Cache-aware Sparse Patterns for the Factorized Sparse Approximate Inverse Preconditioner

Sergi Laut
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain
sergi.lautturon@bsc.es

Ricard Borrell
Barcelona Supercomputing Center
Barcelona, Spain
ricard.borrell@bsc.es

Marc Casas
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
Barcelona, Spain
marc.casas@bsc.es

ABSTRACT

Conjugate Gradient is a widely used iterative method to solve linear systems $Ax = b$ with matrix A being symmetric and positive definite. Part of its effectiveness relies on finding a suitable preconditioner that accelerates its convergence. Factorized Sparse Approximate Inverse (FSAI) preconditioners are a prominent and easily parallelizable option. An essential element of a FSAI preconditioner is the definition of its sparse pattern, which constraints the approximation of the inverse A^{-1}. This definition is generally based on numerical criteria. In this paper we introduce complementary architecture-aware criteria to increase the numerical effectiveness of the preconditioner without incurring in significant performance costs. In particular, we define cache-aware pattern extensions that do not trigger additional cache misses when accessing vector x in the $y = Ax$ Sparse Matrix-Vector (SpMV) kernel. As a result, we obtain very significant reductions in terms of average solution time ranging between 12.94% and 22.85% on three different architectures - Intel Skylake, POWER9 and A64FX - over a set of 72 test matrices.

KEYWORDS

Conjugate Gradient; FSAI; SpMV

1 INTRODUCTION

Many simulation codes require solving linear systems derived from discretization schemes like finite differences or finite elements, which are numerical methods to solve Partial Differential Equations (PDE). Very frequently, these methods produce very large sparse matrices. Direct methods like the sparse LU factorizations are not useful in this context due to their memory requirements and the significant number of steps they take. Thus, iterative methods are the best option and, in particular, Krylov methods are very commonly applied due to their excellent convergence properties. Krylov methods solve a linear system $Ax = b$ by building a solution within a Krylov subspace, which the method creates by following an iterative process that considers powers of matrix A multiplied by vector b, that is, $\{b, Ab, A^2b, ..., A^mb\}$. When dealing with symmetric and positive definite matrices a popular Krylov method, Conjugate Gradient (CG), is typically applied.

The fundamental kernels involved in the CG method are the Sparse Matrix-Vector (SpMV) product $y = Ax$, the dot-product, and the linear combination of two vectors. The performance of the SpMV is significantly influenced by irregular memory access patterns on x driven by the locations of the sparse matrix non-zero coefficients. As such, SpMV is an expensive memory-bound kernel that requires large memory bandwidth capacity and mechanisms like hardware prefetching. On the other hand, dot products and vector-vector additions typically display regular memory access patterns and achieve significant performance enhancements from those resources and mechanisms.

Besides the performance properties of each individual kernel, another aspect that strongly impacts CG capacity of solving linear systems is the condition number of matrix A. In this context, preconditioners are typically used to improve the convergence properties of CG. From simple Block-Jacobi [34] to sophisticated Multi-Grid techniques [18], a large amount of procedures targeting efficient preconditioning have been proposed. The Sparse Approximate Inverse (SAI) preconditioner consists in evaluating an approximation of the inverse $M \approx A^{-1}$ constrained to a certain sparse pattern [11, 12]. Then, the equivalent and better conditioned system $MAx = Mb$ is solved. In practice, the application of the SAI preconditioner consists in an additional SpMV kernel, which makes it highly parallel. In the context of CG, with symmetric and positive definite problems, the Factorized Sparse Approximate Inverse (FSAI) is applied, which implies that A^{-1} is approximated via a factorization G^TG instead of a single matrix. A very important aspect of FSAI is the definition of its corresponding sparse pattern. While state-of-the-art solutions define this pattern by exclusively taking into account numerical considerations, we demonstrate in this paper that low-level architecture-aware concepts should also be taken into account when defining the FSAI sparse pattern.

This paper proposes and evaluates an approach to extend FSAI sparse patterns based on two fundamental concepts: First, a cache-aware algorithm to extend sparse patterns, which reduces CG iteration count while keeping the cost per iteration low. Such cache-aware optimization relies on low-level aspects of the cache hierarchy architecture like indexing mechanisms or virtual memory management approaches. Second, an approach to filter out the smallest entries of the cache-aware FSAI pattern extension without degrading its convergence properties. This paper makes the following contributions over the state-of-the-art:

- We propose a technique to obtain cache-friendly FSAI sparse patterns. By considering some low-level aspects of the cache hierarchy architecture, our algorithm is able to extend sparsity patterns in a way the number of iterations is reduced while the cost per iteration remains low enough to increase performance. Our approach is architecture independent as it just requires the cache line size as architecture input.
- We propose a robust approach to filter out small entries of the inverse approximation without degrading the numerical properties of the FSAI preconditioner.

- We evaluate our proposals via an extensive evaluation campaign considering 72 matrices of the SuiteSparse Collection [13] that fit in the available memory resources of a single node. Our experiments consider three high-end systems: a 48-core Skylake machine, a 40-cores POWER9, and a 48-cores A64FX. In Skylake, our approach reduces time-to-solution by 15.02% on average. In POWER9 and A64FX, these improvements are 12.94% and 22.85%, respectively.

2 BACKGROUND

This section provides some background information on the PCG solver and the FSAI preconditioner.

2.1 Conjugate Gradient

The Conjugate Gradient (CG) method [34] is an iterative solver for linear systems $Ax = b$, where A is a symmetric and positive definite (SPD) matrix. In the ith iteration, the CG algorithm finds the best solution approximation, x_i, with respect to the A-norm, that belongs to the subspace $x_0 + D^i$; where $D^i = \text{span}\{r_0, Ar_0, ..., A^{i-1}r_0\}$, r_0 is the initial residual, and the A-norm is defined as $\| v \|_A = \sqrt{v^T Av}$.

To find the approximation x_{i+1}, an A-orthogonal basis $\{d_0, d_1, ..., d_i\}$ of D^{i+1} is build by using the conjugate Gram-Schmidt process, that is, by adding a new element, d_i, to the previously derived basis $d_0, ...d_{i-1}$ for D^i:

$$\beta_i = \frac{r_i^T r_i}{r_{i-1}^T r_{i-1}},\ d_i = r_i + \beta_i d_{i-1},$$

where r_i refers to the ith residual and β_i is the Gram-Schmidt conjugation coefficient. x_{i+1} can be represented in this new basis as

$$x_{i+1} = \alpha_0\, d_0 + \ldots + \alpha_{i-1}\, d_{i-1} + \alpha_i\, d_i, \tag{1}$$

where the coefficients $\alpha_0, ..., \alpha_{i-1}$ are obtained from the previous iterations. Hence, in the $i + 1$th iteration it is only necessary to evaluate the component α_i associated to d_i to obtain x_{i+1}:

$$\alpha_i = \frac{r_i^T r_i}{d_i^T A d_i},\ x_{i+1} = x_i + \alpha_i d_i \tag{2}$$

Besides scalar operations, note that only three basic linear algebra operations are required through the steps of the algorithm: Sparse Matrix-Vector product (SpMV), linear combination of two vectors, referred as AXPY in the BLAS terminology, and dot product.

