SIAM J. SCI. COMPUT.
Vol. 41, No. 3, pp. C139–C160

A DYNAMIC PATTERN FACTORED SPARSE APPROXIMATE INVERSE PRECONDITIONER ON GRAPHICS PROCESSING UNITS*

MASSIMO BERNASCHI†, MAURO CARROZZO†, ANDREA FRANCESCHINI‡, AND CARLO JANNA§

Abstract. One of the most time-consuming tasks in the procedures for the numerical study of PDEs is the solution to linear systems of equations. To that purpose, iterative solvers are viewed as a promising alternative to direct methods on high-performance computers since, in theory, they are almost perfectly parallelizable. Their main drawback is the need of finding a suitable preconditioner to accelerate convergence. The factorized sparse approximate inverse (FSAI), mainly in its adaptive form, has proven to be an effective parallel preconditioner for several problems. In the present work, we report about two novel ideas to dynamically compute, on graphics processing units (GPUs), the FSAI sparsity pattern, which is the main task in its setup. The first approach, borrowed from the CPU implementation, uses a global array as a nonzero indicator, whereas the second one relies on a merge-sort procedure of multiple arrays. We will show that the second approach requires significantly less memory and overcomes issues related to the limited global memory available on GPUs. Numerical tests prove that the GPU implementation of FSAI allows for an average speed-up of 7.5 over a parallel CPU implementation. Moreover, we will show that the preconditioner computation is still feasible using single precision arithmetic with a further 20% reduction of the setup cost. Finally, the strong scalability of the overall approach in shown in a multi-GPU setting.

Key words. GPUs, approximate inverses, preconditioning

AMS subject classifications. 65F08, 65F50, 68W10

DOI. 10.1137/18M1197461

1. Introduction.

Efficiently solving linear systems of equations

$$A\boldsymbol{x} = \boldsymbol{b}, \tag{1}$$

with $A \in \mathbb{R}^{n\times n}$ and $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$, is a crucial issue in several application fields, in particular when discretizing PDEs. These linear systems may easily incorporate millions or even billions of unknowns and often represent the most time-consuming stage of a simulation process. Especially in large three-dimensional problems, iterative methods are among the most widely used, due to their parsimonious memory requirements, and also because they are quite easily parallelizable as their implementation involves mostly sparse-matrix vector (SpMV) products [34]. In order to be effective, however, any iterative method needs a proper preconditioner, that is, a matrix representing an approximation of A^{-1} but much cheaper to compute and apply to a vector [4]. A suitable choice of a good preconditioner is even more difficult in the context of modern many-core architectures, where performance is obtained by exploiting a high level of concurrency and taking advantage of fine-grained parallelism. Traditional approaches, like incomplete factorization or Gauss–Seidel relaxation, though capable of reducing the iteration count of any Krylov subspace method, may give poor performance on modern systems due to their intrinsically sequential nature. In turn,

*Submitted to the journal's Software and High-Performance Computing section June 29, 2018; accepted for publication (in revised form) February 26, 2019; published electronically May 9, 2019.
http://www.siam.org/journals/sisc/41-3/M119746.html

†Institute for Applied Computing, CNR, 00185 Rome, Italy (m.bernaschi@iac.cnr.it, m.carrozzo@iac.cnr.it).

‡Department ICEA, University of Padova, 35131 Padova, Italy (franc90@dmsa.unipd.it).

§Corresponding author. M^{3}E s.r.l., 35129 Padova, Italy (c.janna@m3eweb.it).

other methods less effective when the processing is sequential may become preferable if parallel processors are available [26].

Graphics processing units (GPUs), one of the possible alternatives for executing computations in parallel, provide a very good trade-off among four crucial goals of any modern computing platform: performance, price, power consumption, and ease of software development. As a matter of fact, although originally designed to accelerate graphics rendering, GPUs are, by now, widely used in many other applications of high-performance computing. The architecture of a GPU is massively parallel, with thousands of simple cores designed for simultaneous, independent, not necessarily identical computations on multiple data inputs [30].

Approximate inverses (AIs) are known to be good candidates for GPU computing as their application involves SpMV operations only. There are several approaches available in the literature, offering both the preconditioner as a whole matrix [12, 16] or in factored form [5, 9, 1]. The reader can refer to [6] for a thorough review on AIs. A very effective AI technique originally introduced in [22, 23, 37, 24] is the factored sparse approximate inverse (FSAI), where the triangular factor of the matrix inverse is computed directly by minimizing its distance from the exact lower Cholesky factor, in the Frobenius norm sense. In its original proposal, FSAI was computed by means of an a priori guess of a lower nonzero pattern that, hopefully, had to capture the most relevant entries of the true inverse factor. We refer to that version of FSAI as *static* (sFSAI). Several heuristics have been proposed to determine in advance the preconditioner pattern [17, 10], and several parallel sFSAI implementations have been proposed taking advantage of both CPUs [11, 20] and GPUs [14, 36, 7]. However, in ill-conditioned problems characterized by abrupt changes in the matrix entries, it is hard to predict an effective sFSAI pattern, and it is often necessary to overestimate the preconditioner density in order to reach convergence in a reasonable number of iterations. As the sFSAI setup involves the solution to dense linear systems whose size is proportional to the number of nonzeros in the preconditioner rows, it may easily cause an excessive processing time. To overcome the issue, an *adaptive* FSAI technique (aFSAI, in what follows) has been developed which is able to dynamically choose a proper nonzero structure for the factor [18]. The aFSAI effectiveness in solving ill-conditioned systems has been thoroughly studied and assessed in several application fields [25, 20, 2]. More recently, aFSAI has proven to work well not only as a standalone preconditioner but also as a smoother in the algebraic multigrid (AMG) framework [33], a very effective method for large-sized problems which is becoming increasingly popular also on GPUs [3, 15, 29, 27]

An efficient application of the aFSAI on a GPU is not an easy task, as it needs the development of several specialized kernels working on both sparse and dense data. Moreover, its adaptivity features result in a very different effort required in the setup of each preconditioner row, thus preventing the straightforward thread-by-row scheduling that is possible in the case of the sFSAI. By contrast, using aFSAI at the iteration stage is straightforward, as its application only requires a couple of SpMV products. In the rest of the paper we focus on the description of our GPU implementation of the aFSAI setup for symmetric positive definite (SPD) systems, whereas we rely on standard CUBLAS [31] and CUSPARSE [32] primitives for the matrix-vector operations required by the preconditioned conjugate gradient (PCG) algorithm.

The rest of the paper is organized as follows. In section 2 we recall the FSAI preconditioner, in both its static and adaptive forms. Section 3 presents the numerical kernels implemented for the GPU version of aFSAI, with particular focus on the two approaches for the computation of the sparse-matrix by sparse-vector (SpM-SpV)

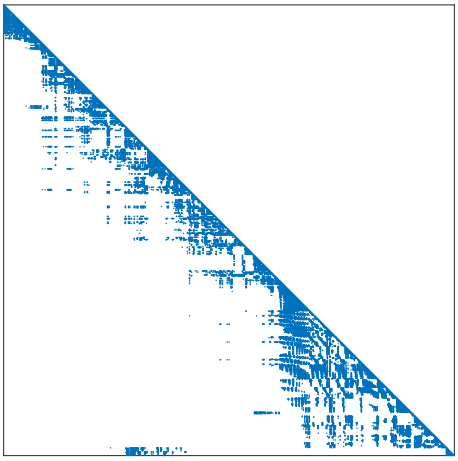

FIG. 1. *Example of a lower triangular nonzero pattern* $\mathcal{S}$.

product. In section 4, we present a set of numerical results showing the gain obtained moving from double to single precision in the aFSAI computation, the speed-up of the GPU with respect to a parallel CPU, and the performance on multiple GPUs. In the end, we draw some conclusions in section 5.

2. Factored sparse approximate inverse preconditioning. The original sFSAI preconditioner for SPD matrices was introduced in [22], with the aim of directly approximating the inverse of A as the product of two triangular factors:

$$M^{-1} = G^T G \simeq A^{-1}, \tag{2}$$

where the lower triangular matrix G is computed by minimizing the Frobenius norm

$$\|I - GL\|_F \tag{3}$$

over the set $\mathcal{W}_{\mathcal{S}}$ of matrices having a prescribed lower triangular nonzero pattern $\mathcal{S}$, as the one shown in Figure 1.

