPUs. Some degree of parallelism can be exploited by level-scheduling, though at the expense of the effectiveness. Very recently a completely asynchronous ILU preconditioner was proposed in [34]. The authors gain parallelism in the set-up phase by iteratively solving a nonlinear problem associated to the factorization and by introducing approximate solves. The method seems very promising, though its feasibility in real-world problems has not been deeply explored yet. More suitable techniques may be represented by polynomial [36] and approximate inverse preconditioners [4, 5, 13], as their application to a vector requires only SpMV products. Unfortunately, polynomial preconditioners usually require a large number of iterations to converge in tough problems, making approximate inverse preconditioners the most attractive choice on this innovative hardware. Several experiments with approximate inverses have been presented [7, 12, 14, 33]. They show how, even though CG exhibits a pretty good performance in the iteration stage, a major bottleneck in the execution of such methods on GPU resides in the set-up phase. As a consequence, it is quite common to let the CPU compute the preconditioner before starting the execution of the CG on the GPU.

To overcome these difficulties we resort to the factored sparse approximate inverse (FSAI) preconditioner, originally proposed in [22, 21] and improved over the years with several contributions [16, 17, 20, 23, 26]. FSAI has been used on GPUs in a previous work [39]. However, all the set-up operations remained on the CPU and only the iteration stage was ported on the GPU. FSAI gives an explicit approximation of A^{-1} in factored form and its main advantage is the very high degree of parallelism exposed by its computation structure. As a matter of fact, every FSAI row can be computed independently from the others with no data dependency at all. The main difficulty is defining a data mapping and a memory access pattern that maximizes the exploitation of the available bandwidth reducing, at the same time, thread conflicts that may slow down the computation. Hereafter, we show how all main computational and basic dense linear algebra kernels have been significantly modified with respect to the original CPU version. Coding the iteration stage of CG is significantly easier, because we rely on the cuSPARSE library provided by NVIDIA [32]. In its original form, FSAI was presented as a preconditioner for symmetric and positive definite (SPD) linear systems and later generalized to nonsymmetric problems in [40]. As the main computational kernels of both variants are very similar, in this work we limit our attention to the SPD case.

The rest of the paper is organized as follows. In the next section we describe the features of GPU architectures, with particular reference to NVIDIA GPUs, and highlight the main challenges in exploiting them. In section 3 we give a brief overview of the FSAI preconditioner and the numerical techniques aimed at increasing its

performance. Section 4 provides a detailed description of the algorithms developed to port the FSAI set-up entirely on NVIDIA GPUs. Numerical results on the performance of the proposed GPU implementation on some large-size benchmarks are given in section 5. Finally, we close the paper with some concluding remarks and ideas for future activities.

2. Architecture of GPUs. A GPU is a specialized processor originally designed to accelerate graphics rendering. In the first half of the last decade, evidence began accumulating [8, 28, 37] that GPUs can be used as effective accelerators in many general-purpose computing applications. The reason why a GPU provides higher peak performance than a CPU is that it trades flexibility for speed. The micro-architecture of a GPU is massively parallel, with thousands of simple cores designed for simultaneous, independent, identical computations on multiple data inputs. Yet, each core can perform only a restricted number of operations, and there are limitations on the operands and flow control as well. If all the limitations can be met, and if the structure of the computation exposes massive fine-grained parallelism to match the parallel architecture of the GPU, the computation itself can be performed with a level of efficiency unknown to general-purpose CPUs. The main drawback of this approach is that the software implementation must be carefully tailored to the micro-architecture, which not only varies from vendor to vendor but also improves incrementally over time for any given vendor. Moreover, any micro-architecture is partly hidden by the vendor's programming framework, which, in turn, evolves over time. In what follows, we will provide some additional details on the micro-architecture of GPUs—and on their limitations thereof—which are required to understand the key issues we faced with the FSAI preconditioner and CG solver. We will concentrate on the micro-architecture of NVIDIA Kepler GPUs as exposed through the CUDA software framework [31], version 6, since this is the solution used in our study.

From a software perspective, a CUDA program is partitioned into *grids*; in turn, each grid is split into *blocks*, and each block contains a certain number of *threads*. A function executed on the GPU is called a *kernel*. A kernel is typically executed in parallel by multiple threads, or even multiple blocks, in an SPMD (single program, multiple data) fashion. Blocks and threads can be both organized into regular, multi-dimensional arrays, so that kernels can be easily invoked across commonplace domains such as vectors or matrices.

CUDA's software hierarchy naturally maps to the underlying hardware hierarchy: in particular, threads are executed on hardware cores. Since thousands of cores are available, thousands of threads must be running at any time to keep the hardware units busy. This is the main reason why a CUDA (actually, any GPU) program must be highly parallel. In general, exposing parallelism is not a trivial task, and there may be theoretical limits to the amount of parallelism that can be extracted from a given computation [2]. Also critical to performance is a balanced workload: all threads – or, at least, the threads belonging to a block—must perform the same amount of work, so that no core remains partly idle. To the same aim, synchronization among threads—for instance, via mutually exclusive (*atomic*) instructions—must be kept to a minimum. Satisfying requirements on thread and blocks is made additionally convoluted by some hardware limitations. First of all, there is a limit to the number of threads per block. Furthermore, threads are executed in groups of 32 called *warps*: performance is significantly improved if threads in the same warp execute the same code with no branch divergence, and they access memory according to certain patterns.

As far as memory is concerned, any GPU accelerator benefits from a dedicated, high-speed memory hierarchy that is separated from the hierarchy of the host CPU. In NVIDIA GPUs, prominent levels of the hierarchy are the *global memory*, accessible by all threads and by the host CPU as well, and the *shared memory*, shared among threads within the same block. Shared memory can be used for communication and synchronization among threads, but it can also be used as a fast scratchpad. A portion of the global memory, called *local memory*, houses "spilled registers" when threads of a block require more registers than can be accommodated in hardware. Starting with CUDA 6.0, all memory copy operations can be delegated to the CUDA runtime, which automatically migrates data between the host and the GPU accelerator. In practice, the cost of memory operations is so high, and the overall structure of the memory hierarchy so complex, that a nonnegligible degree of manual intervention is required to attain maximum performance. From a performance standpoint, memory is a bottleneck—particularly so on GPUs, where a higher fraction of transistors is devoted to computing than on CPUs: as a consequence, the key issue on a GPU is to feed enough data to its numerous computing units, and keep them active. A thorough discussion on memory issues is beyond the scope of this article; it suffices to say that memory access time is highly dependent on the memory access pattern because of the underlying hardware structure. Memory is organized into banks at all levels: bank contention can disrupt performance, to the point where a shared memory access can be slower than a global memory access [30]. On the contrary, performance peaks if *coalescence* is possible, that is, if parallel threads may access consecutive memory locations. Data structures must be arranged, and data padded, according to a *structure-of-arrays* pattern instead of an *array-of-structures* one as in a standard CPU. Another opportunity for optimization lies in the fact that memory transfers can be asynchronous and overlapped with computation. For instance, the NVIDIA Kepler micro-architecture introduced a new warp-level operation called the *shuffle*: this feature allows the threads of a warp to exchange data without passing through shared (or global) memory, and at a lower latency.

As can be seen, the micro-architecture of GPUs makes software optimization a delicate task, with several details to take into account. Below, we will illustrate how the considerations exposed in the present section have been incorporated into our CUDA code for the CG solver, and chiefly in the FSAI preconditioner. To this aim, in the next section we first recall what an FSAI preconditioner is.