2.2 Sparse Approximate Inverse Preconditioner

Sparse approximate inverse (SAI) preconditioners are based on the assumption that the inverse of the system matrix contains many small entries that can be neglected keeping only the largest ones and, consequently, a sparse approximation can be effective. In the setup process of the SAI method, an approximation of the inverse $M \approx A^{-1}$ constrained to a fixed sparse pattern $\mathcal{S}$ is found. The equivalent, but better conditioned, system $MAx = Mb$ is considered.

The preconditioned version of the CG algorithm [34] requires the inverse of the preconditioner at each time step. For SAI, since the approximation of the inverse is computed explicitly, the preconditioning process consists in an SpMV operation, which makes the algorithm attractive due to the parallelizable nature of the SpMV kernel. The SAI preconditioner has been extensively used for the solution of linear systems from different application areas [2, 4, 30, 33]. The easy portability of the SpMV kernel has also fostered its application to high-end architectures such as GPUs [9, 19, 32].

When dealing with symmetric and positive definite problems, to preserve the system symmetry, the Factorized Sparse Approximate Inverse (FSAI) preconditioner is applied and A^{-1} is approximated by a factorization $G^T G$ instead of a single matrix M, which means that two SpMV products are necessary instead of one. G is a sparse lower triangular matrix approximating the inverse of the Cholesky factor, L, of A. To find G, the problem is formulated as:

$$min_{G \in \mathcal{S}} \|I - GL\|_F^2, \tag{3}$$

where $\|.\|_F$ is the Frobenius norm and $\mathcal{S}$ is a lower triangular sparsity pattern. This problem can be solved independently for each row i of G, which can be obtained by solving the local system $A_{\mathcal{S}_i \mathcal{S}_i} g_i = e_i$,where $A_{\mathcal{S}_i \mathcal{S}_i}$ is the restriction of A to the coefficients of the ith row of the sparse pattern $\mathcal{S}_i$ and e_i is the ith column of the identity matrix restricted to the same space [11, 28].

We apply state-of-the-art techniques [11] to find G without explicitly evaluating L, i. e., only using the initial matrix A. In this context, the sparse pattern $\mathcal{S}$ is defined a priori as the pattern of a power N of $\tilde{A}$, where $\tilde{A}$ is obtained from A by dropping small entries. The power used to fix the sparse pattern is referred as the sparse level of the preconditioner. In Algorithm 1, we show the method proposed by Chow [11] to find G.

Algorithm 1 FSAI, $G^T G \approx A^{-1}$

1: Threshold A to produce $\tilde{A}$.
2: Compute the pattern $\tilde{A}^N$, and let the pattern of G be the lower triangular part of the pattern of $\tilde{A}^N$.
3: Compute the nonzero entries in G by solving the Frobenius minimization problem.
4: Drop small entries in new G and rescale.

3 ACCELERATING FSAI

This section describes a high-level view of our cache-aware sparse pattern extension strategy to boost the FSAI performance. Algorithm 2 displays a reformulation of FSAI by adding one step, the cache-friendly extension of the sparse pattern, and replacing the filtering-out and rescaling steps applied to G by a more complex filtering-out process applied to an approximate precalculation of G before its evaluation. The step added in line 3 extends the sparse pattern of G considering architecture-aware criteria to add additional entries that reduce the CG iteration count while incurring a minimal overhead in terms of iteration cost. This step is further described in Section 4. Note that we propose an extension of the sparse pattern, therefore the set of matrices considered in the Frobenius minimization problem of Equation 3 increases. Consequently, the new inverse approximation is more accurate.

The step added in line 4 replaces the filtration that Algorithm 1 performs after the computation of G. In this new filtration strategy, entries of the sparse pattern are filtered out based on an approximate precalculation of G. The resulting pattern is used to calculate the final G, ensuring a Frobenius-minimal A^{-1} approximation, i. e. the

best approximation in this pattern. This step is further explained in Section 5.

Algorithm 2 FSAI, $G^T G \approx A^{-1}$ with pattern extension and precalculation

1: Threshold A to produce $\tilde{A}$.
2: Compute the pattern $\tilde{A}^N$, and let the pattern, $\mathcal{S}$, of G be the lower triangular part of the pattern of $\tilde{A}^N$.
3: **Compute cache-friendly extension of the pattern of G, $\mathcal{S}_{ext}$.**
4: **Precalculate an approximation $\tilde{G}$ of the preconditioner, and filter out entries of $\mathcal{S}_{ext}$ according to its values.**
5: Calculate G on the sparse pattern obtained from the previous step.

4 CACHE-FRIENDLY FILL-IN

In this section we propose a cache-friendly fill-in approach to extend the sparse pattern of the FSAI preconditioner. We propose architecture-aware techniques to extend the sparse pattern in a way that we achieve significant reductions in terms of iteration count while minimizing the iteration cost overhead. In particular, we propose a method to extend the FSAI sparse pattern without increasing the number of cache misses.

Since FSAI is applied via the SpMV kernel $y = Ax$, we must consider the access patterns for all the involved data structures containing y, A, and x. Assuming that we traverse the sparse matrix A in row order and that we store it using the Compressed Sparse Row (CSR) format, the accesses on the data structures containing A display a very simple stride-1 pattern. Since this pattern is easily predictable by hardware prefetchers, there is some flexibility for extending A without suffering a prohibitive performance penalty. Very similar considerations apply when accessing vector y. The most problematic accesses are those coming from accesses to vector x, which follow a random pattern and thus can not be easily predicted by the prefetcher. Our approach extends matrix A without increasing the number of cache misses due to accesses on x. It is important to state that similar considerations can be made when accessing matrix A in column order and storing it with Compressed Sparse Column (CSC) format, with the only difference that the roles of vectors y and x switch. Without loss of generality, we assume for the rest of this section that the SpMV kernel $y = Ax$ is traversed in row order, and that A is stored in CSR.