The matrix L is the exact lower triangular factor of A and, though explicitly appearing in (3), it is not actually needed in the computation of sFSAI. The unknown G entries, g_{ij}, are computed by solving the componentwise system:

$$[GA]_{ij} = \begin{cases} 0, & i \neq j, \quad (i,j) \in \mathcal{S}, \\ l_{ii}, & i = j, \end{cases} \tag{4}$$

which is obtained by differentiating (3) with respect to g_{ij} and setting it equal to zero. The symbol $[\cdot]_{ij}$ in (4) indicates 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 (4) is replaced by 1 and the matrix $\widetilde{G}$ is computed instead by solving

$$[\widetilde{G}A]_{ij} = \delta_{ij}, \tag{5}$$

where δ_{ij} is the Kronecker delta. Operatively, to set up the ith row of $\widetilde{G}$, say $\widetilde{\boldsymbol{g}}_i^T$, first the set $\mathcal{P}_i$ is defined by collecting all the column indices that belong to the ith row of $\mathcal{S}$, that is,

$$\mathcal{P}_i = \{j : (i,j) \in \mathcal{S}\}. \tag{6}$$

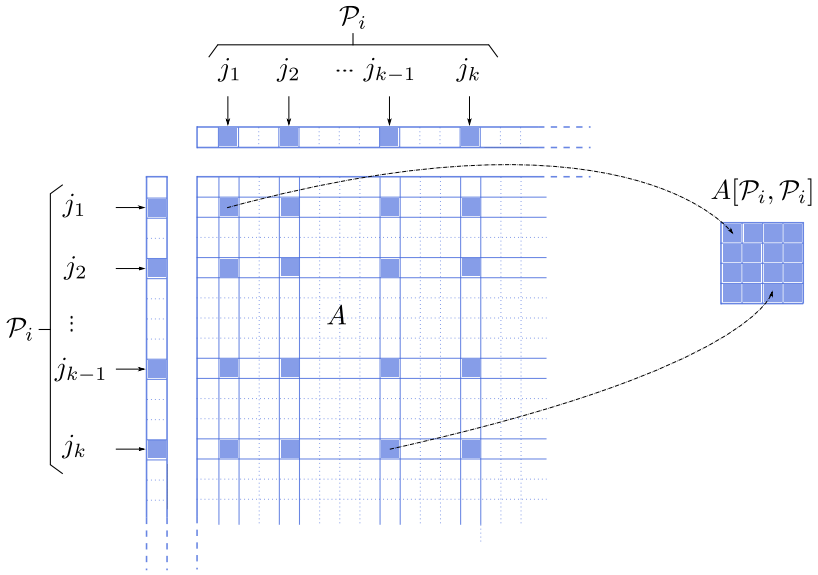

FIG. 2. *Schematic representation of the dense linear system gathering for a given set $\mathcal{P}_i$ (with cardinality k).*

Then the dense matrix $A[\mathcal{P}_i, \mathcal{P}_i]$ is formed by gathering the entries of A having row/column indices in $\mathcal{P}_i$, and the linear system

$$A[\mathcal{P}_i, \mathcal{P}_i]\widetilde{G}[i, \mathcal{P}_i]^T = \boldsymbol{e}_{m_i} \tag{7}$$

is solved, with $\widetilde{G}[i, \mathcal{P}_i]$ being the dense vector containing the nonzero entries of $\widetilde{\boldsymbol{g}}_i$ and the right-hand side $\boldsymbol{e}_{m_i}$ given by the last vector of the canonical basis of $\mathbb{R}^{m_i}$ with $m_i = |\mathcal{P}_i|$. Figure 2 gives an idea of the gathering of the dense linear systems $A[\mathcal{P}_i, \mathcal{P}_i]$ given a set $\mathcal{P}_i$. Each system is the collection of all the entries of A such that both i and j belong to the set $\mathcal{P}_i$.

An equivalent but more practical way to compute sFSAI (see [18]) can be found by assuming a unitary diagonal for $\widetilde{G}$ and solving the linear system

$$A[\overline{\mathcal{P}_i}, \overline{\mathcal{P}_i}]\widetilde{G}[i, \mathcal{P}_i]^T = -A[\overline{\mathcal{P}_i}, i], \tag{8}$$

where $\overline{\mathcal{P}_i} = \mathcal{P}_i \setminus i$. Thus, $\widetilde{G}[i, \mathcal{P}_i]$ now contains the *off-diagonal* nonzero entries of $\widetilde{G}$. The final G is obtained by scaling each row in order to guarantee a unitary diagonal on the preconditioned matrix

$$\operatorname{diag}(D_G\widetilde{G}A\widetilde{G}^T D_G) = \operatorname{diag}(GAG^T) = I, \tag{9}$$

where $\operatorname{diag}(\cdot)$ is the operator returning the diagonal matrix having as entries the diagonals of its argument and I is the identity matrix. Simple algebra shows that the entries of D_G are given by

$$D_G[i, i] = \frac{1}{A[i, i] - \widetilde{G}[i, \mathcal{P}_i]A[\overline{\mathcal{P}_i}, \overline{\mathcal{P}_i}]\widetilde{G}[i, \mathcal{P}_i]^T}. \tag{10}$$

This last condition ensures that G, over all the matrices $B \in \mathcal{W}_{\mathcal{S}}$, is the unique

minimizer of the Kaporin number of the preconditioned matrix:

$$\kappa = \frac{\frac{1}{n}\mathrm{tr}(GAG^T)}{\det(GAG^T)^{\frac{1}{n}}}, \tag{11}$$

which gives a measure of the PCG convergence rate [21].

The sFSAI preconditioner for an SPD matrix is breakdown-free, and its application to a vector is inherently parallel since it only requires SpMV, like any approximate inverse. A very attractive feature of sFSAI is the high degree of parallelism in its construction that makes it an ideal candidate for GPUs. Nevertheless, to be competitive with other approaches, the sFSAI preconditioner needs to be sparse and its nonzero pattern has to collect all the most significant entries of the true inverse factor of A. The appropriate choice of $\mathcal{S}$ plays a crucial role but, unfortunately, it is not an easy task.

Another option consists in determining $\mathcal{S}$ *dynamically* while $\widetilde{G}$ is computed [20]. The basic concept in the dynamic computation of aFSAI is the improvement of the quality of a given initial factor G_0, satisfying (8) and (9), by adding to its pattern those positions that mostly contribute in reducing the Kaporin number of $G_0AG_0^T$. Let us call $\mathcal{S}_0$ the nonzero pattern of G_0, and let the identity matrix be our default initial guess. The Kaporin number κ of the preconditioned matrix $G_0AG_0^T$ is explicitly written as

$$\kappa = \frac{\frac{1}{n}\mathrm{tr}(G_0AG_0^T)}{\det(G_0AG_0^T)^{\frac{1}{n}}} = \frac{\frac{1}{n}\mathrm{tr}(D_{G_0}\widetilde{G}_0A\widetilde{G}_0^TD_{G_0})}{\det(D_{G_0}\widetilde{G}_0A\widetilde{G}_0^TD_{G_0})^{\frac{1}{n}}}. \tag{12}$$

Recalling that $G_0AG_0^T$ has unitary diagonal entries and noting that $\det(\widetilde{G}_0) = 1$, it follows that

$$\kappa = \frac{1}{\det(A)^{\frac{1}{n}}}\ \frac{1}{\det(\widetilde{D}_0)^{\frac{2}{n}}} = \frac{\det\left[\mathrm{diag}(\widetilde{G}_0A\widetilde{G}_0^T)\right]^{\frac{1}{n}}}{\det(A)^{\frac{1}{n}}}. \tag{13}$$

Denoting by $\widetilde{\boldsymbol{g}}_{0,i}^T$ the ith row of $\widetilde{G}_0$, the numerator of (13) can be written as

$$\det\left[\mathrm{diag}(\widetilde{G}_0\widetilde{A}\widetilde{G}_0^T)\right] = \prod_{i=1}^{n}\widetilde{\boldsymbol{g}}_{0,i}A\widetilde{\boldsymbol{g}}_{0,i}^T = \prod_{i=1}^{n}\psi_{0,i}, \tag{14}$$

where $\psi_{0,i} = \widetilde{\boldsymbol{g}}_{0,i}A\widetilde{\boldsymbol{g}}_{0,i}^T$. Hence, the Kaporin conditioning number of the preconditioned matrix reads

$$\kappa = \left(\frac{\prod_{i=1}^{n}\psi_{0,i}}{\det(A)}\right)^{\frac{1}{n}}. \tag{15}$$

Our goal is to find an augmented pattern $\mathcal{S}_\infty$ allowing for the largest possible reduction of κ in (15). To that goal, we compute the gradient of κ with respect to $\widetilde{G}_0$ and add in $\mathcal{S}_\infty$ the positions corresponding to its largest components in absolute value. Since each $\psi_{0,i}$ in (15) does not depend on any other row of $\widetilde{G}_0$ except the ith, the pattern expansion can be carried out on each row concurrently. The gradient of

$\psi_{0,i}$, which we denote by $\nabla\psi_{0,i}$, is obtained by collecting its partial derivatives with respect to the components of $\widetilde{\boldsymbol{g}}_{0,i}$:

$$\frac{\partial\psi_{0,i}}{\partial\tilde{g}_{0,ij}} = 2\left(\sum_{r=1}^{n} a_{jr}\,\tilde{g}_{0,ir} + a_{ji}\right) \qquad \forall j = 1,\ldots,i-1. \tag{16}$$

The computational cost of its evaluation is a single sparse-matrix by sparse-vector product. After computing $\nabla\psi_{0,i}$ using (16), the nonzero pattern $\overline{\mathcal{P}}_i^0$ is enlarged by adding the s positions corresponding to the largest entries of $\nabla\psi_{0,i}$ in absolute value. A new pattern $\overline{\mathcal{P}}_i^1$ is obtained and then $\widetilde{\boldsymbol{g}}_{1,i}$ is computed by solving equations (8), after replacing the sub-/superscript "0" with "1". The above procedure is repeated $k_{\max}$ times to find all the rows of $\widetilde{G}_2$, $\widetilde{G}_3$, ..., up to $\widetilde{G}_{k_{\max}}$. The quality of the preconditioner can be easily monitored by computing the value of $\psi_{k,i}$. Hence it is possible to stop the adaptive procedure independently for every row when

$$\frac{\psi_{k,i}}{\psi_{0,i}} = \frac{\widetilde{\boldsymbol{g}}_{k,i}A\widetilde{\boldsymbol{g}}_{k,i}^T}{\widetilde{\boldsymbol{g}}_{0,i}A\widetilde{\boldsymbol{g}}_{0,i}^T} \leq \varepsilon, \tag{17}$$

where ε is a user-specified tolerance. Once the approximation of $\widetilde{G}$ is satisfactory, a diagonal scaling is applied, as in the static procedure, to ensure that the diagonal of GAG^T is unitary.