3. The FSAI preconditioner. Introduced in [22], the original FSAI preconditioner M^{-1} for a symmetric positive definite matrix A can be written as

$$M^{-1} = G^T G \simeq A^{-1}, \tag{3.1}$$

where G is computed by minimizing the Frobenius norm

$$\|I - GL\|_F \tag{3.2}$$

over the set $\mathcal{W}_{\mathcal{S}}$ of matrices having a prescribed lower triangular nonzero pattern $\mathcal{S}$. The matrix L appearing in (3.2) is the exact lower triangular factor of A. The key of the feasibility of FSAI in a parallel setting is that L is not required to get G. In fact, differentiating (3.2) with respect to the G entries g_{ij} and setting to zero gives

$$[GA]_{ij} = \begin{cases} 0, & i \neq j, \quad (i,j) \in \mathcal{S}, \\ l_{ii}, & i = j, \end{cases} \tag{3.3}$$

where $[\cdot]_{ij}$ is the entry in row i and column j of the matrix between square brackets, and l_{ii} is the ith diagonal element of L. Since L is unknown, l_{ii} in (3.3) is replaced by 1 and the matrix $\widetilde{G}$ is computed instead by solving

$$[\widetilde{G}A]_{ij} = \delta_{ij}, \tag{3.4}$$

where δ_{ij} is the Kronecker delta. From a practical viewpoint, to compute the ith row of $\widetilde{G}$, say $\widetilde{\boldsymbol{g}}_i$, we define the set $\mathcal{P}_i$ of all the column indices belonging to the ith row of $\mathcal{S}$, that is,

$$\mathcal{P}_i = \{j : (i,j) \in \mathcal{S}\}, \tag{3.5}$$

then we solve the linear system

$$A[\mathcal{P}_i, \mathcal{P}_i]\boldsymbol{w} = \boldsymbol{e}_{m_i}, \tag{3.6}$$

where $A[\mathcal{P}_i, \mathcal{P}_i]$ is the dense matrix obtained by collecting the entries of A having row/column index in $\mathcal{P}_i$ and $\boldsymbol{e}_{m_i}$ is the m_ith basis vector of $\mathbb{R}^{m_i}$, with $m_i = |\mathcal{P}_i|$. After solving (3.6), the entries of $\boldsymbol{w}$ are scattered into the nonzero positions of $\widetilde{\boldsymbol{g}}_i$. In the end, the final G is obtained by scaling $\widetilde{G}$ as

$$G = D\widetilde{G} \qquad \text{with} \qquad D = [\mathrm{diag}(\widetilde{G})]^{-1/2} \tag{3.7}$$

in such a way that the diagonal entries of the preconditioned matrix GAG^T have value 1. This condition guarantees that G is the unique minimizer of the Kaporin number over all the matrices $B \in \mathcal{W}_{\mathcal{S}}$ [21]. Defined as the ratio between the arithmetic and geometric mean of the eigenvalues of an SPD matrix, the Kaporin number gives a measure for the preconditioned conjugate gradient (PCG) convergence rate. The FSAI preconditioner is very robust as it can be computed for any choice of the nonzero pattern $\mathcal{S}$. Moreover, it is always SPD by construction.

However, the FSAI effectiveness strongly depends on the choice of $\mathcal{S}$, which is not an easy task. There are ways to estimate $\mathcal{S}$ statically, i.e., $\mathcal{S}$ is determined before $\widetilde{G}$ is computed, or adaptively, i.e., while $\widetilde{G}$ is computed: see, e.g., [20] for a thorough review. Usually, nonzero patterns improved adaptively during the FSAI set-up are more effective, but the required procedure is by far more complex. Since an experienced user can obtain good results even with static patterns, we restrict our attention to static FSAI only and leave the GPU implementation of adaptive FSAI to a future paper. The problem of choosing good a priori patterns for the inverse of a matrix has been already investigated by several authors [15, 9]. The best way to guess a nonzero pattern for an approximation of A^{-1} is probably choosing the pattern of the $\tilde{A}^k$, i.e., the kth power of the sparsified matrix $\tilde{A}$, which is obtained by dropping the off-diagonal entries a_{ij} of A satisfying

$$|a_{ij}| \leq \tau\sqrt{a_{ii}a_{jj}} \tag{3.8}$$

for some user-specified parameter $\tau \in [0,1]$. However, in our case we are interested in estimating the pattern of L^{-1}, with L itself unknown. In this work we adopt the strategy suggested in [16] and successfully implemented also in [20], where k steps of the simple recursion,

$$B_{p+1} \leftarrow \mathrm{Low}(B_p\ \tilde{A}) \qquad p = 0, 1, \ldots, k-1, \tag{3.9}$$

are used to predict the pattern of G, starting from $B_0 = I$. In (3.9), the matrix-by-matrix products are carried out only symbolically and $\mathrm{Low}(\cdot)$ is the function returning the lower triangular part of a matrix. Unfortunately, even with a good nonzero pattern several relatively small entries of G remain that burden the preconditioner application without improving the convergence rate. For such a reason, it is advisable to postfilter G to eliminate such dead load [23]. However, a lightweight preconditioner $\widehat{G}$ cannot be obtained by simply neglecting small entries of G, otherwise the favorable condition

$$\mathrm{diag}(\widehat{G}A\widehat{G}^T) = I \tag{3.10}$$

would no longer be satisfied. The most effective way to post-filter G is the following. Let us consider $\boldsymbol{g}_i$, the ith row of G, and decompose it as

$$\boldsymbol{g}_i = \boldsymbol{z}_i + \boldsymbol{\epsilon}_i \tag{3.11}$$

in such a way that all the $\boldsymbol{g}_i$ entries smaller than $\delta\|\boldsymbol{g}_i\|_2$ are stored in ϵ_i for some user-specified $\delta \in [0, 1]$. Due to (3.7), we can write

$$1 = [GAG^T]_{ii} = (\boldsymbol{z}_i + \boldsymbol{\epsilon}_i)^T A(\boldsymbol{z}_i + \boldsymbol{\epsilon}_i) = \boldsymbol{z}_i^T A\boldsymbol{z}_i + 2\boldsymbol{\epsilon}_i^T A\boldsymbol{g}_i - \boldsymbol{\epsilon}_i^T A\boldsymbol{\epsilon}_i. \tag{3.12}$$

The observation that $\boldsymbol{\epsilon}_i^T A\boldsymbol{g}_i = 0$ by construction because of (3.4) allows us to evaluate the scaling factor for $\boldsymbol{z}_i$ as

$$\widehat{d}_{ii} = \frac{1}{\sqrt{1 + \boldsymbol{\epsilon}_i^T A\boldsymbol{\epsilon}_i}}, \tag{3.13}$$

which is directly included in the G factor during set-up. Notice that the computation of $\boldsymbol{\epsilon}_i^T A\boldsymbol{\epsilon}_i$ can be easily carried out by defining the set $\mathcal{E}_i$ collecting all the column indices of $\boldsymbol{\epsilon}_i$, and then gathering the entries of $A[\mathcal{E}_i, \mathcal{E}_i]$ and performing a dense matrix-by-vector product followed by a dense scalar product. It is worth saying that (3.13) adds little overhead in the set-up phase only, which is largely offset during iteration as in some cases the above procedure offers the chance of neglecting most of the preconditioner nonzeroes. The overall procedure to compute a static FSAI preconditioner for an SPD matrix A is summarized in Algorithm 3.1. It can be easily recognized that this algorithm shows a potential high degree of parallelization in the sparse matrix by sparse matrix product in line 4 and in the two loops in lines 7–12 and 13–22. However, a finer-grained parallelism needs to be exposed to fully exploit the computing power of GPUs. While dropping elements of A and G according to the tolerances τ and δ is a relatively easy task to be performed on GPU, there are other kernels such as

- the symbolic sparse matrix by sparse matrix product used in the static pattern construction,
- the gathering of small dense linear systems $A[\mathcal{P}_i, \mathcal{P}_i]$ and $A[\mathcal{E}_i, \mathcal{E}_i]$,
- the solution of small dense linear systems,

that need to be completely redesigned. Once all of these kernels are optimized for the GPU hardware, linking them together is relatively easy since it requires only a few procedures to convert from a given storage format to another.