4.1 Cache Alignment of Vector x

Our approach drives the extension of the FSAI sparse pattern by taking into account the cache alignment of vector x. The main idea consists in extending A with coefficients that will require elements of x belonging to the same cache line as the elements of x where the initial sparse pattern accesses. In other words, the idea is to add new coefficients in A that fully exploit the spatial locality on memory accesses to vector x. For example, if the first row of the initial sparse pattern accesses a double-precision element x_i located in the first position of a 64Bytes cache line, we will not produce any additional cache miss on x by adding up to seven additional non-zeroes right after the element that accesses x_i.

The approach relies on determining the relative position of vector elements x_i in the cache lines storing them by using its corresponding virtual address. In general, physical and virtual addresses are composed by tag, index and offset bits, being this last subset the one that determines the position within a cache line. Since the first levels of the cache hierarchy are virtually indexed and physically tagged to overlap the indexing process with the Transaction Lookaside Buffer (TLB) traversal, the index and the offset bits of both virtual and physical addresses are the same. Importantly, this means that we can determine the relative position within a cache line of a certain x_i value by looking at the offset bits of its virtual address. If x_i values are 64-bit floating-point numbers, a 64B cache line will store 8 of them. That means that the virtual address modulo 8, $address_{virtual}(x[i])$ mod 8, gives the relative position of x_i within its cache line. When dealing with larger cache line sizes we need to use the corresponding number of elements stored per cache line. For example, when dealing with the 256B cache lines of A64FX, if x_i values are 64-bit floating-point numbers, we will compute the virtual address modulo 32, $address_{virtual}(x[i])$ mod 32.

Our approach is architecture independent since i) it can be adapted to any cache line size by simply adjusting this value before applying the cache-friendly fill-in procedure; ii) it can be adapted to any cache indexing mechanism using virtual addresses; and, finally, iii) it can be applied to any Instruction Set Architecture (ISA). Section 7 shows an exhaustive evaluation of our cache-friendly fill-in considering Skylake, POWER9 and A64FX architectures.

4.2 Algorithm for Cache-Friendly Fill-In

We propose an algorithm to produce a cache-friendly fill-in of a generic FSAI sparse pattern. The inputs of our algorithm are the initial sparse pattern $\mathcal{S}$, and the array x that will be used in the SpMV operation. Algorithm 3 displays the pseudocode of our Cache-Friendly Fill-In algorithm. The main loop iterates over the initial pattern rows and, per each row, traverses all its non-zero entries (lines 4-13). For each entry j, it identifies the cache line of the corresponding x_j component of the multiplying vector (line 9), using the procedure described in Section 4.1. Then it extends the sparse pattern by inserting non-previously existing entries corresponding to the portion of vector x stored in this same cache line (lines 10 and 11). This algorithm can be easily parallelized using threading-based approaches like OpenMP or Posix threads.

4.3 Applying the Cache-Friendly Fill-In to FSAI

As described in Section 2, the FSAI preconditioner approximates A^{-1} by the factorization $G^T G$. The sparse pattern of the original G matrix is extended using the algorithm described in Section 4.2 to obtain an extended G_{ext} matrix. Applying FSAI to a right-hand-side vector p implies computing the product $G_{ext}^T G_{ext} p$. We use the CSR format to store the G_{ext} matrix. The application of Algorithm 3 keeps the number of cache misses triggered by accesses to the p vector when computing $G_{ext}p$ similar as Gp, while the extended number of entries of G_{ext} decreases CG iteration count.

For the G_{ext}^T matrix, we also use the CSR storage format. Although the application of Algorithm 3 does not explicitly target the reduction of cache misses triggered by the G_{ext}^T multiplication, additional entries added in the same row of G belong to the same

Algorithm 3 Cache-Friendly Fill-In

1: $\mathcal{S} \rightarrow$Sparse pattern to extend
2: $x \rightarrow$ Multiplying vector in the SpMV
3: **procedure** ExtendPattern
4: Loop over every row i of pattern $\mathcal{S}$
5: Loop over every entry j in row i
6: If j is in an already considered column block
7: Move to next j
8: Endif
9: Identify the cache line of its corresponding x_j coefficient
10: Compute initial and final columns matching the cache line of x_j
11: Add to the pattern the columns of the block that are not already present
12: Endloop
13: Endloop

column of G^T, and being close to previous entries of G implies belonging to adjacent or almost adjacent rows in G^T. Therefore, when multiplying by G_{ext}^T, the entries generated by the extension are accessed in consecutive rows. In conclusion, the spatial locality optimization for the G_{ext} product results in a temporal locality optimization of the G_{ext}^T product.

Section 7 confirms that applying the G_{ext} matrices reduces the number of CG iterations while keeping the cost per iteration low, which produces significant performance benefits.

4.4 Cache-Friendly Fill-In Process Example

We show in Figure 1 an example of the cache-friendly fill-in process. The left-hand side plot displays an initial sparse lower triangular pattern for a generic 64x64 matrix where the non-zero entries are represented by black squares. The array to its right represents the multiplying vector x. The cache-friendly fill-in Algorithm 3 loops over every entry in the initial pattern. For each entry, it captures the relative position in memory of the corresponding multiplying array component. Suppose the first component of x is stored in double precision and aligned in memory such that its relative position in a 64B cache line is 0. Whenever this entry is accessed, a block of 8 entries composed of the initial one plus the following 7 will be loaded. Every time an entry within this block is accessed, the other seven will be available. However, the initial pattern may not require to access them. The algorithm adds column entries in the pattern such that all loaded array elements are used every time their cache line is accessed (except if they correspond to entries above the diagonal). The center plot in Figure 1 displays a lower triangular pattern that has been extended using Algorithm 3. Black squares correspond to entries in the initial pattern, and grey squares are cache-friendly added entries. Difference in grey color serves as a guide to identify the multiplying array cache line they are added for. The pattern formed by the black and grey squares is the one used to compute the values of extended G. Although this extension targets the optimization of the spatial locality in the SpMV products involving G in FSAI, it also optimizes temporal locality for the products involving G^T.

5 FILTERING OUT SMALL G ENTRIES

Matrix G, computed in Step 3 of Algorithm 1, produces the best approximation G^TG to A^{-1} restricted to the $\mathcal{S}$ sparse pattern with respect to the Frobenius norm (see Equation 3). A common strategy to make the resulting approximation more efficient is to filter out some G entries containing small values in absolute terms. Removing these entries may have computational benefits in terms of accelerating its two associated SpMV products, G^TGp, although it may also degrade the numerical quality of the preconditioner. By filtering out small entries the sparse pattern is modified and the resulting G does not necessarily minimize Eq. 3.