The parameters that control the aFSAI quality are the following:

1. $k_{\max}$, the maximum number of steps of the iterative procedure for each row;
2. s, the number of new positions that are added to the nonzero pattern of each row in a single step;
3. ε, the relative tolerance on the Kaporin number reduction used to stop the procedure.

Though the final factor G needs to be necessarily the same, on GPU the setup algorithm has to be redesigned with respect to its CPU counterpart to exploit the features of the device at their best. In our previous work [7], we obtained excellent performance on several real-world problems by totally recasting the two kernels responsible for gathering and solving the sequence of small dense linear systems $A[\mathcal{P}_i,\mathcal{P}_i]$ in the static setup. In that case, only a *local* modification of the setup algorithm was required. In aFSAI, this modification is deeper and more involved. More specifically, aFSAI exposes a high potential parallelism because the computation of each G row can take place independently of the others. However, the number of significant nonzeros of each row is highly variable over G and, as a consequence, also the resources to be allocated for its computation may be completely different from row to row. In sFSAI, the number of nonzeros per row is known at the early stage of setup, and thus a proper scheduling of the work to be carried out can be easily determined at the beginning. By contrast, in aFSAI, the number of nonzeros is the result of an iterative process controlled dynamically by ε at the level of the single row, and it is therefore unpredictable. The main consequence is that, unlike the CPU implementation where the outermost loop involves the matrix rows [20], on GPU we iterate over all the matrices up to a maximum of $k_{\max}$ steps but excluding rows from the actual computation as soon as they fulfill the criterion defined in 17.

A summary of the procedure just described is provided in Algorithm 1, where the array `Active` of size n is used to indicate whether the processing of a row is executed or not (the processing stops when the criterion 17 is fulfilled). Lines 8 and

Algorithm 1 aFSAI - FSAI with dynamic pattern generation.

Input: Setup parameters: $k_{\max} \geq 0$, $s \geq 1$, $\varepsilon \geq 0.0$
Input: $A \in \mathbb{R}^{n \times n}$
Output: $G \in \mathbb{R}^{n \times n}$

1. $\widetilde{G}_0 \leftarrow I$
2. `Active`$(1:n) \leftarrow 1$
3. **Do** $(k = 0;\ k < k_{\max};\ k{+}{+})$
4. **Do** $(i = 0;\ i < n;\ i{+}{+})$
5. **If** `Active`(i) **Then**
6. Compute $\nabla\psi_{k,i}$
7. Form $\overline{\mathcal{P}}_i^{k+1}$ by adding to $\overline{\mathcal{P}}_i^{k}$ the s largest positions of $\nabla\psi_{k,i}$
8. Gather $A[\overline{\mathcal{P}}_i^{k+1}, \overline{\mathcal{P}}_i^{k+1}]$ and $A[\overline{\mathcal{P}}_i^{k+1}, i]$ from A
9. Solve $A[\overline{\mathcal{P}}_i^{k+1}, \overline{\mathcal{P}}_i^{k+1}]\widetilde{G}[i, \overline{\mathcal{P}}_i^{k+1}]^T = -A[\overline{\mathcal{P}}_i^{k+1}, i]$
10. **If** $\psi_{k,i} \leq \varepsilon \cdot \psi_{0,i}$ **Then**
11. `Active`$(i) \leftarrow 0$
12. **End If**
13. **End If**
14. **End Do**
15. **End Do**

9 of the algorithm call the two functions for gathering and solving dense systems already developed in our previous work [7]. In lines 6 and 7 another function is called to compute $\nabla\psi_{k,i}$, which was not required in the static setup. It will be shown in the next section that the efficient implementation of the latter function is crucial for obtaining good performance.

The ability of aFSAI in capturing the most important entries of A^{-1} allows for the computation of a much sparser factor G with respect to sFSAI. This is not only advantageous in the preconditioner application, but it also enables us to use single-precision arithmetic during setup. As a matter of fact, we will show in section 4 that, for severely ill-conditioned problems, sFSAI may break down in single precision, because it requires the solution of dense systems with up to a few hundred of unknowns. Conversely, aFSAI, solving much smaller dense systems, does not pose the same strong requirement of higher floating-point precision, thus taking better advantage of one of the distinguishing features of the GPU.

3. GPU implementation of FSAI. In this section we describe the features of the CUDA kernels developed to carry out the most computationally intensive phases of the aFSAI setup. These three kernels correspond to

1. the computation of $\nabla\psi_i$, hereafter named the *Kaporin gradient*, and selection of its most relevant components, lines 6 and 7 of Algorithm 1, respectively;
2. the collection of A entries to form $A[\overline{\mathcal{P}}_i, \overline{\mathcal{P}}_i]$ and $A[\overline{\mathcal{P}}_i, i]$, named *systems gather*, line 8;
3. the solution to the sequence of collected dense SPD systems, named *batch Cholesky decomposition*, line 9.

Since the last two kernels have already been described in detail in [7], we simply recall their implementation, whereas we provide more details about the first one.

For a better understanding of our work, we report also some basic information

about the micro-architecture of NVIDIA GPUs as exposed through the CUDA software framework [30], since this is the solution used in our study. From a hardware standpoint, an NVIDIA GPU is an array of *streaming multiprocessors* (SMs); each SM contains a certain number of CUDA *cores*. From a software perspective, a CUDA program is a sequence of computing *grids*; in turn, each grid is split into *blocks*, and each block contains a certain number of *threads*. Each function executed on the GPU on behalf of the CPU is called a *kernel*. To attain a significant fraction of the theoretical peak performance, *occupancy* (i.e., the fraction of active computing elements at a given time) must be consistently kept high, hence thousands of threads must be ready to be scheduled at any time. Threads are executed by an SM in groups of 32 units called *warps*, and performance is significantly improved if threads in the same warp execute the same code with no divergence and access memory according to patterns that privilege *thread* locality, i.e., if threads belonging to the same warp access consecutive memory locations (memory *coalescing*, in CUDA jargon). With every new generation of hardware, incremental improvements for specific memory access needs are introduced. For instance, the NVIDIA Kepler micro-architecture introduced a new warp-level operation called *shuffle* that allows threads belonging to the same warp to exchange data using registers without passing through higher-latency components of the memory hierarchy. We exploit the shuffle primitives in the most computing intensive part of the preconditioner setup as described below.

The GPU device used during the development of the present work is an NVIDIA Tesla P100 (based on the Pascal CUDA architecture) equipped with 56 SMs for a total of 3584 cores and 16 GByte of GDDR5 memory. We also tested a Titan V card (based on Volta, the most recent CUDA architecture) having 5120 cores and 12 GByte of GDDR5 memory. The performance obtained with the GPU kernels is compared to that of a highly tuned CPU implementation of the same kernels executed on an Intel Xeon E5-1620 processor with 4 cores running at 3.50GHz and 64 GByte of RIMM synchronous 2400 MHz 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].

3.1. Kernel for the evaluation of the Kaporin gradient. The computation of the Kaporin gradient of the input matrix is the part of the preconditioner setup that requires more time (on average, $\sim 50\%$ of the total time). In the CPU version, for each row, the computation is repeated until either the convergence criterion is fulfilled or the maximum number of iterations (usually of the order of a few tens) is reached. The evaluation can be carried out independently for each row, so in the CPU version there is an external loop on the rows that is executed in parallel and an internal loop on the refinement iterations of the Kaporin gradient. Since the number of rows is always much higher than the number of available CPU cores, each CPU core actually executes the computation for many rows and, on average, the workload among the CPU cores is well balanced. However, on the GPU the number of cores is much higher (order of thousands on latest generation NVIDIA cards), and having threads that *diverge* (i.e., follow a different execution path), although supported, is strongly discouraged within the CUDA programming model. As a consequence, for the GPU implementation we follow a different approach and, as pointed out in section 2, there is an external loop on the refinement iterations (up to the same maximum number of the CPU version) in which the CUDA kernel for the computation of the Kaporin gradient of (subsets of) rows of A is invoked.