4. GPU implementation of FSAI. In this section we describe the design of the specialized kernels developed to carry out the most computationally intensive phases of the FSAI set-up. The GPU device used for this development is an NVIDIA Tesla K80 board containing 2 Kepler GK210 GPUs, each equipped with 2496 cores

ALGORITHM 3.1. STATIC FSAI COMPUTATION.

Input: Set-up parameters: $\tau \geq 0$, $k \geq 1$, $\delta \geq 0$
Input: $A_{n\times n}$
Output: $G_{n\times n}$

Static pattern computation

1. Generate $\tilde{A}$ by dropping the entries $a_{ij} \leq \tau\sqrt{a_{ii}a_{jj}}$
2. $B_0 \leftarrow I$
3. **for** (p = 0; p < k-1; p++) **do**
4. $\quad B_{p+1} \leftarrow \text{Low}(B_p\tilde{A})$
5. **end for**
6. Store into $\mathcal{S}$ the nonzero pattern of B_k

Compute the nonzero entries of G

7. **for** (i = 0; i < n; i++) **do**
8. $\quad$ Gather $A[\mathcal{P}_i, \mathcal{P}_i]$ from A
9. $\quad$ Solve $A[\mathcal{P}_i, \mathcal{P}_i]\boldsymbol{w} = \boldsymbol{e}_{m_i}$
10. $\quad \boldsymbol{w} \leftarrow \boldsymbol{w}/\sqrt{w_{m_i}}$
11. $\quad$ Scatter $\boldsymbol{w}$ into $\boldsymbol{g}_i$
12. **end for**

Filter the nonzero entries of G

13. **for** (i = 0; i < n; i++) **do**
14. $\quad$ **for** (j = 0; j < i; j++) **do**
15. $\quad\quad$ **if** $\boldsymbol{g}_{i,j} > \delta\|\boldsymbol{g}_i\|_2$ **then**
16. $\quad\quad\quad \boldsymbol{z}_{i,j} \leftarrow \boldsymbol{g}_{i,j}$
17. $\quad\quad$ **else**
18. $\quad\quad\quad \boldsymbol{\epsilon}_{i,j} \leftarrow \boldsymbol{g}_{i,j}$
19. $\quad\quad$ **end if**
20. $\quad\quad \boldsymbol{g}_i \leftarrow \frac{\boldsymbol{z}_i}{\sqrt{1+\boldsymbol{\epsilon}_i^T A\boldsymbol{\epsilon}_i}}$
21. $\quad$ **end for**
22. **end for**

and 12 GB of GDDR5 memory. The performance obtained with the GPU kernels is compared to that of a *state-of-the-art* CPU implementation of the same kernels on an Intel Xeon E5645 processor with 6 cores running at 2.40 GHz and 72 GB of DDR3 memory. These latter CPU kernels have been borrowed from FSAIPACK, which provides a wide set of functions for the FSAI computation on shared memory machines [20].

4.1. Kernel for static pattern generation. This kernel computes the nonzero pattern for G by performing k steps of recursion (3.9). The basic procedure is reported in Algorithm 4.1, where $\text{Low}(\cdot)$ returns the lower triangular part of the matrix specified as argument and $1_{NZ}(\cdot)$ returns the matrix in input with nonzeroes replaced by 1's.

Since in each iteration the reference matrix A is multiplied by the matrix computed in the previous step, we developed a CUDA kernel for the core matrix-matrix product

$$L^{out} \leftarrow \text{Low}(L^{in} \cdot A)$$

ALGORITHM 4.1. COMPUTATION OF MATRIX PATTERN.
Input: Set-up parameter: $k \geq 1$
Input: $A_{n \times n}$
Output: $1_{NZ}(L^k)$

1. $L^1 \leftarrow \text{Low}(A)$
2. **for** (i = 2; i < k+1; i++) **do**
3. $\quad L^i \leftarrow \text{Low}(L^{i-1} \cdot A)$
4. **end for**

that is launched $k-1$ times. In this way, each launch acts as a global synchronization mechanism between subsequent products.

The kernel uses different data structures to store different matrices. Since the matrix A is accessed rowwise (see below) in read-only mode, it is stored in CSR format so that only the nonzeroes are stored in memory and each row is stored in consecutive memory locations. Moreover, since the pattern computation requires the locations of the non-zeroes rather than their values, the CSR is represented with only the row offset and the column index arrays (nonzero values are not stored).

The matrices L^{in}/L^{out} are implemented according to a double buffering scheme so that the output matrix in each step becomes the input for the next step. In order to accommodate for the variable number of nonzeros resulting from each product and to support a number of recursions as large as possible, the two matrices are stored in ELLPACK format with a pad size, denoted as *maxcol*, determined by the amount of available memory (less than or equal to n). Analogously to the CSR matrix, also for the ELLPACK matrices we do not store nonzero values. Details on these sparse matrix storage formats can be found in [3].

If during a product one row of the output matrix requires more elements than the maximum allowed, an error condition is signaled. The mechanism is simple (but effective) and relies on a global memory flag that is set by any thread that would write beyond the pad limit. After the kernel execution, the flag is copied to host memory to check for errors and, if any, the computation is interrupted.

The kernel performs the $\text{Low}(L^{in} \cdot A)$ operation following a warp-centric approach. It is launched with the maximum number of warps that can be simultaneously active on all the multiprocessors and each warp computes a block of n/N_{warp} rows of the output matrix L^{out}. Since we are only interested in the location of nonzeros of the product matrix (no floating point operation is performed) each row is computed as the union of a collection of sets. The warp in charge of the rth row appends to $L^{out}_{r,*}$ the column indices less than or equal to r found in the CSR matrix A at the rows indexed by the entries in $L^{in}_{r,*}$. More precisely, considering the rows of the matrices as sets of column indices corresponding to nonzeroes, row r of the output matrix is computed as

$$L^{out}_{r,*} = \bigcup_{s \in L^{in}_{r,*}} \Big\{ c \in A_{s,*} \mid c \leq r \Big\}.$$

We avoid repetitions in the output rows by using a map with n elements for each warp in order to keep track of elements already appended. These maps form an additional matrix used by the kernel, the *map matrix*.

Algorithm 4.2 shows a pseudocode for the kernel. It requires the CSR and ELLPACK representation of A and L^{in}, respectively, as input and returns L^{out} in ELL-

PACK format as output. It is launched with N_{warp} warps and each warp processes n/N_{warp} rows of the input ELLPACK matrix L^{in} (loop at line 1) producing the corresponding output rows in L^{out}. Rows are processed in groups of 32 indices (loop at line 4) and, for each group, the warp in charge computes in parallel the set union of the indices in the corresponding rows of the CSR. Groups are read one element per thread (lines 5–7) and are processed sequentially (line 8). At each iteration, one thread broadcasts its element to the warp (line 12, lines 9–11 handle final groups with fewer than 32 elements) for the parallel processing of the corresponding row in the CSR matrix. Also for the CSR, rows are processed in chunks of 32 elements in search of new column-indices to be appended to the current output row (loop at line 13).