To get the best possible inverse approximation on the final sparse pattern, we propose a new filtration strategy, prior to the evaluation of G, that replaces the common filtration and rescaling steps performed at the end of the preconditioner setup. This new strategy is based on an approximate precalculation of G. To filter out G entries with low absolute value, we just require an approximation describing their order of magnitude. To generate this low-cost approximation, we solve Eq. 3 via several iterations of the CG method with a relatively high tolerance to obtain an approximate solution. This solution allows to discriminate the entries with high absolute values from those with small ones using a scale-independent order of magnitude comparison of non-diagonal entries with respect to diagonal entries. Differently, for the evaluation of G in the resulting sparse pattern we use a direct solver (line 5 of Algorithm 2). Section 7 demonstrates the robustness of this novel approach.

Note that the cache-friendly extension and the new filtration strategies described here can be applied to any given sparse pattern. Section 8 provides a short overview of different existing options to define the initial sparse pattern. In any case, the steps followed to apply our optimizations are:

(1) To extend the given pattern with cache-friendly entries. This step corresponds to line 3 of Algorithm 2.
(2) To precompute G on the extended pattern.
(3) To filter out entries of the extension featuring low absolute values.
(4) To compute G coefficients on the sparse pattern generated in the previous steps using Equation 3. This step corresponds to line 5 of Algorithm 2.

Figure 1 illustrates these steps applied to a 64x64 sparse matrix. The left-hand side image shows the lower triangular part of the initial matrix. In the middle plot, we show how the pattern is cache-friendly extended assuming a cache size of 64B and entries stored in double precision. Note that different cache sizes would lead to different extended patterns. An inverse approximation is precalculated on this extended pattern. The right-hand side image shows in red the final entries of the cache-friendly extension, being the initial entries represented in black.

6 EXPLOITING SPATIAL AND TEMPORAL LOCALITY IN THE Gp AND G^Tp PRODUCTS

This section describes a method to exploit spatial and temporal locality of products Gp and G^Tp, respectively. Previous sections describe how to extend the sparse pattern of matrix G to obtain an extended G_{ext} where the additional entries do not increase the

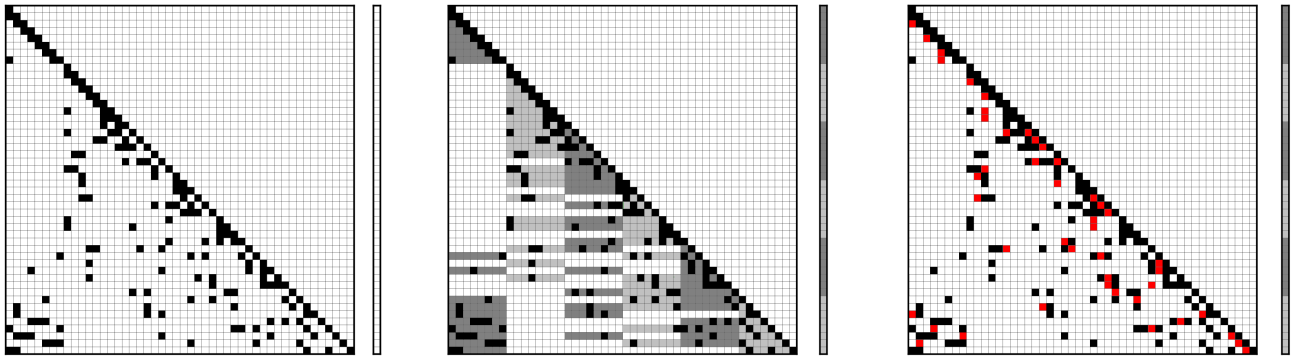


Figure 1: Graphical representation of the pattern extension strategy. Left: Initial lower triangular pattern of a given matrix, A (black squares) plus the multiplying vector x. Center: Cache-friendly pattern extension. Right: Filtered pattern.

number of cache misses triggered by accesses to multiplying vector p when computing $G_{ext}p$. This extension is based on a spatial locality optimization of the product of G, and produces a temporal locality optimization of the product of G^T. The pattern of G can be extended to optimize both spatial and temporal locality for the two SpMV products in FSAI. First, the extension, precalculation and filtration steps have to be applied to G. Second, the extension, precalculation and filtration steps have to be repeated on the extended transposed pattern, G^T_{ext}. Applying the extension to the transposed matrix G^T_{ext} optimizes its spatial locality and the temporal locality of the initial product. As a result the spatial and temporal locality of both products is optimized. It is of utter importance to note that the extensions have to be performed in two steps to ensure the cache-friendliness of all the entries in the extended patterns of both G and G^T. Applying the cache-friendly fill-in algorithm to simultaneously extend G and G^T_{ext}, and precalculating and filtering out the resulting pattern may produce non cache-friendly extended entries. The steps to improve the spatial and temporal locality of FSAI are the following:

(1) Extend initial lower triangular pattern, $\mathcal{S}$, to cache-friendly entries optimizing x accesses in the G product.
(2) Precalculate approximation on the extended pattern.
(3) Filter out the entries in extended positions to obtain $\mathcal{S}_{ext}$.
(4) Extend $(\mathcal{S}_{ext})^T$ to cache-friendly entries optimizing x accesses in the G^T product.
(5) Precalculate approximation on the extended pattern.
(6) Filter out the entries in extended positions to obtain $(\mathcal{S}_{ext})^T_{ext}$.
(7) Calculate final G using the transposed sparse pattern $(\mathcal{S}_{ext})^T_{ext}$.

These steps are compiled in Algorithm 4. Section 7 confirms that using pattern extensions generated via Algorithm 4 reduces the number of iterations while keeping the cost per iteration low, which produces very significant performance benefits.

Algorithm 4 FSAI, $G^TG \approx A^{-1}$ with pattern extension and precalculation optimizing spatial and temporal locality for both products

1: Threshold A to produce $\tilde{A}$.
2: Compute pattern $\tilde{A}^N$, and define the initial pattern $\mathcal{S}$ of G as the lower triangular part of the pattern of $\tilde{A}^N$.
3: Compute cache-friendly extension of pattern $\mathcal{S}$ optimizing Gp product.
4: Precalculate the nonzero entries in the obtained cache-friendly pattern and keep entries larger than parameter $filter$. Define $\mathcal{S}_{ext}$ as the sparse pattern of the extended matrix G_{ext}.
5: Compute cache-friendly extension of S^T_{ext} optimizing $G^T_{ext}p$ product.
6: Precalculate the nonzero entries in the obtained cache-friendly pattern and keep entries larger than parameter $filter$. Define $(\mathcal{S}_{ext})^T_{ext}$ as the sparse pattern of matrix G^T.
7: Compute G coefficients on the transposed pattern $(S_{ext})^T_{ext}$.