In principle the Kaporin gradient could be computed in parallel for *all* the rows of

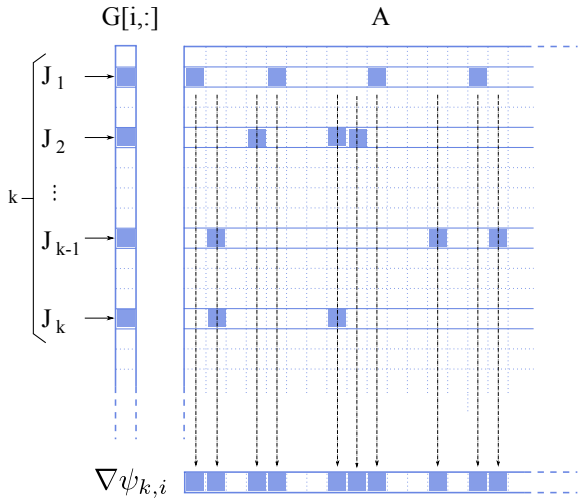

FIG. 3. *Schematic representation of the Kaporin gradient evaluation* $\nabla\psi_{k,i}$ *for row* i *at refinement step* k.

the input matrix; however, due to possible memory constraints imposed by the need of having temporary buffers for each row and the limited (compared to the CPU) amount of *global* memory on the GPU, we chose to execute the Kaporin gradient kernel multiple times on the minimum number of rows required for keeping all the SMs of the GPU busy, taking into account that each row is actually managed by a warp of threads. When the evaluation of the Kaporin gradient reaches the convergence criterion for a row, we set a flag in an array of boolean values considering it as "completed" and skip its evaluation in the successive iterations of the external loop. In Algorithm 1, such array is called `Active`. Within the kernel, we use a warp for each row to exploit at its best the memory bandwidth between the GPU global memory and the SM. By using a warp, nonzero elements (that are stored consecutively in the compressed sparse row (CSR) format) of any row can be loaded with the minimum number of memory transactions (maximizing the memory *coalescing* as suggested by the CUDA programming best practices). The actual computation of the Kaporin gradient consists of two steps: (i) the accumulation of the scaled elements on the same column, and (ii) the selection of the element having the largest absolute value. The accumulation is schematically described in Figure 3 and, from an implementational standpoint, corresponds to a sparse matrix by sparse vector product. More specifically, at the kth refinement iteration, the rows $j_1, j_2, \ldots, j_k$ corresponding to the k nonzero columns of $\widetilde{G}[i,:]$ are scaled and summed together to form $\nabla\psi_{k,i}$.

The main difficulty in this sparse accumulation is to keep track of which columns have been already selected, since the first time a column is selected, its value is simply copied in $\nabla\psi_{k,i}$, whereas for successive selections the value must be summed to the previous one. For the CPU implementation, an effective way to handle this sparse accumulation relies on three temporary arrays. The first two, `WI` and `WR`, integer and real (single or double), have a relatively small size (on the order of a few times the maximum number of nonzeros per row). They are used to store unsorted column indices and nonzero entries of $\nabla\psi_{k,i}$, respectively. The type of the third one, `JWN`, is *short* integer. Its size is equal to n and works as a nonzero indicator: its jth value is

zero if the jth entry of $\nabla\psi_{k,i}$ is zero; otherwise its value gives the position in `WI` and `WR` of the nonzero entry. Unfortunately, the simple and effective solution of having a temporary array with a size equal to the number of columns cannot always be used when the evaluation of the gradient is carried out for thousands of rows concurrently. For instance, if the number of possible columns is on the order of one million, the temporary space required would be equal to $1000000 \times 10240 \times \texttt{sizeof}(\textit{short int})$, where 10240 is the number of rows whose Kaporin gradient can be computed in parallel on a Titan V. This corresponds to a work space of ∼20 GBytes, which is obviously too much since most GPUs have no more than 16 GByte of *global* memory and only a few, recently announced top cards have 32 GBytes. To overcome this problem, we developed two alternative solutions: The first one is used for relatively small matrices (up to ∼200000 rows), and it actually resorts to the three temporary arrays as described above (the required work space is about 2 GBytes on a Titan V card). When the size of the matrix is bigger, we combine the columns using a multiple-arrays-merge-sort algorithm. Algorithm 2 shows pseudocode for this variant of the accumulation phase (kernel `kapGrad_merge`). Each thread in the warp dedicated to the evaluation of the Kaporin gradient for a row is in charge of one of the other rows involved in the iterative procedure. The symbol % is the *mod* operator that is used to ensure that $0 \leq \texttt{lane} < \texttt{WARP\_SIZE}$. Note that the iteration index k also determines the number of entries in the ith row of the current pattern of $\widetilde{G}$. Since the same column indices could be present (meaning that the corresponding value is not equal to zero) for more than one row, the accumulation is done by using the CUDA primitive *atomicAdd* (line 14). The selection of the element having the maximum absolute value among those accumulated during the first phase in the `WS` array is carried out by all threads in the warp. Each thread in the warp computes a candidate maximum by scanning with a *stride* equal to `WARP_SIZE` the `WS` array. Then the maximum among all the candidates is determined by using the CUDA *shuffle* primitives that allow one to exchange values stored in the registers among threads belonging to the same warp, as shown in Algorithm 3, with `lane` and `wsindex` having the same meaning as in Algorithm 2. Also the COMPUTEMIN function (line 9 of Algorithm 2) resorts to *shuffle* primitives (obviously, in the evaluation of the minimum, the test comparison at line 5 changes from $>$ to $<$). The exchange pattern used in the __shfl_xor primitive in Algorithm 3 corresponds to the "butterfly diagram" used in the FFT/DFT transform. The kernel, compiled with CUDA 9.0, requires 31 registers per thread. In order to maximize *occupancy* (i.e., the utilization of the GPU resources), the launch bounds configuration has been set equal to 1024 threads *per* block and 2 blocks *per* SM.

Besides the present solution, we considered several alternatives, including those based on hash tables already used in packages that implement or require sparse matrix–sparse matrix multiplication [28, 29]. The idea is to precompute just the actual number of resulting nonzero entries, reserve the memory required, and organize it according to a hashing table scheme. Ideally the hash function should assign each key to a unique bucket. However, that is exactly the situation using the array having a size equal to the number of columns (in other words, using the identity as hash function). So the solution based on the single large array represents, somehow, the lower bound of the time required using a hash table. Since in our tests the multiple-array-merge-sort-based solution showed a performance equal to (and, at times, better than; see section 4) that of the single-large-array-based solution, we decided to refrain from implementing a solution based on hash tables.

Algorithm 2 CUDA warp-centric row-merge.

Define `WARP_SIZE`
Define `MAXINT`
Input: matrix A in CSR format: `StartRow`$[n]$, `ColInd`$[nnz]$, `Value`$[nnz]$
Input: k: iteration index ($0 \leq k < k_{max}$)
Output: `WS`

```
1. lane ← threadIndex%WARP_SIZE
2. If lane > k Then
3.    return
4. End If
5. wsindex ← 0
6. icol ← StartRow[lane]
7. jcol ← ColInd[icol]
8. loop
9.    jmin ← COMPUTEMIN(jcol)
10.   If jmin == MAXINT Then
11.      return
12.   End If
13.   If jcol == jmin Then
14.      WS[wsindex] ← atomicAdd Value[jcol]
15.      icol++
16.      jcol ← ColInd[icol]
17.   End If
18.   SYNCHRONIZE_THREADS()
19.   wsindex++
20. End loop
21. End
```

3.2. Kernel for systems gather. The purpose of this kernel (`gatherFullSys`) is to gather the dense local systems $A[\overline{\mathcal{P}}_i, \overline{\mathcal{P}}_i]$ from the input matrix A. Basically this requires using elements of the input matrix according to the pattern determined by the previous kernel for the evaluation of the Kaporin gradient with the size of the dense systems increasing with the iteration count. Moreover, the size of the dense systems is the same for all the rows of the input matrix at each iteration. This is a very different situation with respect to our previous work on the *static* FSAI, where the number of elements in the dense systems varied significantly, so that to achieve a better workload balance we had to 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) [8]. In the present case we follow a simple approach where the building of the dense systems is carried out by a single CUDA thread *per* row of the input matrix A. Each thread executes a code that is very similar to the CPU version, but thousands of rows are processed in parallel (to maximize also in this case the *occupancy* of the GPU). The dense systems are stored interleaving the elements of each matrix in a way that could be defined as a *matrix of arrays* so that if $a^l_{i,j}$ is the element in row i, column j of matrix l (corresponding to row k of the input matrix), the memory layout is (assuming row major order) $a^0_{0,0}, a^1_{0,0}, \ldots, a^{n-1}_{0,0}, a^0_{0,1}, a^1_{0,1}, \ldots, a^{n-1}_{0,1}, \ldots, a^{n-1}_{k-1,k-1}$, where n is the number of rows

Algorithm 3 *Shuffle*-based evaluation of the max.

Define `WARP_SIZE`
Input: `WS`
Output: `max`

```
1. max ← 0
2. Do (i = lane; lane < WSINDEX; i += WARP_SIZE)
3.    If (abs(WS[i]> max) Then
4.       max = abs(WS[i])
5.    End If
6. End Do
7. offset ← WARP_SIZE/2
8. Do (offset > 0)
9.    scratch ← __shfl_xor(max, offset)
10.   If (scratch > max) Then
11.      max ← scratch
12.   End If
13.   offset ← offset / 2
14. End Do
15. return max
```

of the input matrix and k is the iteration index. This organization improves *thread locality* and provides a significant advantage in the successive Cholesky decomposition. Note that since the matrices are symmetric, only their upper triangular part is stored in global memory. In the kernel there are only memory copy operations (from the input matrix A to the small dense local systems $A[\overline{\mathcal{P}}_i, \overline{\mathcal{P}}_i]$. Nevertheless it is the second most time-consuming kernel, taking slightly less than 30% of the total execution time. We developed an alternative version using a warp per row. However, its performance was not better than the simple version using a thread *per* row, so we dropped it.

3.3. Kernel for batched Cholesky decomposition. The third most time-consuming part of the preconditioner set-up is the Cholesky decomposition of the small dense matrix that is built for each row of the input matrix (kernel `cudacholdc`). In our GPU implementation of the static FSAI, the Cholesky decomposition was the most time-consuming part. The reason was that the size of the dense matrices was large (up to 256) compared to the present case in which, usually, it is no more than 30. This significant difference in the relative weight of Cholesky in sFSAI and aFSAI is easily explained recalling that the computational cost of the decomposition for an $N \times N$ matrix is $\sim N^3/3$. Based on the extensive set of experiments we carried out in our previous work, evaluating several alternatives, we already knew that for matrices of this size the best solution is to use a single thread *per* system provided that the memory layout of the small dense matrices is the one described in section 3.2. In this way, we limit the number of memory transactions to the (slow) global memory to the bare minimum and thus performance is limited mainly by instructions latencies and dependencies. We take advantage also from the much higher memory bandwidth (up to 3×) provided by the Pascal and Volta architectures with respect to the cards based on the Kepler architecture that we used in our previous work. The final result is that the Cholesky decomposition takes less 15% of the total execution time, on average.