To that purpose, each thread computes a warp-wide mask of the threads that read an index that must be appended, i.e., a column index less than or equal to the index of the current row r whose entry in the map matrix is not set (lines 14–17). Each masked thread then sets the map matrix entry corresponding to its column index (line 19) and appends it to the output row at an offset equal to the number of masked threads with lower lane ID (that is the thread ID within the warp according to the CUDA jargon, line 21). In that way threads perform an efficient warp-local *compact* operation of the CSR rows without using *atomic* instructions. In case the row would grow over the maximum allowed size, the global error flag is set so that, after the execution, host code can invalidate the output of the kernel (line 23). After the append, each thread updates the current size of the output row by the number of appended elements (line 26) and the next chunk of the CSR row is processed. When the processing of the current L^{in} row is completed, the map matrix row associated to the warp is reset for the next iteration (lines 30–32) and the final row size is stored in the length array of the output matrix (line 33). In order to perform the map reset efficiently, only those elements that have actually been set are reset to zero.

4.2. Performance of the static pattern generation on GPU. All data structures but the map matrix are implemented by using 4-byte integers. The map matrix is implemented as a byte-map instead of a bit-map in order to avoid the use of *atomic* instructions to modify its elements. Since this matrix has a limited number of rows, the trade-off between the increase of memory requirements and the increase in access efficiency is definitely favorable. Each of the 13 SMX of our Kepler K80 GPU supports up to 2048 active threads for a total of $64 \times 13 = 832$ concurrently active warps. For this test we chose the sparse matrix Cube3D (see Table 4) that has $190,581$ rows and $7,531,389$ nonzeroes (with a minimum and maximum number of nonzeroes per row equal to 12 and 96); thus the device memory footprint of the code is the following: the CSR matrix (A_{row}, A_{col}) requires about 30 MB, whereas the map matrix (M) requires 150 MB ($190,581$ single-byte entries for each one of the 832 warps). Although CUDA offers a dynamic memory management within kernels, the corresponding overhead can seriously affect performance. To avoid the problem, we reserve directly all available memory to the ELLPACK matrices (L^{in}, L^{out}) in order to support the maximum number of recursion for the static pattern. With the memory available on the K80, it is possible to use a pad size of 6944, for a total of 9.8 GB. That memory is obviously released when it is no longer required.

The kernel is compiled with CUDA 6.5 and, without specific directives, requires 38 registers per thread. In order to maximize occupancy, the launch bounds configuration has been tweaked specifying at most 128 thread per block and at least 16 blocks per multiprocessor. The introduced register pressure allows US to drop the registers per

ALGORITHM 4.2. CUDA WARP-CENTRIC $L^{out} \leftarrow \text{Low}(L^{in} \cdot A)$.

Input: CSR matrix: $A_{row}[n]$, $A_{col}[nnz_A]$
Input: ELLPACK matrix: $L^{in}[n][maxcol]$, $len^{in}[n]$
Work: Map matrix: $M[N_{warp}][n]$
Output: ELLPACK matrix: $L^{out}[n][maxcol]$, $len^{out}[n]$
Output: Error flag: GError

```
1.  for (r = wid; r < n; r += N_warp) do
2.      rind = 0
3.      ilen = len^in[r]
4.      for (i = 0; i < ilen; i += 32) do
5.          if (i+lid) < ilen then
6.              myrow = L^in[r][i + lid]
7.          end if
8.          for (j = 0; j < 32; j++) do
9.              if (i+j) ≥ ilen then
10.                 break
11.             end if
12.             cur_row = __shfl(myrow, j)
13.             for (k = A_row[cur_row]; k < A_row[cur_row+1]; k += 32) do
14.                 c = ((k+lid) < A_row[cur_row+1]) ? A_col[k+lid] : (wid+1)
15.                 t = (c ≤ r)&&(!M[wid][c])
16.                 z = __ballot(t);
17.                 l = __popc(z & ((1<<lid)-1))
18.                 if t == True then
19.                     M[wid][c] = 1
20.                     if (rind+l) < maxcol then
21.                         L^out[r][rind+l] = c
22.                     else
23.                         GError = 1
24.                     end if
25.                 end if
26.                 rind += __popc(z)
27.             end for
28.         end for
29.     end for
30.     for (i = lid; i < rind; i += 32) do
31.         M[wid][L^out[r][i]] = 0
32.     end for
33.     if lid == 0 then
34.         len^out[r] = rind
35.     end if
36. end for
37. GError = 0
```

thread count to 32 but resulted in the allocation of 8 bytes into local memory. The net effect, however, is an increase in performance of 8.4% with respect to the untweaked version.

TABLE 1

Execution times, on GPU and CPU, respectively, and number of nonzeroes of the output matrix obtained running the kernel for the sparse matrix Cube3D with $190,581$ *rows and* $7,531,389$ *nonzeroes varying the number of steps* k *with no preliminary sparsification, i.e., setting* $\tau = 0$.

k	Output nnz	GPU time (ms)	CPU time (ms)
2	19,880,688	17.7	355.2
3	60,114,684	98.0	1,450.9
4	134,573,295	351.6	7,920.1
5	250,615,200	701.0	11,818.4
6	413,949,594	1,448.7	33,771.2
7	628,388,562	2,711.5	79,311.8

TABLE 2

Comparison of the running time required for systems gather on a K80 GPU and an Intel Xeon E5645 at 2.40 GHz (6 cores) for a set of test cases. The table also provides the sparsification tolerance τ*, the number of steps* k *used to generate the pattern, and its number of nonzeroes* nnz_G.

Test case	τ	k	nnz_G	GPU time (ms)	CPU time (ms)
Apache2	1.0×10^{-2}	9	32,635,435	42	2,291
Cube3D	2.5×10^{-2}	6	4,474,548	33	482
G3_circuit	2.0×10^{-2}	7	32,229,868	42	1,935
Thermal2	2.0×10^{-2}	6	47,930,985	74	3,603
PFlow_742	5.0×10^{-2}	5	67,518,143	511	11,721

Table 1 compares the time required on the GPU and on the CPU to compute the pattern of the aforementioned sparse matrix varying the number of steps k with no preliminary dropping, i.e., setting $\tau = 0$. An average speedup of about a factor 20 is obtained for all k, with a peak of almost a factor 30 for the largest one.

4.3. Kernel for systems gather. The purpose of this kernel is to gather the dense local systems $A[\mathcal{P}_i, \mathcal{P}_i]$ from the global matrix needed by line 8 of Algorithm 3.1. Basically this requires we use elements of the input matrix according to the pattern, determined from the input matrix by k steps of the previous kernel. Since the number of elements per row varies significantly, the simple idea of using one thread per row would lead to a dramatic workload unbalance. To achieve a better workload balance we resort to a nontrivial mapping between CUDA threads and data, following an approach that we originally devised for a different problem (parallel breadth first search) [6] where we used the same data structure (CSR) to store the adjacency list of a graph. The central idea is to use one thread for each element of the system to be gathered. The mapping requires prefix-sum operations and a binary search function. Figure 1 shows (in a simplified form) the data structures involved and the required operations. The number of elements that can be processed in parallel is limited only by the amount of memory available on the GPU. The advantage of using a large number of CUDA threads for this operation is apparent by looking at the results shown in Table 2, which reports the timings for different test cases on GPU and CPU, respectively, along with the sparsification tolerance τ, the number of steps k used in the pattern generation, and the number of nonzeroes nnz_G of the G factor. The same thread-data mapping may be applied to other situations where there is a potential workload unbalance. Further details can be found in [6].