7 EVALUATION

7.1 Experimental Setup

The numerical experiments of this paper consider all symmetric and positive-definite matrices with non-zero coefficient count from 48K up to 4.8M of the SuiteSparse Matrix Collection [13]. We exclude the matrices that do not converge after 10000 iterations of the conjugate gradient method preconditioned with FSAI. The 72 resulting matrices along with some of their characteristics are listed in the second, third, fourth, and fifth columns of Table 1. Matrices come from a variety of fields such as Electromagnetics, Computational Fluid Dynamics (CFD), Acoustics, or Circuit Simulation.

We run our experiments in Skylake, POWER9 and A64FX shared-memory systems. The Skylake machine is composed of two 24-core Intel Xeon® Platinum 8160 processors at 2.1GHz with 12x8GB DDR4-2667 DIMMS (2GB/core). The POWER9 system is composed of two 20-core IBM Power9 8335-GTH processors at 2.4GHz with 16x32GB DIMMS. Both Skylake and POWER9 systems have 64Bytes cache lines. The A64FX system is composed of a 48-core A64FX Fujitsu processor at 2.2GHz with 256Bytes cache lines.

The code has been developed in C language and compiled with GCC 10.1.0 on Skylake and POWER9, and GCC 10.2.0 on A64FX. Our thread-based parallel code uses OpenMP 5.0 constructs. To solve the linear systems involved in Step 7 of Algorithm 4, we use

Table 1: Test matrices along with key properties and results. Results are provided as the preprocessing, solving times (in seconds) and iterations to convergence for the basic FSAI and FSAIE(sp) and FSAIE(full) with a 0.01 $filter$, respectively. For the cases of FSAIE(sp) and FSAIE(full) we also provide the percentage of lower triangular pattern entries increase with respect to FSAI pattern after the extensions (% NNZ).