We show an example of the contribution of the three main kernels in the following

fragment of the CUDA profiler (`nvprof`) output for a medium-sized matrix:

```
Time(%)      Time   Calls       Avg       Min       Max  Name
47.58%  112.02ms     420  266.71us  3.1680us  706.08us  kapGrad_merge
29.85%  70.282ms      30  2.3427ms  630.24us  3.3261ms  gatherFullSys
13.50%  31.772ms      30  1.0591ms  5.4720us  2.8277ms  cudacholdc
...
```

The three kernels take more than 90% of the time. The remaining kernels are used to check if the convergence criterion for the refinement has been reached and to build, in the end, the preconditioner. None of them takes more than 0.5% of the total execution time and are all pretty simple.

3.4. Multi-GPU implementation. As already mentioned, one of the advantages of the adaptive FSAI is that the computations can be carried out in a concurrent way for all the rows of the input matrix. If more than one GPU is available, it is straightforward to split the work among the available GPUs, taking into account the differences in terms of raw performance in case the GPUs are not the same model (or architecture). Each GPU is in charge of a subset of the rows of the input matrix and, in the end of the preconditioner setup, all the subparts of the preconditioner are copied back to the most powerful GPU available for the solution of the preconditioned system (at this time in the iterations of the conjugate gradient we simply invoke the suitable primitives of the CUSPARSE library). Using more than one GPU is highly efficient, as shown in section 4, where we present the experimental results.

4. Numerical results. In this section, we present the results obtained with the CUDA implementation of the aFSAI, comparing them with those produced by our highly tuned CPU version. For the tests we used 16 SPD matrices with a number of unknowns growing from 150,000 to 3,000,000. The main features of the matrices—number of rows, nonzero entries, and average number of nonzero per row—are reported in Table 1. Most of the matrices come from the SuiteSparse Matrix Collection [13]. The others, indicated by an asterisk, arise from the discretization through the finite element method of either the Cauchy indefinite equilibrium equations or the Poisson equation. A numerical ID is associated to each matrix to ease the notation in the tables and figures that follow. In the numerical experiments, the right-hand side $\boldsymbol{b}$ corresponds to a unitary solution, i.e., $\boldsymbol{b} = A\mathbf{1}$, the initial PCG vector is $\boldsymbol{x}_0 = G^T G\boldsymbol{b}$, and the convergence is considered achieved when the relative (iterative) residual falls below 10^{-6} in the Euclidean norm. The choice $\boldsymbol{x}_0 = G^T G\boldsymbol{b}$ as initial vector is quite standard, since it allows for taking advantage of the preconditioner from the beginning of the iterative process. From a practical viewpoint, however, other $\boldsymbol{x}_0$ choices would only slightly change the iteration count in the selected examples. In all the tests, except when otherwise specified, we compute the preconditioner using single-precision floating-point arithmetic, then upgrade to double precision for the iterations of the PCG.

As for the setup parameters, the default values are $k_{\max} = 30$, $s = 1$, and $\varepsilon = 10^{-3}$. The GPU and CPU platforms used for the experiments are those described in section 3. Both the memory required to store the aFSAI factor G and the time required for its application to a vector grow with the density μ:

$$\mu = \frac{\mathrm{nnz}(G)}{\mathrm{nnz}(A)}, \tag{18}$$

where $\mathrm{nnz}(\cdot)$ is the function returning the number of nonzeros of its argument matrix.

TABLE 1

Size, number of nonzeros and number of nonzeros per row of the matrices used as benchmark in the numerical experiments.

ID	Matrix	nrows	Nonzeros	nnz_{avg}	Application field
01	Res_small*	137,140	6,234,348	45.46	P1 geomechanical model
02	Cube*	190,581	7,531,389	39.52	Q1 structural problem
03	pwtk	217,918	11,634,424	53.39	Pressurized wind tunnel
04	hood	220,542	10,768,436	48.83	Structural problem
05	BenElechi1	245,874	13,150,496	53.48	2D/3D problem
06	Mexico*	297,945	12,294,999	41.27	P1 subsurface structural model
07	msdoor	415,863	19,173,163	46.10	Structural problem
08	Fault_639	638,802	27,245,944	42.65	Contact mechanics
09	Pflow_742	742,793	37,138,461	50.00	3D pressure in reservoir
10	Res_big*	910,122	40,019,202	43.97	P1 geomechanical model
11	Emilia_923	923,136	41,005,206	44.42	3D geomechanics
12	thermal2	1,228,045	8,580,313	6.99	Thermal problem
13	StocF-1465	1,465,137	21,005,389	14.34	CFD
14	G3_circuit	1,585,478	7,660,826	4.83	Circuit simulation
15	pi8grid8*	2,689,537	18,816,513	7.00	2D anisotropic Laplacian
16	Bump_2911	2,911,419	127,729,899	43.87	3D geomechanics

Results are reported in terms of CPU/GPU time for the following phases: preconditioner setup (T_p), solution with PCG (T_s), and total time (T_t), which is the sum of the previous two values. When needed, also the time for the Kaporin gradient computation (T_{kg}) is shown.

4.1. Comparison between static and adaptive FSAI. This section aims at showing the advantages provided by the aFSAI by comparing its performance and usability with respect to the sFSAI. We briefly recall that the user parameters to control the static setup are the following:

1. τ, the prefiltration tolerance;
2. k, the power to which the original pattern is raised;
3. δ, the postfiltration tolerance.

Although the number of parameters is the same as in aFSAI, combining them in an optimal way is by no means a simple task, as pointed out in other works [20, 7]. The optimal combination of τ, k, and δ depends strongly on the matrix under study, and it may be completely different when the matrix changes as shown in Table 2. Moreover, a slight change in k or in τ may cause an abrupt increase in the preconditioner density with a consequent outbreak of the setup time.

From Table 1, we selected four matrices and tuned the user parameters to minimize the total solution time, i.e., the time to build the preconditioner and execute the PCG iteration, on the GPU card described above using 8-byte arithmetic. Table 2 provides setup parameters, G density, and computing times for sFSAI (top table) and aFSAI (bottom table). Note that for sFSAI, two densities, μ_0 and μ, are provided which are related to the preconditioner before and after postfiltration, respectively. Even though μ_0 measures data only temporarily stored, it still has an impact on the overall memory footprint. It is apparent, looking at Table 2 and Figure 4, as well, that aFSAI always outperforms sFSAI in terms of total solution time. Moreover, it is clear that the sFSAI shows a high variability of the setup parameters for the different matrices, whereas only slight changes are needed for aFSAI. Finally, we note that while the final G factors have comparable sizes, sFSAI always requires the preliminary computation of a significantly denser factor in order to compute an effective preconditioner (see Figure 5).

TABLE 2
Density, iterations and time performance of sFSAI (top) and aFSAI (bottom) with optimal setup.

ID	τ	k	δ	μ_0	μ	Iter	T_p [s]	T_s [s]	T_t [s]
05	0.00	1	0.05	0.509	0.211	2262	0.34	1.82	2.16
08	0.05	4	0.05	0.412	0.149	384	1.11	0.61	1.72
10	0.05	3	0.05	0.269	0.125	351	1.71	0.70	2.41
11	0.05	5	0.05	0.310	0.158	644	1.56	1.36	2.92

ID	$k_{\max}$	s	ε	μ	Iter	T_p [s]	T_s [s]	T_t [s]
05	20	1	0.001	0.392	1405	0.60	1.39	1.99
08	10	1	0.01	0.204	338	0.29	0.75	1.04
10	10	1	0.05	0.136	360	0.23	1.44	1.66
11	10	1	0.01	0.204	454	0.49	1.30	1.79

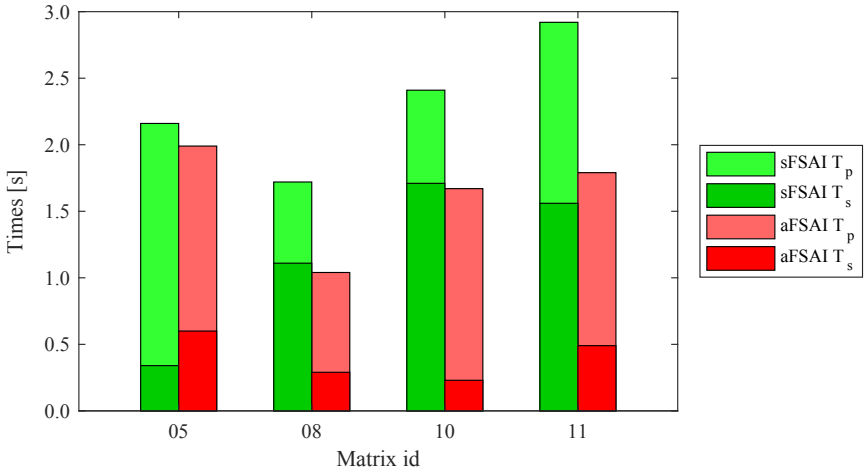

FIG. 4. *Time for preconditioner computation T_p and for PCG iteration T_s for sFSAI and aFSAI. (See color online.)*

4.2. Time performance and memory requirements of the algorithms for computing the Kaporin gradient. As explained in section 3 we investigated two different algorithms for the Kaporin gradient computation. The first one, which resembles the CPU implementation, uses a work array of size n as a nonzero indicator for each row whose Kaporin gradient is computed in parallel. Since thousands of rows are processed concurrently, its memory requirements impose a serious limitation on the size of the matrix. The second one is based on a multiple-arrays-merge-sort and is much lighter. The memory limits of the first approach allow us to compare only the first two matrices out of the 16 we used for all the other tests.