4.4. Kernel for batched Cholesky decomposition. The preconditioner needs to carry out the Cholesky decomposition of a small (size $N \leq 256$) dense matrix for each row of the input matrix. In general GPUs are more suitable to manage a sin-

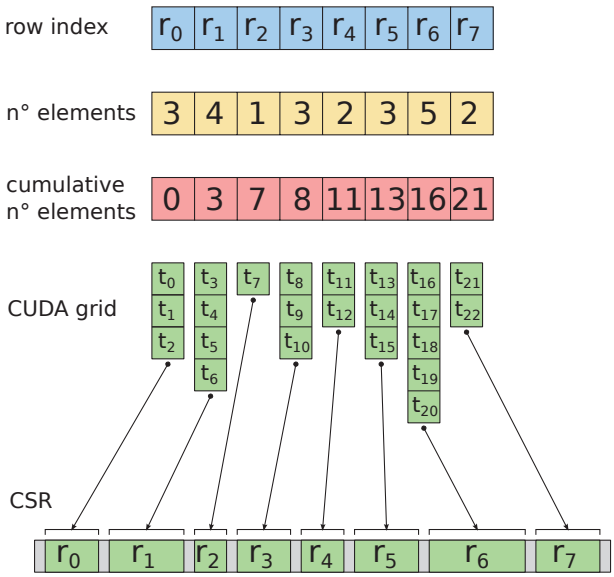

FIGURE 1. *Threads to data mapping assigning one thread per element. From top to bottom, the first array contains the row index of the CSR representation of the input matrix pattern. The second array contains the number of elements in each row. It is computed as the difference of two consecutive elements of the first array. The third array results from an exclusive scan of the second array. Each thread finds its row index by means of a binary search in the third (cumulative) array of the greatest entry less than its global ID. Finally, threads mapped to the same row process the list of elements in that row.*

gle large matrix (provided the matrix fits in the GPU global memory) rather than many small matrices. However, since a large amount of time is spent in this part of the preconditioning, we experimented with several alternatives to determine the most efficient solution and, in the end, we employed our own kernel instead of using the available CUBlas primitives for *batched* systems. In our solution, each input matrix is managed by a single block of threads. Moreover, the large number (65,536) of 32-bit registers available in Kepler SMXes allows the threads to perform the factorization using exclusively registers and a very small amount of shared memory. In this way, we limit the access to the (slow) global memory to the bare minimum, i.e., to read input and to write output data, and thus performance is limited mainly by instruction latencies and dependencies.

Since the matrices are symmetric, only their upper triangular parts are stored in global memory. The Cholesky kernel is launched with a one-dimensional (1D) grid of 2D blocks, one for each system in input. Initially, each thread block reads its upper triangular matrix into arrays local to its threads. Threads are mapped onto the matrices according to a tiling scheme such that each row (column) of the matrix is read by a single row (column) of threads in the block. In this way, a matrix of size $N = 256$ is read by a block of 16×16 threads into 256 arrays of size $16 \cdot 17/2 = 136$. Figure 2 shows an example of such mapping.

After the matrix is loaded in the registers, threads perform the factorization via the *outer-product* method [1]. From a data dependency point of view, this method iterates through the rows of the dense matrix and, for each row j, performs three updates, in sequence:

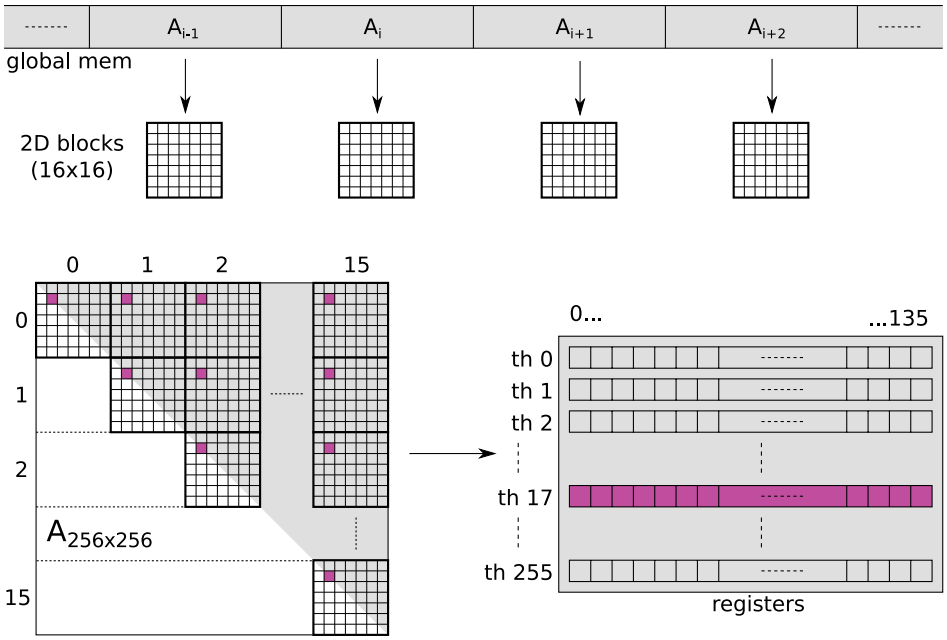

FIGURE 2. *Matrix tiled reads from a 2D block of* 16×16 *threads.*

element update: the diagonal element of row j is updated based on its own value;
row update: the rest of row j is updated based on the value of the diagonal element;
submatrix update: the rows $i, i > j$ are updated based on the values of row j.

The kernel works according to the following scheme. The N rows are processed in groups of $N/bsize$, with $bsize$ the block size (i.e., 16), and, for each row in the group, the element, row, and submatrix updates are performed. The element update is performed only by the thread whose block identifiers are $(j\%bsize, j\%bsize)$, where we use % as a symbol for the modulo operation. The row update is performed by all the threads with row index equal to $j\%bsize$ and the submatrix is updated by the whole block. Because of data dependencies, threads synchronize after each update (via the `__syncthreads()` intrinsic). Updates involve exclusively the values of local arrays so that no global memory location is ever accessed during the factorization. Moreover, elements shared among threads during the row and submatrix updates are copied to shared memory after their new values are computed. This amounts to a total of $N \cdot$`sizeof(real)` bytes (enough to contain one row of the matrix).

In order to obtain the best performance, it is crucial that the local arrays are allocated in the register file rather than in local memory (which, as a matter of fact, is part of the *slow* global memory). This simply follows from the fact that registers are the fastest memory available on the GPU, whereas global memory is the slowest. The `nvcc` compiler allocates arrays declared inside kernels in the registers only if it can compute, at compile time, the location of each access. For this reason, the kernel is written addressing local arrays exclusively with constant values. This is done by using the template meta-programming features of C++ to generate a kernel for every required matrix size. In our application, the kernel is specialized for any size between 32 and 256 that is a multiple of the 2D block size (actual matrices with different sizes are suitably padded and processed with the kernel for the next available size).