					FSAI			FSAIE(sp)				FSAIE(full)			
ID	**Matrix**	**#rows**	**NNZ**	**Type**	**Setup**	**Solve**	**Iter**	**Setup**	**Solve**	**Iter**	**% NNZ**	**Setup**	**Solve**	**Iter**	**% NNZ**
1	shipsec5	179860	4598604	Structural	9.58E-02	1.08E+00	1615	3.01E-01	1.01E+00	1465	14.95	5.39E-01	1.04E+00	1437	20.82
2	offshore	259789	4242673	Electromagnetics	8.91E-02	8.97E-01	782	2.82E-01	8.25E-01	771	16.39	6.82E-01	9.07E-01	751	30.86
3	smt	25710	3749582	Structural	3.14E-01	4.32E-01	884	9.78E-01	3.06E-01	574	20.49	1.78E+00	2.95E-01	515	33.19
4	parabolic_fem	525825	3674625	CFD	6.45E-02	2.26E+00	1460	2.16E-01	1.87E+00	1144	65.78	5.31E-01	1.67E+00	1054	119.98
5	Dubcova3	146689	3636643	2D/3D	7.36E-02	1.19E-01	153	2.99E-01	1.46E-01	138	62.55	5.89E-01	1.22E-01	107	110.01
6	shipsec1	140874	3568176	Structural	8.53E-02	1.10E+00	1985	2.26E-01	1.09E+00	1982	14.49	4.39E-01	1.12E+00	1945	19.49
7	nd3k	9000	3279690	2D/3D	1.75E+00	1.97E-01	406	4.19E+00	1.76E-01	357	2.07	6.05E+00	1.65E-01	336	3.03
8	cfd2	123440	3085406	CFD	5.81E-02	1.21E+00	2600	2.01E-01	1.14E+00	1969	74.60	4.17E-01	1.21E+00	1862	120.11
9	nasasrb	54870	2677324	Structural	8.90E-02	1.10E+00	2768	1.96E-01	1.11E+00	2761	5.95	2.92E-01	1.10E+00	2739	8.87
10	oilpan	73752	2148558	Structural	5.94E-02	5.85E-01	1620	1.24E-01	5.53E-01	1452	24.81	2.22E-01	5.36E-01	1326	47.70
11	cfd1	70656	1825580	CFD	4.74E-02	3.56E-01	932	1.49E-01	3.33E-01	801	62.48	3.15E-01	3.41E-01	739	113.35
12	qa8fm	66127	1660579	Acoustics	4.12E-02	4.14E-03	13	9.71E-02	3.58E-03	11	22.44	1.68E-01	3.57E-03	11	28.70
13	2cubes_sphere	101492	1647264	Electromagnetics	4.74E-02	5.60E-03	12	1.24E-01	5.74E-03	12	10.20	2.50E-01	5.34E-03	11	17.30
14	thermomech_dM	204316	1423116	Thermal	6.11E-02	5.80E-03	9	9.11E-02	5.85E-03	9	2.08	1.97E-01	5.79E-03	9	2.42
15	msc10848	10848	1229776	Structural	1.85E-01	2.18E-01	712	3.50E-01	2.04E-01	651	8.36	6.03E-01	1.64E-01	528	21.51
16	Dubcova2	65025	1030225	2D/3D	4.31E-02	6.04E-02	158	1.08E-01	5.60E-02	131	88.81	2.15E-01	4.90E-02	106	162.91
17	gyro	17361	1021159	Model Reduction	7.22E-02	1.72E+00	4457	1.55E-01	1.11E+00	3576	25.93	2.67E-01	1.38E+00	3400	35.16
18	gyro_k	17361	1021159	DMR	7.30E-02	1.54E+00	4444	1.49E-01	1.24E+00	3599	25.93	2.60E-01	1.02E+00	3450	35.16
19	olafu	16146	1015156	Structural	5.42E-02	4.17E-01	1782	1.10E-01	3.80E-01	1524	14.00	1.71E-01	3.15E-01	1336	22.64
20	bundle1	10581	770811	CG/V	1.08E-01	6.82E-03	22	2.11E-01	5.83E-03	20	0.01	3.00E-01	6.04E-03	20	0.01
21	G2_circuit	150102	726674	Circuit Simulation	3.25E-02	3.84E-01	1026	6.69E-02	3.23E-01	808	146.97	1.11E-01	3.18E-01	772	215.71
22	Pres_Poisson	14822	715804	CFD	5.85E-02	6.53E-02	285	9.99E-02	3.89E-02	160	35.75	1.62E-01	3.23E-02	130	61.49
23	thermomech_TC	102158	711558	Thermal	3.50E-02	3.94E-03	9	7.83E-02	3.90E-03	9	3.13	1.14E-01	3.96E-03	9	3.65
24	cbuckle	13681	676515	Structural	5.07E-02	2.48E-02	114	8.83E-02	2.55E-02	108	12.55	1.38E-01	2.30E-02	101	24.08
25	finan512	74752	596992	Economic	3.12E-02	2.88E-03	10	5.95E-02	2.72E-03	9	34.16	8.94E-02	2.79E-03	9	42.53
26	crystm03	24696	583770	Materials	2.91E-02	3.45E-03	13	6.04E-02	3.51E-03	12	14.04	9.75E-02	3.26E-03	11	26.34
27	thermal1	82654	574458	Thermal	3.19E-02	2.80E-01	735	5.85E-02	2.24E-01	603	95.58	1.34E-01	2.26E-01	532	189.89
28	wathen120	36441	565761	Random 2D/3D	2.97E-02	6.10E-03	25	5.20E-02	4.88E-03	19	76.92	8.82E-02	4.85E-03	19	98.41
29	apache1	80800	542184	Structural	2.89E-02	4.43E-01	1663	4.46E-02	4.17E-01	1582	66.59	6.92E-02	4.32E-01	1574	73.41
30	gridgena	48962	512084	Optimization	3.06E-02	4.32E-01	1729	4.39E-02	3.49E-01	1350	80.29	8.16E-02	3.23E-01	1205	141.49
31	wathen100	30401	471601	Random 2D/3D	4.20E-02	4.67E-03	25	4.65E-02	3.76E-03	19	76.75	7.21E-02	3.84E-03	19	98.18
32	bcsstk17	10974	428650	Structural	3.42E-02	1.27E-01	627	5.83E-02	1.09E-01	551	15.84	9.04E-02	1.08E-01	491	28.78
33	cvxbqp1	50000	349968	Optimization	3.77E-02	1.60E+00	5032	4.29E-02	1.58E+00	5051	0.10	6.37E-02	1.61E+00	5045	0.22
34	Kuu	7102	340200	Structural	4.34E-02	3.01E-02	147	7.40E-02	2.75E-02	128	25.63	1.07E-01	2.57E-02	115	44.54
35	shallow_water2	81920	327680	CFD	4.47E-02	3.42E-03	14	3.79E-02	3.07E-03	12	99.87	5.79E-02	2.72E-03	10	161.23
36	shallow_water1	81920	327680	CFD	3.00E-02	2.00E-03	8	3.82E-02	1.94E-03	7	43.21	5.21E-02	1.50E-03	6	59.76
37	crystm02	13965	322905	Materials	3.52E-02	3.05E-03	13	5.83E-02	2.61E-03	12	13.17	7.05E-02	2.79E-03	11	18.40
38	bcsstk16	4884	290378	Structural	3.47E-02	2.32E-02	83	5.92E-02	2.34E-02	80	11.83	8.83E-02	2.38E-02	79	16.08
39	s2rmq4m1	5489	263351	Structural	2.96E-02	7.46E-02	360	4.83E-02	7.57E-02	356	9.71	5.80E-02	7.64E-02	353	17.41
40	s1rmq4m1	5489	262411	Structural	3.25E-02	6.17E-02	299	4.16E-02	6.29E-02	299	13.27	5.84E-02	6.29E-02	290	20.99
41	Dubcova1	16129	253009	2D/3D	2.68E-02	1.75E-02	84	5.10E-02	1.49E-02	67	89.09	7.57E-02	1.26E-02	55	167.32
42	bcsstk25	15439	252241	Structural	2.76E-02	6.97E-01	3880	3.82E-02	6.49E-01	3584	24.04	5.16E-02	6.31E-01	3366	38.13
43	bcsstk28	4410	219024	Structural	3.31E-02	2.21E-01	1003	4.78E-02	1.81E-01	849	19.23	8.64E-02	1.62E-01	715	39.46
44	s2rmt3m1	5489	217681	Structural	2.74E-02	7.72E-02	384	4.04E-02	7.78E-02	365	16.28	5.45E-02	7.30E-02	350	29.05
45	s1rmt3m1	5489	217651	Structural	3.22E-02	6.36E-02	320	3.78E-02	6.22E-02	310	20.11	5.54E-02	5.90E-02	301	32.16
46	minsurfo	40806	203622	Optimization	2.67E-02	9.21E-03	42	3.95E-02	7.43E-03	32	228.76	5.14E-02	6.95E-03	29	356.20
47	jnlbrng1	40000	199200	Optimization	2.73E-02	1.38E-02	62	3.24E-02	1.32E-02	60	58.40	3.95E-02	1.38E-02	60	58.40
48	torsion1	40000	197608	Duplicate Optimization	2.70E-02	6.88E-03	31	4.19E-02	5.60E-03	24	194.19	4.25E-02	5.30E-03	23	206.92
49	obstclae	40000	197608	Optimization	2.72E-02	6.80E-03	31	3.46E-02	5.45E-03	24	194.19	4.33E-02	5.33E-03	23	206.92
50	t2dah_e	11445	176117	DMR	2.70E-02	6.01E-03	32	3.56E-02	2.84E-03	15	98.79	4.33E-02	3.14E-03	15	127.74
51	nasa2910	2910	174296	Structural	5.79E-02	1.06E-01	390	8.01E-02	1.01E-01	378	7.63	1.19E-01	8.97E-02	331	24.55
52	Muu	7102	170134	Structural	2.93E-02	1.84E-03	10	3.90E-02	1.61E-03	9	9.69	4.87E-02	1.50E-03	8	16.54
53	bcsstk24	3562	159910	Structural	2.79E-02	1.51E-01	773	3.77E-02	8.93E-02	438	10.57	5.05E-02	7.10E-02	363	20.17
54	bcsstk18	11948	149090	Structural	2.63E-02	1.16E-01	547	3.54E-02	1.10E-01	522	20.52	4.48E-02	1.06E-01	489	34.02
55	ted_B	10605	144579	Thermal	2.79E-02	1.62E-03	9	3.22E-02	1.39E-03	8	12.70	4.13E-02	1.39E-03	8	14.54
56	ted_B_unscaled	10605	144579	Thermal	2.67E-02	1.53E-03	9	3.51E-02	1.39E-03	8	12.70	3.80E-02	1.36E-03	8	14.54
57	bodyy6	19366	134208	Structural	2.55E-02	1.35E-01	594	3.01E-02	1.42E-01	599	20.19	3.68E-02	1.32E-01	599	24.55
58	bodyy5	18589	128853	Structural	2.51E-02	6.06E-02	241	3.20E-02	5.98E-02	243	26.01	3.35E-02	6.10E-02	243	31.81
59	aft01	8205	125567	Acoustics	2.54E-02	8.13E-02	418	3.33E-02	6.73E-02	335	43.54	3.38E-02	6.09E-02	320	54.98
60	bodyy4	17546	121550	Structural	2.73E-02	2.35E-02	97	3.03E-02	2.38E-02	97	36.66	3.35E-02	2.34E-02	97	44.64
61	bcsstk15	3948	117816	Structural	2.65E-02	5.81E-02	240	3.48E-02	5.66E-02	225	30.67	4.44E-02	5.46E-02	220	41.91
62	crystm01	4875	105339	Materials	2.53E-02	3.97E-03	13	3.27E-02	3.84E-03	12	11.19	3.77E-02	3.50E-03	11	17.26
63	nasa4704	4704	104756	Structural	3.17E-02	3.06E-01	1410	3.13E-02	2.73E-01	1277	18.59	3.95E-02	2.66E-01	1217	32.10
64	msc04515	4515	97707	Structural	2.86E-02	1.03E-01	572	2.91E-02	7.97E-02	450	27.51	3.32E-02	7.87E-02	434	50.49
65	fv3	9801	87025	2D/3D	2.47E-02	2.46E-02	126	2.93E-02	2.33E-02	123	76.01	3.10E-02	2.51E-02	124	97.97
66	fv2	9801	87025	2D/3D	2.86E-02	2.83E-03	15	2.78E-02	2.63E-03	14	71.15	3.01E-02	2.66E-03	14	97.97
67	fv1	9604	85264	2D/3D	2.48E-02	2.82E-03	15	2.70E-02	2.47E-03	14	71.13	2.95E-02	2.54E-03	14	93.14
68	bcsstk13	2003	83883	CFD	3.79E-02	1.76E-01	566	5.68E-02	1.79E-01	535	21.94	8.39E-02	1.70E-01	496	41.15
69	sts4098	4098	72356	Structural	2.81E-02	1.81E-02	100	3.91E-02	1.66E-02	92	32.30	5.21E-02	1.57E-02	86	51.71
70	nasa2146	2146	72250	Structural	2.54E-02	2.12E-02	108	3.12E-02	2.02E-02	105	20.97	3.37E-02	2.05E-02	105	31.30
71	bcsstk14	1806	63454	Structural	2.59E-02	2.61E-02	116	3.06E-02	2.45E-02	109	10.60	3.84E-02	2.53E-02	105	16.71
72	bcsstk27	1224	56126	Structural	2.75E-02	1.84E-02	90	3.17E-02	1.89E-02	90	8.51	3.95E-02	1.79E-02	89	15.70