In more detail, both variants of the kernel for the computation of the Kaporin gradient process in parallel a number of rows set with the aim of exploiting all the computing capabilities of the GPU in use. This requires taking into account both the number of SMs and the number of threads that each SM supports. For the Titan V, the resulting number of rows processed in parallel is 10240. With the first approach, for each of the 10240 rows, there is the need of an array of short integers (used as nonzero indicator) having as many elements as the total number of matrix columns. In formula, $10240 \times N_{columns} \times \texttt{sizeof}(\textit{short int})$ bytes of memory. For a relatively small matrix, having $N_{columns} = 200000$, this means $\sim$ 4 GByte. The approach

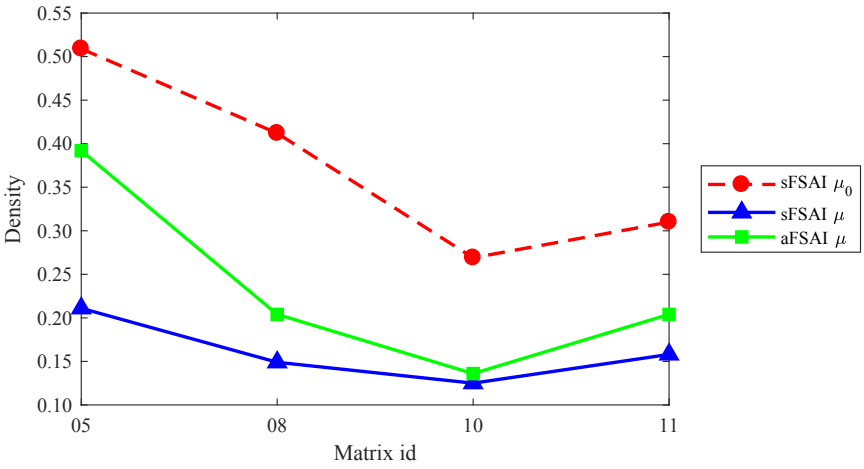

FIG. 5. *Factor density for sFSAI, before (circles) and after (triangles) postfiltration, and aFSAI (squares).*

TABLE 3

Comparison in terms of time performance and memory requirements between the global-array and the multiple-arrays-merge-sort algorithms for the Kaporin gradient computation.

ID	$k_{\max}$	μ	Global array		Multiple-merge	
			T_{kg} [s]	Memory [MB]	T_{kg} [s]	Memory [MB]
01	10	0.198	0.05	2714.5	0.05	35.3
	20	0.297	0.15	2750.4	0.13	71.9
	30	0.334	0.22	2786.3	0.22	107.8
02	10	0.278	0.08	3753.5	0.09	31.2
	20	0.531	0.35	3784.8	0.34	62.5
	30	0.776	0.96	3816.0	0.89	93.7

based on the multiple-arrays-merge-sort needs just an array of integers having size $k_{\max} \times \text{nnz}_{avg}$ for each row processed. Then the total amount of memory (on a Titan V) is $10240 \times k_{\max} \times \text{nnz}_{avg} \times \texttt{sizeof}(int)$. For a matrix with an average number of nonzero elements *per* row equal to ∼50, this corresponds to a few tens of megabytes (assuming $k_{\max} = 30$).

Table 3 compares time and memory required by the two algorithms with increasing k_{max} on the two smallest test matrices, 01 and 02. The multiple-arrays-merge-sort algorithm not only proves slightly faster in terms of time, but also allows for a huge memory saving. For the above reasons, our final implementation adopts only the multiple-arrays-merge-sort algorithm.

4.3. On the use of single-precision arithmetic in the aFSAI setup. To further motivate the choice of aFSAI with respect to sFSAI, we provide the performance gain arising from the use of single-precision arithmetic. To that purpose, we ran the GPU code building aFSAI alternating single and double precision and report the outcome in Table 4 and, graphically, in Figure 6. Using single-precision arithmetic allows for, on average, a 20% reduction of the setup time, without affecting the preconditioner quality as shown by the fact that the iteration count for the convergence never changes significantly. Note that the different arithmetic does not change the PCG solution time T_s because, in any case, we upgrade the preconditioner to double precision by type casting before entering the final iterative solution stage.

TABLE 4
Comparison of single (left) and (double) precision aFSAI on the benchmark matrices.

ID	Iter	T_p [s]	T_s [s]	T_t [s]
01	209	0.22	0.44	0.66
02	506	0.89	0.50	1.39
03	1125	0.98	1.35	2.33
04	126	0.75	0.32	1.07
05	1182	1.29	1.40	2.69
06	647	0.79	1.67	2.46
07	1037	1.83	2.00	3.83
08	228	2.35	1.05	3.40
09	524	2.52	2.40	4.92
10	189	3.78	1.14	4.92
11	223	3.13	1.69	4.82
12	453	2.38	1.54	3.92
13	540	1.09	7.20	8.29
14	235	1.50	3.02	4.52
15	723	2.57	12.65	15.22
16	241	11.57	4.95	16.52

(a) Single precision.

ID	Iter	T_p [s]	T_s [s]	T_t [s]
01	212	0.27	0.44	0.71
02	506	0.95	0.50	1.45
03	1111	1.23	1.43	2.66
04	126	0.95	1.34	2.29
05	1191	1.59	1.45	3.04
06	646	0.95	1.67	2.62
07	1036	2.30	1.98	4.28
08	228	2.89	1.05	3.94
09	524	3.28	2.40	5.68
10	189	4.64	1.14	5.78
11	223	3.95	1.69	5.64
12	453	3.51	1.54	5.05
13	540	1.33	7.20	8.53
14	235	2.02	3.02	5.04
15	723	3.65	12.65	16.30
16	241	14.40	4.95	19.35

(b) Double precision.

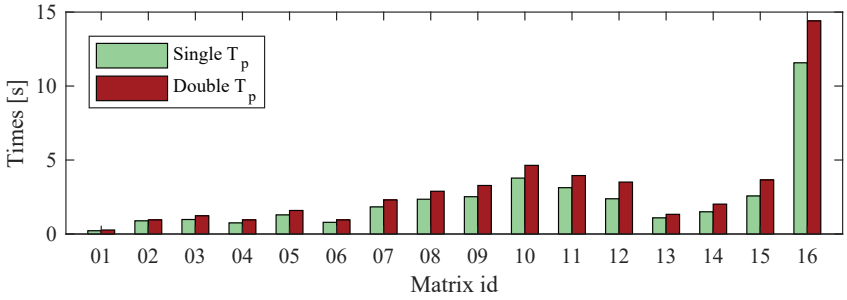

FIG. 6. *aFSAI setup time with single- and double-precision arithmetic.*

By contrast, using single precision is not always possible while using sFSAI, because ill-conditioned problems may originate breakdowns during the Cholesky decompositions of dense linear systems. To better illustrate this issue, we report in Figures 7 and 8 the size (left) and conditioning (right) of the sequence of linear systems arising while setting up sFSAI and aFSAI. We observe that static FSAI requires the solution of much larger systems with a size strongly varying from row to row. By contrast, in aFSAI the size of the systems are highly concentrated and all below the k_{max} number of refinement iterations. As a consequence, the maximum conditioning number arising in sFSAI is more than three orders of magnitude larger than in aFSAI and is thus responsible for the breakdown.

4.4. Benchmarking of aFSAI performance on real-world problems. In the last part of the present section we assess the overall performance of the GPU implementation of aFSAI. First of all, we present a comparison with a state-of-the-art CPU implementation of the dynamic pattern FSAI for shared-memory multiprocessors based on OpenMP directives. As a baseline, we present also the CPU performance of Jacobi-PCG, the simplest form of preconditioning, and PARDISO [35], a state-of-the-art multithreaded direct solver available from the Intel MKL library. For these

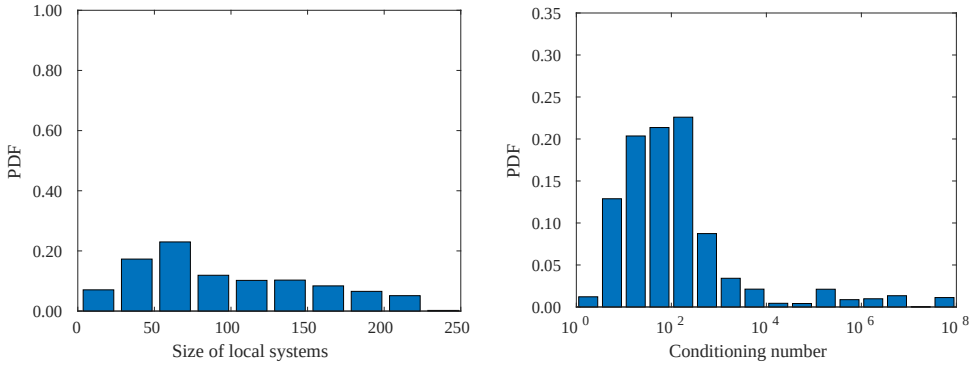

FIG. 7. *sFSAI. Probability density function for the conditioning number (left) and size (right) for all subsystems from matrix* 01.