TABLE 3

Execution times, flop-count, and GFlop/s measured running the Cholesky batched kernel on $13,000$ *matrices on a Kepler K*80 *GPU, in both single and double floating-point precision, and on an Intel Xeon E*5645 *(*6 *cores) in double precision. GPU flop-counts are measured using the* flop_count_sp *and* flop_count_dp *metrics provided by the* nvprof *CUDA profiler.*

		GPU single precision			GPU double precision			CPU
Matrix size	Block size	GFlop count	Time (ms)	GFlop/s	GFlop count	Time (ms)	GFlop/s	Time (ms)
16	8x8	0.03	0.84	39.81	0.04	1.11	31.82	13.02
32	8x8	0.21	1.85	112.08	0.21	2.57	82.10	52.04
64	8x8	1.40	4.99	280.61	1.41	6.55	214.83	217.45
96	8x8	4.43	10.28	431.25	4.44	18.83	235.89	617.80
128	8x8	10.15	18.77	540.75	10.16	62.95	161.49	1036.87
160	16x16	20.64	45.16	457.06	20.66	80.90	255.34	1885.65
192	16x16	34.86	70.31	495.81	34.88	146.33	238.37	2815.09
224	16x16	54.43	109.36	497.71	54.45	187.54	290.37	3505.80
256	16x16	80.21	136.61	587.13	80.23	446.06	179.87	4469.03

4.5. Performance of batched Cholesky decomposition. We ran extensive testings of the kernel on our Nvidia K80 GPU with all the matrix sizes to find the block size that results in higher performance with each case (16, 32, 64, 96, 128, 160, 192, 224 and 256). Table 3 summarizes the performance results obtained solving $13,000$ matrices (1000 systems per SMX in the K80 GPU) in both single and double precision. For the sake of comparison, times to solve the same systems in double precision on the Intel Xeon E5645 CPU are also provided. For what concerns single precision, to have the maximum number of active blocks (i.e., systems) per SMX we compiled the code limiting the number of registers per thread to the smallest number that prevents all kernel instances from spilling registers to local memory (168 in our case). With double precision, on the other hand, applying register pressure didn't improve the performance as all the kernel instances use almost entirely the registers available per thread. The overall speedup from GPU to CPU in the double precision computation is around 20, although it varies significantly with the dense matrix size. The performance drop for the 128 and 256 cases with double precision is due to the register spilling caused by the large amount of registers required by the corresponding kernels.

5. Numerical results. In this section we analyze the overall performance of the present GPU implementation of FSAI preconditioned CG and compare it with the FSAIPACK CPU implementation [20]. The reference CPU implementation makes use of openMP directives for shared memory parallel computations and thus is able to fully exploit all the cores of a given processor. The hardware used in these tests is the same used in the previous section, that is, an NVIDIA Tesla K80 board and an Intel Xeon E5645 processor with 6 cores. For our tests we chose 5 SPD matrices coming from various application field which have been already used in experiments presented in other papers [17, 18, 19, 27, 38] and thus can be considered good benchmarks for our FSAI implementation. All of these matrices are available from the University of Florida sparse matrix collection [11] and their main characteristics are reported in Table 4.

As pointed out, selecting good parameters for a static FSAI preconditioner is not a trivial task as a good trade-off between preconditioner weight and effectiveness has to be found. The cost for applying the FSAI preconditioner is directly proportional to its density:

$$\mu = \frac{\mathrm{nnz}(G)}{\mathrm{nnz}(A)}, \tag{5.1}$$

TABLE 4

Main characteristics of the matrices used in the experiments with a brief description of the problem they arise from: size (n), number of nonzeroes (nnz), average number of nonzeroes per row (AVG nnzr), maximum number of nonzeroes per row (MAX nnzr), and standard deviation of the number of nonzeroes per row (StD nnzr).

Matrix name	n	nnz	AVG $nnzr$	MAX $nnzr$	StD $nnzr$	Description
Apache2	715,176	4,817,870	6.74	8	0.45	Structural
Cube3D	190,581	7,531,389	39.52	96	16.19	Structural
G3_circuit	1,585,478	7,660,826	4.83	6	0.64	Circuit sim.
Thermal2	1,228,045	8,580,313	6.99	11	0.81	Thermal
PFlow_742	742,793	37,138,461	50.00	137	19.97	Porous flow

TABLE 5

Set-up parameters, τ, k and δ, for nearly optimal FSAI performance on both GPU and CPU with corresponding densities of the FSAI factors before and after filtration, $\widehat{G}$ and G, respectively.

Matrix name	τ	k	δ	$\mu(\widehat{G})$	$\mu(G)$
Apache2	1.0×10^{-2}	9	5.0×10^{-2}	6.774	0.998
Cube3D	2.5×10^{-2}	6	1.0×10^{-2}	0.594	0.284
G3_circuit	2.0×10^{-2}	7	7.0×10^{-2}	4.207	1.465
Thermal2	2.0×10^{-2}	6	7.0×10^{-2}	5.586	1.737
PFlow_742	5.0×10^{-2}	5	3.5×10^{-3}	1.818	1.168

TABLE 6

Comparison of the number of iterations required by CG convergence by using Jacobi preconditioning and FSAI computed with the set-up parameters of Table 5.

Test case	FSAI	Jacobi
Apache2	853	2834
Cube3D	993	2005
G3_circuit	390	2054
Thermal2	1221	4056
PFlow_742	1529	+10000

TABLE 7

Comparison of the FSAI performance on GPU and CPU. The time for the preconditioner set-up, T_p, the time for the PCG iterations, T_s, and the total time, $T_t = T_p + T_s$, are provided for both the hardware platforms.

	GPU			CPU		
Test case	T_p (s)	T_s (s)	T_t (s)	T_p (s)	T_s (s)	T_t (s)
Apache2	1.49	2.46	3.95	15.66	15.33	30.99
Cube3D	0.33	2.01	2.33	2.41	9.45	11.86
G3_circuit	1.65	2.49	4.14	15.64	16.75	32.39
Thermal2	1.72	7.05	8.77	22.54	51.19	73.73
PFlow_742	5.07	21.59	26.66	49.15	151.35	200.50

where $\mathrm{nnz}(\cdot)$ stands for the number of nonzeroes of the argument matrix. For each test matrix we selected, through a few trials, a preconditioner set-up able to give nearly optimal performance on both the GPU and the CPU architecture. Table 5 provides for each matrix the prefiltration tolerance τ, the number of steps k used in determining the pattern, and the postfiltration tolerance δ, together with the densities of the FSAI factors before and after postfiltration, $\widehat{G}$ and G, respectively. Note that the computed preconditioner is usually heavier than the original matrix, but after postfiltration a cheaper factor can be used in the iterations.

We computed a right-hand side corresponding to the unitary solution, i.e., $\boldsymbol{b} = A\boldsymbol{e}$ with $\boldsymbol{e}$ the unitary vector, and solved the related linear system through PCG until

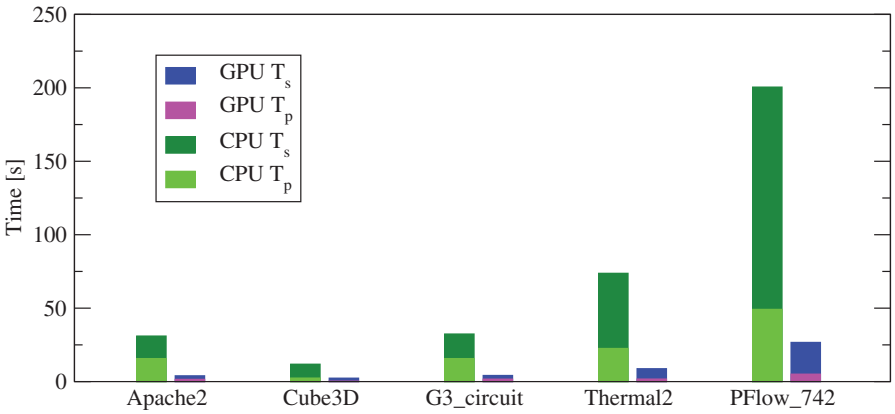

FIGURE 3. *Comparison between the time performance of the FSAI conjugate gradient implementation on GPU and CPU.*

the relative residual decreases below a tolerance of 1.0×10^{-8}. To give an idea of the importance of a strong preconditioner as FSAI, Table 6 compares the number of iterations required for convergence by a simple Jacobi preconditioning and FSAI computed with set-up parameters of Table 5. It shows that to reduce the residual by 8 orders of magnitude, Jacobi requires always more than twice the iterations than FSAI and in the most difficult case, PFlow_742, it is not even able to solve the system at all. Table 7 compares the outcome of the GPU and CPU implementations, reporting the time for the preconditioner computation T_p (including all data read operations and data structures generation and conversion), the time for the PCG iterations T_s and the total time $T_t = T_p + T_s$. The number of PCG iterations performed by the GPU implementation is the same reported in the corresponding column of Table 6. Due to the different round-off error produced by the two hardware platforms, the CPU implementation performs a slightly different number of iterations.