the MKL 2017.4 library in Skylake, LAPACK 3.8.0 in POWER9 and OPENBLAS 0.3.10 in A64FX. We consider our numerical experiments to converge when the initial residual norm is reduced by eight orders of magnitude, being the initial guess always zero. For every matrix, a random right-hand side is generated with values ranging from -1 to 1 and normalized to the matrix max norm. For the time measurements we have considered the minimum time over 20 and 50 repetitions of each experiment for the setup and solve phases, respectively.

Our evaluation considers the state-of-the-art FSAI approach, and two other preconditioners where the sparse pattern is extended using our cache-friendly methods. These are the precise descriptions of all considered approaches:

- FSAI - *Factorized Sparse Approximate Inverse preconditioner.* This is the state-of-the-art FSAI preconditioner [11] described in Algorithm 1. We use as pattern the lower triangular pattern of A without thresholding and filtering only null entries.
- FSAIE(sp) - *Factorized Sparse Approximate Inverse preconditioner with a sparse pattern extension exploiting spatial locality.* It extends the lower triangular sparse pattern of A, which is the pattern used by FSAI. It exploits spatial locality in the first product and temporal locality in the second. FSAIE(sp) corresponds to Algorithm 4 without steps 5 and 6. G coefficients are computed on the pattern $\mathcal{S}_{ext}$. We consider values 0.0, 0.001, 0.01, and 0.1 for the $filter$ parameter. The filtering-out process is applied using a scale-independent order of magnitude comparison of non-diagonal entries with respect to diagonal entries and removes only entries of the extension.
- FSAIE(full) -*Factorized Sparse Approximate Inverse preconditioner with pattern extension exploiting spatial and temporal locality.* It extends the lower triangular sparse pattern of A. It exploits spatial and temporal locality in both products. FSAIE(full) corresponds to Algorithm 4. We set the filtering-out parameter, $filter$, to values 0.0, 0.001, 0.01, and 0.1. The same value is used in the steps corresponding to lines 4 and 6. We apply the filtering-out process considering a scale-independent order of magnitude comparison of non-diagonal entries with respect to diagonal entries. The filtering-out processes remove only entries added in the extensions

We use a common configuration for all matrices to avoid a parameter fine-tuning for each experiment of the evaluation campaign. Convergence tolerance is kept constant in all experiments for all the approaches. We report results in terms of iteration and time reductions, Gflop/s, and data cache misses.

7.2 Performance Improvement on Skylake

In this section we describe the performance of FSAIE(sp) and FSAIE(full) with respect to FSAI. Table 1 shows results for the three techniques and $filter = 0.01$ in the Skylake system. Columns 6-8 report setup time, solve time, and iterations required to convergence for FSAI. These are the baseline results on Skylake against which we compare our pattern extension methods. Columns 9-12 report, respectively, setup time, solve time, iterations to convergence, and the percentage of entries FSAIE(sp) adds to the lower triangular pattern of A. Columns 13-16 report these same metrics concerning FSAIE(full). In many cases we observe both FSAIE(sp) and FSAIE(full) to successfully decrease iteration count and solve time. FSAIE(full) obtains larger pattern extensions than FSAIE(sp), which produces larger iteration count and solution time decreases.

Table 2: Percentage of average iteration and time improvements, highest time improvement, and highest time degradation of FSAIE(sp) and FSAIE(full) for $filter$ values 0.0, 0.001, 0.01, 0.1, and best $filter$ per matrix on Skylake.