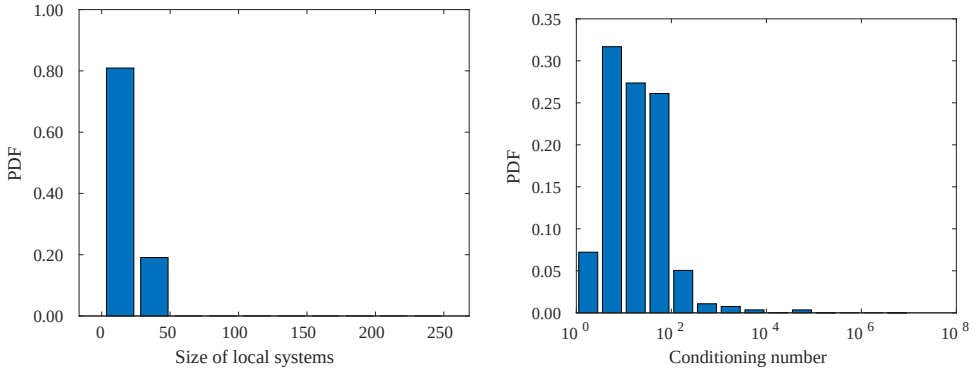

FIG. 8. *aFSAI. Probability density function for the conditioning number (left) and size (right) for all subsystems from matrix* 01.

experiments, we use standard setup parameters, $k_{\max} = 30$, $s = 1$, and $\varepsilon = 10^{-3}$, and ran the tests on the Titan V card and the 4-core Intel 238 Xeon E5-1620 processor. Table 5 provides the performance of Jacobi-PCG and PARDISO. Table 6 compares the performance obtained on GPU (left) and CPU (right) in terms of setup and solution time. We highlight that the G factors differ only for the arithmetic used since the iteration count and densities are identical. Jacobi-PCG provides a very cheap preconditioner in terms of setup time and memory consumption; however, it is significantly outperformed by aFSAI in terms of total time, especially in the most ill-conditioned cases. On the other hand, PARDISO requires always a larger amount of memory and is more effective in the smallest example or those dominated by a two-dimensional geometry, while it suffers in large three-dimensional settings.

For ease of comparison, Figure 9 shows the speed-up offered by the GPU with respect to the CPU in the two phases of the solution, setup (S_p) and PCG (S_s), as well as the total speed-up (S_t). As to the preconditioner computation time, which is the focus of the present work, we observe that an average speed-up of 7.5 is obtained with a minimum only slightly smaller than 6.

Finally we show the almost perfect scalability of our multi-GPU implementation using a standard set of user parameters on matrices 10, 11, and 16 with up to 8 cards. Table 7 reports the setup times along with speed-up varying the number of GPUs.

TABLE 5

CPU performance of Jacobi-PCG and PARDISO. Results are provided in terms of iteration count, setup and PCG times for Jacobi-PCG, and total solution time for PARDISO. μ denotes preconditioner density for Jacobi and number of nonzeros in the factor divided by $nnz(A)$ for PARDISO.

	Jacobi-PCG					PARDISO	
ID	μ	Iter	T_p [s]	T_s [s]	T_t [s]	μ	T_t [s]
01	0.022	> 10000	0.01	32.41	32.42	7.645	2.11
02	0.025	1973	0.02	7.95	7.97	14.932	4.13
03	0.019	5800	0.01	33.34	33.35	4.438	1.09
04	0.020	1579	0.01	9.23	9.24	2.685	0.82
05	0.019	7892	0.02	51.11	51.13	4.333	1.07
06	0.024	3690	0.02	35.18	35.20	15.712	6.04
07	0.022	4697	0.02	47.42	47.44	2.904	1.56
08	0.023	4418	0.03	69.18	69.20	40.497	88.56
09	0.020	9115	0.03	193.26	193.29	14.254	22.53
10	0.023	5220	0.04	121.33	121.37	44.520	156.35
11	0.023	7620	0.04	181.37	181.41	45.124	162.72
12	0.143	2222	0.04	31.37	31.41	6.482	5.38
13	0.070	4658	0.08	105.55	105.63	54.898	58.01
14	0.207	1188	0.05	22.27	22.37	13.944	7.44
15	0.143	5443	0.07	199.91	199.98	9.196	12.68
16	0.023	2611	0.09	202.51	202.60	out of memory	

TABLE 6

CPU vs. GPU performance comparison. Results are presented in terms of iteration count; setup and PCG times with preconditioner density μ also provided.

		GPU				CPU			
ID	μ	Iter	T_p [s]	T_s [s]	T_t [s]	Iter	T_p [s]	T_s [s]	T_t [s]
01	0.334	209	0.22	0.44	0.66	216	2.44	1.09	3.53
02	0.776	506	0.89	0.50	1.39	506	6.64	4.21	10.85
03	0.500	1125	0.98	1.35	2.33	1113	6.27	10.98	17.24
04	0.444	126	0.75	0.32	1.07	127	6.23	1.34	7.57
05	0.577	1182	1.29	1.40	2.69	1191	8.42	14.04	22.46
06	0.392	647	0.79	1.67	2.46	647	7.81	13.01	20.82
07	0.588	1037	1.83	2.00	3.83	1036	11.99	19.01	30.99
08	0.534	228	2.35	1.05	3.40	228	15.26	6.20	21.46
09	0.392	524	2.52	2.40	4.92	524	13.82	16.89	30.71
10	0.570	189	3.78	1.14	4.92	189	26.58	8.09	34.68
11	0.468	223	3.13	1.69	4.82	223	18.23	10.71	28.94
12	4.205	453	2.38	1.54	3.92	452	23.07	18.73	41.80
13	0.608	540	1.09	7.20	8.29	540	5.66	18.95	24.61
14	3.539	235	1.50	3.02	4.52	235	12.28	9.06	21.34
15	1.997	723	2.57	12.65	15.22	723	22.35	52.54	74.89
16	0.513	241	11.57	4.95	16.52	241	82.42	38.88	121.30

A graphical comparison of the speed-up obtained from the experiments and the ideal one (equal to the number of GPUs) is shown in Figure 10.

5. Conclusions. We presented two new approaches to implement aFSAI on GPU. The main goal was running an effective PCG solely on the GPU with minimal interaction with the host CPU after the initial data transfer. While the iteration phase relies only on standard library calls, as those provided by, e.g., the CUSPARSE, the preconditioner computation with a dynamic determination of the nonzero pattern is completely new. Some kernels were already available from a previous work on static FSAI, such as the Cholesky decomposition. However, the SpM-SpV product required

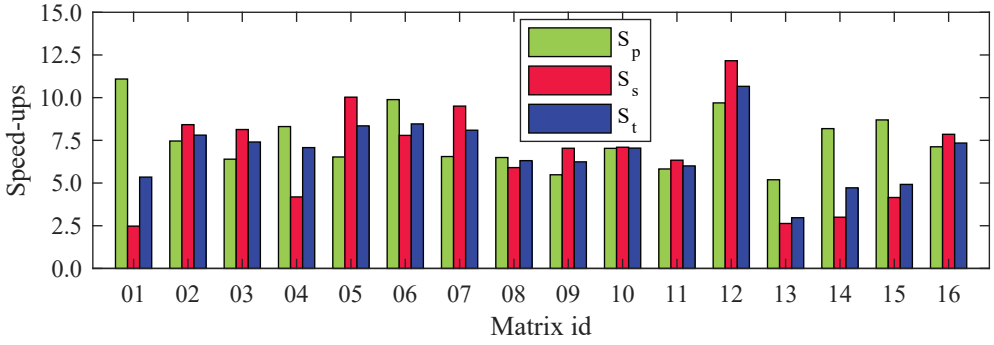

FIG. 9. *Setup, PCG, and total speed-ups for aFSAI on GPU over aFSAI on CPU for the whole set of matrices.*

TABLE 7
Setup times and speed-ups using multiple GPUs. Tests has been carried out on matrices 10, 11, *and* 16.