A graphical representation of the time performance obtained on GPU and CPU is provided in Figure 3, where different colors have been used to distinguish between the set-up and the iteration phases. An outright understanding of the advantage offered by the GPU hardware can be evinced from Figure 4, where the GPU over CPU speedups, i.e., the ratio between the time required on CPU and on GPU, are provided for T_p, T_s and T_t. The largest speedup is obtained during the preconditioner set-up stage with a peak of about 13.14 for the Thermal2 test case and an average of about 10.03 over all the tests. The speedup obtained on the PCG iteration is slightly lower than the previous one, with a peak of 7.26 for Thermal2 again and an average of about 6.39. This is not surprising, as it is well-known that CG is a sequence of SpMV, dot products, and vector updates which are bandwidth limited operations, and thus the greatest advantage provided by the GPU implementation takes place in the set-up stage, which is more computational intensive. Nevertheless, the Kepler K80 GPU allows for an overall average speedup of 7.34 over a high-end Intel Xeon 6-cores processor.

Finally, Figure 5 shows how the time required for the FSAI computation is distributed among the different kernels. The main cost is represented by the batched Cholesky solution of the dense linear systems, followed by the pattern generation and the post-filtration procedure. The cost for gathering the dense linear systems has

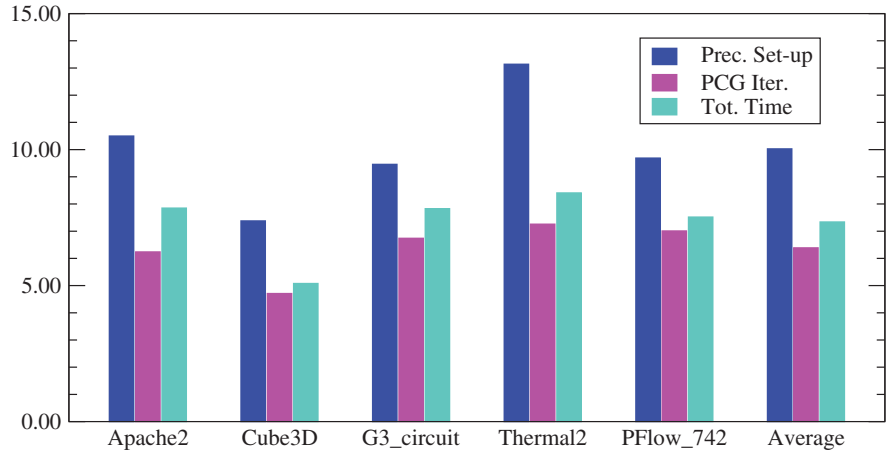

FIGURE 4. *Ratio between the time performance obtained on CPU and on GPU in the preconditioner set-up phase T_p, in the CG iteration stage T_s and in the whole solution of the linear system $T_t = T_p + T_s$.*

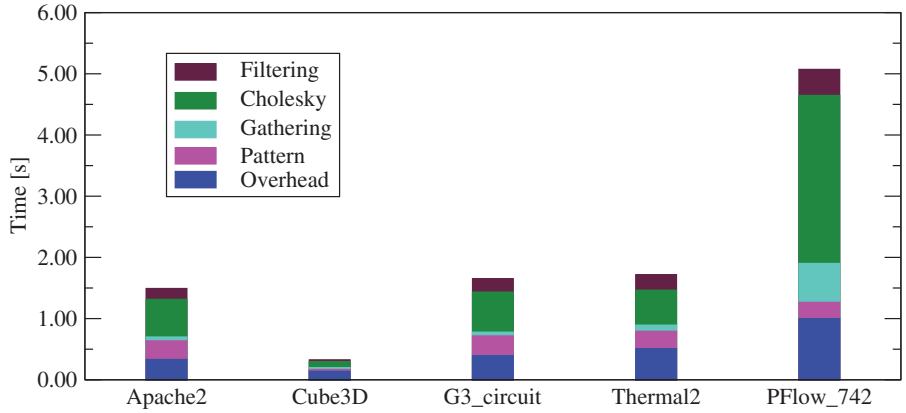

FIGURE 5. *Distribution of overall FSAI set-up time among the various kernels.*

only a little impact on the overall solution; however, the effectiveness of the kernel developed may deserve interest in adaptive procedures where the repeated gathering of small systems is required. The overhead time is quite high as it includes a lot of unavoidable expensive operations such as the transfer of the matrix A from the host to the device, some changes in the matrix storage format to have an optimal layout in every kernel, and the final transposition of the FSAI factor since in the CG algorithm both G and G^T are used.

6. Conclusions. We presented a novel FSAI preconditioned conjugate gradient solver running completely on GPU devices. While CG iterations are performed relying on standard sparse linear algebra libraries, cuSPARSE in our case, the set-up phase has required an ad hoc design of new numerical kernels able to take full advantage of the GPU architecture. The comparison of the present implementation with an equivalent CPU variant on up-to-date hardware has shown the following:

- The numerical performance in the set-up stage is significantly higher with a reduction of the wall clock time up to a factor 13. This speedup is mainly due to the more computationally intensive kernels for the pattern generation and the Cholesky decomposition developed herein. Larger speedups could have been obtained if the overall procedure had not damped by the overhead, caused by some unavoidable costly data movements.
- The CG iteration stage is not overwhelmed by a tricky preconditioner application as FSAI implies only another SpMV product. Unfortunately, this phase of the solution process is limited by the bandwidth, and hence the maximum speedup achieved is about a factor 7.
- The maximum speedup achieved for the complete solution of our test matrices, which are borrowed from real-world applications, is larger than 8, justifying the effort in recasting the CPU kernels.

This positive experience also raises some ideas for future research:

- It is a matter of fact that GPUs are mainly designed for single precision operations, whereas double precision computations are possible at the cost of considerable performance reduction, as shown, for instance, in the batched Cholesky decomposition kernel. If an accurate solution of a large size system is desired, single precision cannot be used in the CG process; however, sacrificing some accuracy in the preconditioner is reasonable and can give good results.
- Better results may be obtained in the iteration stage by developing new SpMV operations targeted on matrices arising from selected applications. In some situations, a specific matrix layout can make possible a better exploitation of the available bandwidth.
- It could be interesting to port on the GPU hardware adaptive pattern FSAI preconditioners as well. These latter are usually more effective at the price of a higher set-up cost, and hence the advantage will be more pronounced, especially taking into account the excellent performance exhibited by our three new kernels, which have to be called several times in adaptive algorithms.