	Matrix 10		Matrix 11		Matrix 16	
# GPUs	T_p [s]	Speed-up	T_p [s]	Speed-up	T_p [s]	Speed-up
1	3.00	—	2.69	—	9.31	—
2	1.67	1.80	1.49	1.81	4.75	1.96
4	0.91	3.30	0.77	3.49	2.66	3.50
8	0.46	6.52	0.39	6.90	1.52	6.13

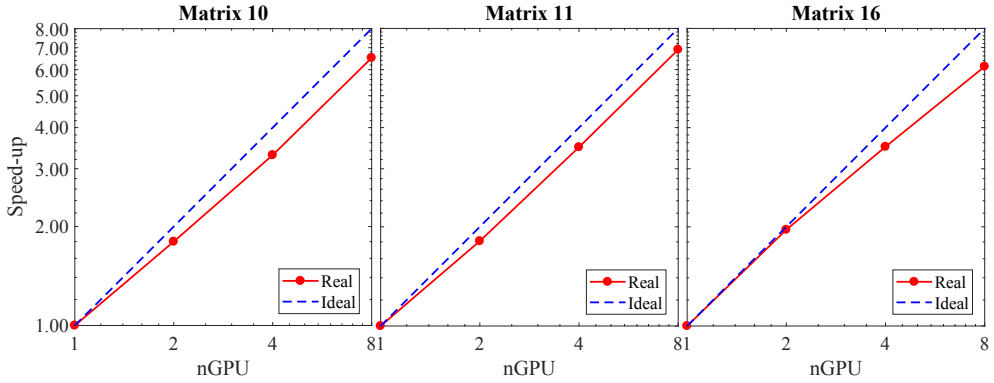

FIG. 10. *Experimental (solid line) and ideal (dotted line) speed-up with up to* 8 *GPU cards. The matrix ID is reported on the top of each plot.*

to minimize the Kaporin number of the preconditioned system has been designed from scratch. The main contributions of this work are the following:

- Two novel solutions have been proposed to efficiently carry out the SpM-SpV product on GPU, one very similar to the CPU implementation and the other based on a merge-sort procedure of multiple arrays. Even if both approaches provide quite the same performance from a computing time viewpoint, the second one proves more suitable for GPU in terms of memory since it requires only a very limited amount of work memory space.
- As already verified on CPU, the adaptive pattern FSAI is superior to sFSAI also on GPU, where we achieve an average speed-up of 7.5 with respect to a parallel CPU on a set of real-world matrices with size up to 3,000,000 rows.

- Single-precision arithmetic, which affects stability in sFSAI, is shown to be feasible within the dynamic pattern setup, allowing for an average 20% reduction of the setup time.
- The strong scalability of our implementation is assessed using up to 8 GPUs, with a resulting speed-up quite close to the ideal one.

The results obtained so far are already very promising, but we believe there is room for further interesting research activities. For instance, supernodal strategies, which have already been successfully tested on CPUs [19], increase data locality and may accelerate the aFSAI setup on GPUs as well. The use of single-precision arithmetic can be partly extended also to the iteration phase, giving rise to a more efficient and faster PCG. Finally, we expect to plug aFSAI as a powerful smoother into an AMG framework running entirely on GPUs.

Acknowledgments. The authors gratefully thank Elena Agostini, Mauro Bisson, and Massimiliano Fatica for the access to NVIDIA multi-GPU systems.

REFERENCES

[1] H. Anzt, T. K. Huckle, J. Bräckle, and J. Dongarra, *Incomplete sparse approximate inverses for parallel preconditioning*, Parallel Comput., 71 (2018), pp. 1–22.

[2] R. Baggio, A. Franceschini, N. Spiezia, and C. Janna, *Rigid body modes deflation of the preconditioned conjugate gradient in the solution of discretized structural problems*, Comput. & Structures, 185 (2017), pp. 15–26, https://doi.org/10.1016/j.compstruc.2017.03.003.

[3] N. Bell, S. Dalton, and L. N. Olson, *Exposing fine-grained parallelism in algebraic multigrid methods*, SIAM J. Sci. Comput., 34 (2012), pp. C123–C152, https://doi.org/10.1137/110838844.

[4] M. Benzi, *Preconditioning techniques for large linear systems: A survey*, J. Comput. Phys., 182 (2002), pp. 418–477, https://doi.org/10.1006/jcph.2002.7176.

[5] M. Benzi, C. D. Meyer, and M. Tůma, *A sparse approximate inverse preconditioner for the conjugate gradient method*, SIAM J. Sci. Comput., 17 (1995), pp. 1135–1149, https://doi.org/10.1137/S1064827594271421.

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

[7] M. Bernaschi, M. Bisson, C. Fantozzi, and C. Janna, *A factored sparse approximate inverse preconditioned conjugate gradient solver on graphics processing units*, SIAM J. Sci. Comput., 38 (2016), pp. C53–C72, https://doi.org/10.1137/15M1027826.

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

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

[10] E. Chow, *A priori sparsity patterns for parallel sparse approximate inverse preconditioners*, SIAM J. Sci. Comput., 21 (2000), pp. 1804–1822, https://doi.org/10.1137/S106482759833913X.

[11] E. Chow, *Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns*, Intern. J. High Performance Comput. Appl., 15 (2001), pp. 56–74, https://doi.org/10.1177/109434200101500106.

[12] E. Chow and Y. Saad, *Approximate inverse preconditioners via sparse-sparse iterations*, SIAM J. Sci. Comput., 19 (1998), pp. 995–1023, https://doi.org/10.1137/S1064827594270415.

[13] T. A. Davis and Y. Hu, *The University of Florida Sparse Matrix Collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1:1–1:25, http://doi.acm.org/10.1145/2049662.2049663.

[14] M. Ferronato, C. Janna, and G. Pini, *Parallel solution to ill-conditioned FE geomechanical problems*, Internat. J. Numer. Anal. Methods Geomech., 36 (2012), pp. 422–437.

[15] R. Gandham, K. Esler, and Y. Zhang, *A GPU accelerated aggregation algebraic multigrid method*, Comput. Math. Appl., 68 (2014), pp. 1151–1160, https://doi.org/10.1016/j.camwa.2014.08.022.

[16] M. J. Grote and T. Huckle, *Parallel preconditioning with sparse approximate inverses*, SIAM

J. Sci. Comput., 18 (1997), pp. 838–853, https://doi.org/10.1137/S1064827594276552.
[17] T. Huckle, *Approximate sparsity patterns for the inverse of a matrix and preconditioning*, Appl. Numer. Math., 30 (1999), pp. 291–303.
[18] C. Janna and M. Ferronato, *Adaptive pattern research for block FSAI preconditioning*, SIAM J. Sci. Comput., 33 (2011), pp. 3357–3380, https://doi.org/10.1137/100810368.
[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, https://doi.org/10.1137/140956026.
[20] C. Janna, M. Ferronato, F. Sartoretto, and G. Gambolati, *FSAIPACK: A software package for high-performance factored sparse approximate inverse preconditioning*, ACM Trans. Math. Softw., 41 (2015), pp. 10:1–10:26, http://doi.acm.org/10.1145/2629475.
[21] I. E. Kaporin, *New convergence results and preconditioning strategies for the conjugate gradient method*, Numer. Linear Algebra Appl., 1 (1994), pp. 179–210, https://doi.org/10.1002/nla.1680010208.
[22] L. Yu. Kolotilina and A. Yu. Yeremin, *Factorized sparse approximate inverse preconditioning* I. *Theory*, SIAM J. Matrix Anal. Appl., 14 (1993), pp. 45–58, https://doi.org/10.1137/0614004.
[23] L. Yu. Kolotilina and A. Yu. Yeremin, *Factorized sparse approximate inverse preconditioning* II: *Solution of* 3*D FE systems on massively parallel computers*, Internat. J. High Speed Comput., 7 (1995), pp. 191–215.
[24] L. Yu. Kolotilina and A. Yu. Yeremin, *Factorized sparse approximate inverse preconditionings*. IV: *Simple approaches to rising efficiency*, Numer. Linear Algebra Appl., 6 (1999), pp. 515–531.
[25] S. Koric, Q. Lu, and E. Guleryuz, *Evaluation of massively parallel linear sparse solvers on unstructured finite element meshes*, Comput. & Structures, 141 (2014), pp. 19–25.
[26] R. Li and Y. Saad, *GPU-accelerated preconditioned iterative linear solvers*, J. Supercomput., 63 (2013), pp. 443–466.
[27] K. Mukundakrishnan, K. Esler, D. Dembeck, J. Shumway, and Y. Zhang, *Accelerating tight reservoir workflows with GPUs*, in SPE Reservoir Simulation Symposium, 2015, pp. 1–15.
[28] Y. Nagasaka, A. Nukada, and S. Matsuoka, *High-performance and memory-saving sparse general matrix-matrix multiplication for NVIDIA Pascal GPU*, in the 46th International Conference on Parallel Processing (ICPP), 2017, pp. 101–110.
[29] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, *AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods*, SIAM J. Sci. Comput., 37 (2015), pp. S602–S626, https://doi.org/10.1137/140980260.
[30] NVIDIA Corporation, *NVIDIA, CUDA C Programming Guide*, pg-02829-001_v7.0, 2015, http://docs.nvidia.com/cuda/cuda-c-programming-guide/.
[31] NVIDIA Corporation, *NVIDIA, CUBLAS Library User Guide*, du-06702-001_v9.2, 2018, https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf.
[32] NVIDIA Corporation, *NVIDIA, CUSPARSE Library User Guide*, du-06709-001_v9.2, 2018, https://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf.
[33] V. A. Paludetto Magri, A. Franceschini, and C. Janna, *A novel algebraic multigrid approach based on adaptive smoothing and prolongation for ill-conditioned systems*, SIAM J. Sci. Comput., 41 (2019), pp. A190–A219, https://doi.org/10.1137/17M1161178.
[34] Y. Saad and H. A. Van der Vorst, *Iterative solution of linear systems in the* 20*th century*, J. Comput. Appl. Math., 123 (2000), pp. 1–33.
[35] O. Schenk and K. Gartner, *Solving unsymmetric sparse systems of linear equations with PARDISO*, Future Generation Computer Systems, 20 (2004), pp. 475–487, https://doi.org/10.1016/j.future.2003.07.011.
[36] 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 Elements Anal. Des., 47 (2011), pp. 387–393.
[37] A. Yu. Yeremin, L. Yu. Kolotilina, and A. A. Nikishin, *Factorized sparse approximate inverse preconditionings* III. *Iterative construction of preconditioners*, Zapiski Nauchnykh Seminarov POMI, 248 (1998), pp. 17–48.