Acknowledgment. The authors thank Andrea Castronuovo for his valuable contribution in implementing the first version of the GPU code.

REFERENCES

[1] M. J. Anderson, D. Sheffield, and K. Keutzer, *A predictive model for solving small linear algebra problems on GPU registers*, in proceeding of the 26th International Parallel and Distributed Processing Symposium, IEEE, 2012.

[2] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, *Minimizing communication in numerical linear algebra*, SIAM J. Math. Anal. Appl., 32 (2011), pp. 866–901.

[3] N. Bell and M. Garland, *Efficient Sparse Matrix-Vector Multiplication on CUDA*, NVIDIA Technical report NVR-2008-004, NVIDIA Corporation, 2008.

[4] M. Benzi and M. Tůma, *A comparative study of sparse approximate inverse preconditioners*, Appl. Numer. Math., 30 (1999), pp. 305–340.

[5] M. Benzi, C. D. Meyer, and M. Tůma, *A sparse approximate inverse preconditioner for the conjugate gradient method*, SIAM J. Sci. Comput., 17 (1996), pp. 1135–1149

[6] M. Bernaschi and E. Mastrostefano, *Efficient breadth first search on multi-GPU systems*, J. Parallel Distr. Comput. 73 (2013), pp. 1292–1305.

[7] D. Bertaccini and S. Filippone, *Approximate inverse preconditioners for Krylov methods on heterogeneous parallel computers*, Adv. Parallel Comput., 25 (2014), pp. 183–192.

[8] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, *Sparse matrix solvers on the GPU: Conjugate gradients and multigrid*, ACM Trans. Graph., 22 (2003), pp. 917–924.

[9] E. Chow, *A priori sparsity patterns for parallel sparse approximate inverse preconditioners*, SIAM J. Sci. Comput., 21 (2000), pp. 1804–1822.
[10] H. V. Dang and B. Schmidt, *CUDA-enabled sparse matrix-vector multiplication on GPUs using atomic operations*, Parallel Comput., 39 (2013), pp. 737–750.
[11] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Software, 38 (2011), pp. 1–25.
[12] M. M. Dehnavi, D. M. Fernandez, J. L. Gaudiot, and D. D. Giannacopoulos, *Parallel sparse approximate inverse preconditioning on graphic processing units*, IEEE Trans. Par. Distr. Syst., 24 (2013), pp. 1852–1862.
[13] M. J. Grote and T. Huckle, *Parallel preconditioning with sparse approximate inverses*, SIAM J. Sci. Comput., 18 (1997), pp. 838–853.
[14] R. Helfenstein and J. Koko, *Parallel preconditioned conjugate gradient algorithm on GPU*, J. Comput. Appl. Math., 236 (2012), pp. 3584–3590.
[15] T. Huckle, *Approximate sparsity patterns for the inverse of a matrix and preconditioning*, Appl. Numer. Math., 30 (1999), pp. 291–303.
[16] T. Huckle, *Factorized sparse approximate inverses for preconditioning*, J. Supercomput., 25 (2003), pp. 109–117.
[17] C. Janna and M. Ferronato, *Adaptive pattern research for block FSAI preconditioning*, SIAM J. Sci. Comput., 33 (2011), pp. 3357–3380.
[18] C. Janna, M. Ferronato, and G. Gambolati, *Enhanced block FSAI preconditioning using domain decomposition techniques*, SIAM J. Sci. Comput., 35 (2013), pp. S229–S249.
[19] C. Janna, M. Ferronato, and G. Gambolati, *The use of supernodes in factored sparse approximate inverse preconditioning*, SIAM J. Sci. Comput., 37 (2015), pp. C72–C94.
[20] C. Janna, M. Ferronato, F. Sartoretto, and G. Gambolati, *FSAIPACK: A software package for high performance FSAI preconditioning*, ACM Trans. Math. Software, 41 (2015), article 10.
[21] I. E. Kaporin, *New convergence results and preconditioning strategies for the conjugate gradient method*, Numer. Linear Algebra Appl., 1 (1994), pp. 179–210.
[22] L. Y. Kolotilina and A. Y. Yeremin, *Factorized sparse approximate inverse preconditioning.* I. *Theory*, SIAM J. Math. Anal. Appl., 14 (1993), pp. 45–58.
[23] L. Y. Kolotilina, A. A. Nikishin, and A. Y. Yeremin, *Factorized sparse approximate inverse preconditioning.* IV. *Simple approaches to rising efficiency*, Numer. Linear Algebra Appl., 6 (1999), pp. 515–531.
[24] Z. Koza, M. Matyka, S. Szkoda, and L. Miroslaw, *Compressed multirow storage format for sparse matrices on graphics processing units*, SIAM J. Sci. Comput., 36 (2014), pp. C219–C239.
[25] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units*, SIAM J. Sci. Comput., 36 (2014), pp. C401–C423.
[26] B. S. Lazarov and O. Sigmund, *Factorized parallel preconditioner for the saddle point problem*, Int. J. Numer. Methods Biomed. Eng., 27 (2011), pp. 1398–1410.
[27] R. Li and Y. Saad, *GPU-accelerated preconditioned iterative linear solvers*, J. Supercomput., 63 (2013), pp. 443–466.
[28] M. Macedonia, *The GPU enters computing's mainstream*, Computer, 36 (2003), pp. 106–108.
[29] M. Maggioni and T. Berger-Wolf, *An architecture-aware technique for optimizing sparse matrix-vector multiplication on GPUs*, Procedia Comput. Sci. 18 (2013), pp. 329–338.
[30] X. Mei, K. Zhao, C. Liu, and X. Chu, *Benchmarking the Memory Hierarchy of Modern GPUs*, in Network and Parallel Computing, Springer, Berlin, 2014, pp. 144–156.
[31] CUDA C Programming Guide, PG-02829-001_v7.0, Chapter 2. http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (2015).
[32] cuSPARSE Library, DU-06709-001_v7.0 http://docs.nvidia.com/cuda/cusparse (2015).
[33] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva, *MPI-CUDA sparse matrix-vector multiplication for the conjugate gradient method with an approximate inverse preconditioner*, Comput. Fluids, 92 (2014), pp. 244–252.
[34] A. Patel and E. Chow, *Fine-grained parallel incomplete LU factorization*, SIAM J. Sci. Comput., 37 (2015), pp. C169–C193.
[35] J. C. Pichel, F. F. Rivera, M. Fernández, and A. Rodríguez, *Optimization of sparse matrix-vector multiplication using reordering techniques on GPUs*, Microprocess. Microsyst., 36 (2012), pp. 65–77.
[36] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, 2003.

[37] C. J. Thompson, S. Hahn, and M. Oskin, *Using modern graphics architectures for general-purpose computing: A framework and analysis*, in Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 35), IEEE Computer Society Press, Los Alamitos, CA, 2002, pp. 306–317.

[38] M. Verschoor and A. C. Jalba, *Analysis and performance estimation of the conjugate gradient method on multiple GPUs*, Parallel Comput., 38 (2012), pp. 552–575.

[39] K. Xu, D. Z. Ding, Z. H. Fan, and R. S. Chen, *FSAI preconditioned CG algorithm combined with GPU technique for the finite element analysis of electromagnetic scattering problems*, Finite Elem. Anal. Des., 47 (2011), pp. 387–393.

[40] A. Yu. Yeremin and A. A. Nikishin, *Factorized-sparse-approximate-inverse preconditionings of linear systems with unsymmetric matrices*, J. Math. Sci., 121 (2004), pp. 2448–2457.