FSAIE(sp)				
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.0	12.40	2.89	51.20	-68.94
0.001	12.25	5.99	51.05	-30.18
0.01	11.76	9.59	52.80	-22.82
0.1	6.32	5.54	43.64	-16.48
Best filter	**11.45**	**11.16**	**52.80**	**-3.34**
FSAIE(full)				
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.0	18.41	-3.69	56.72	-170.23
0.001	17.88	8.68	53.81	-41.99
0.01	16.71	12.75	53.09	-2.38
0.1	8.90	8.90	43.74	-19.84
Best filter	**16.60**	**15.02**	**56.72**	**-2.06**

Table 2 shows, per each $filter$ value we consider, the average improvement in terms of iteration count and solution time in the second and third columns, respectively. The *Best* $filter$ category shows results when the best $filter$ value per matrix is considered. Table 2 also shows the highest time improvement and degradation in its fourth and fifth columns, respectively. We report results considering the experimental setup of 72 matrices described in Section 7.1 and two cases: FSAIE(sp) and FSAIE(full). Note that for the $filter = 0.01$ the results of Table 2 are a summary of Table 1. We observe for the two methods how low filter values, e.g. $filter = 0.0$, lead to high iterations decrease, which is not translated into time-to-solution decrease. In these scenarios, the cost per iteration is high since extended patterns have a large amount of entries with low values that do not improve the preconditioner quality but have a significant computational cost. When increasing the $filter$ value, the gap between average iteration decrease and average time decrease is reduced as the iteration overhead reduces as well. For large values this gap becomes very small, although the improvement in terms of iteration count is relatively small since we add a small number of entries to the sparse pattern. The best improvements are obtained when $filter = 0.01$. The achieved accuracy for all matrices is always lower than 1E-08, which is the value set for all tests, and does not significantly change when performing pattern extensions. Following experiments show the same behavior.

Table 2 shows how the FSAIE(full) method achieves much better results than FSAIE(sp). FSAIE(full) obtains larger average performance improvements for $filter$ values 0.001, 0.01, and 0.1. Extending the sparse pattern by considering spatial and temporal locality in the two SpMV operations, $G^T Gp$, produces larger and

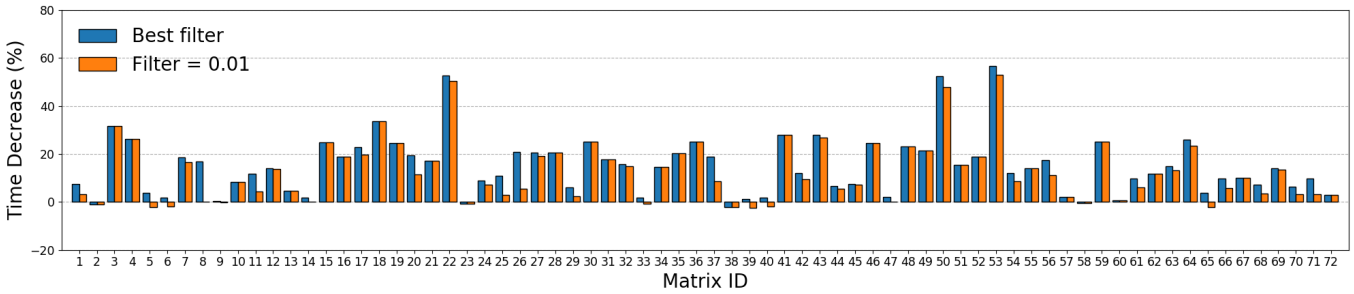


Figure 2: Time decrease of the FSAIE(full) vs FSAI using the best $filter$ value per matrix (blue columns) and $filter = 0.01$ value (orange columns) on the Skylake architecture.

more efficient sparse patterns. These improved sparse pattern extensions obtain larger reductions in terms of iteration count while succeeding in keeping the iteration cost low. FSAIE(full) obtains average performance improvements of 12.75% and 8.90% when parameter $filter$ is set to 0.01 and 0.1, respectively. FSAIE(full) also obtains time reductions of more than 50% for some matrices with $filter = 0.01$. In the following sections, all experiments consider FSAIE(full) as described in Algorithm 4.

Table 3: Average iteration increase percentage and highest iteration increase percentage of FSAIE(sp), when using standard filtering-out against the proposed filtering-out strategy. The $filter$ values 0.0, 0.001, 0.01, 0.1 are considered.

Filter value	Avg. iter. inc.	Highest iter. inc.
0.0	0.0	0.88
0.001	0.0	1.95
0.01	1.63	113.9
0.1	7.95	114.96

It is not possible to find a single $filter$ value optimal for all the matrices belonging to the extensive and heterogeneous set we consider in this paper. Previous work [5, 6, 15] tunes parameters for each particular matrix when applying FSAI. By considering the best $filter$ value for each matrix we obtain a 15.02% average time improvement, being the best result among all matrices a time decrease of 56.72% and the worst a time increase of 2.06%.

Figure 2 depicts the improvement obtained in Skylake by the FSAIE(full) method with respect to FSAI for each matrix of the experimental set. In the x-axis we show Matrix IDs, which are defined in the first column of Table 1. In the y-axis we display the performance improvement in terms of time decrease percentage. We show the results of the best performing $filter$ value for each matrix (blue columns) and results for $filter = 0.01$ (orange columns). The two approaches displayed in Figure 2 show similar trends since for most of the matrices $filter = 0.01$ is the best option.

Table 3[1] compares the state-of-the-art filtration strategy of Algorithm 1 with the new strategy proposed in Section 5. Results are presented in terms of iterations increase when the standard approach is used instead of the new one, for various $filter$ values. Results are presented in terms of iterations since, for each $filter$ value, the final number of entries is the same in both approaches so the computational cost is equivalent. For low $filter$ values both methods lead to similar iteration results. However, for larger $filter$ values, i.e. 0.1, the standard approach causes a significant degradation of the convergence for some matrices of the experimental set. On average, with $filter = 0.1$, matrices converge with 7.95% more iterations with respect to the proposed filtering-out strategy. In all cases, our new filtration strategy avoids convergence degradation while providing a higher degree of robustness to the method.

[1]This Table does not consider one case in the experimental set that did not converge when using the state-of-the-art filtering strategy with $filter = 0.1$.

7.3 Effects on Data Cache Misses and FLOPS

In this section we describe the benefits of the FSAIE(full) method in terms of data cache misses and floating-point operations per second (flop/s) when its extended sparse patterns are used on Skylake. FSAIE(full) bases its effectiveness on two aspects: First, achieving a reduction in terms of iteration count by extending the preconditioner sparse pattern; and, second, keeping low the additional iteration cost the pattern extension incurs. Section 7.2 has shown the benefits of FSAIE(full) in terms of time to solution and iteration count with respect to FSAI in the Skylake system. This section demonstrates the effectiveness of FSAIE(full) in this second aspect.

Figure 3 shows three histograms displaying the average L1 data cache misses triggered by accesses to vector p when computing the preconditioning operation G^TGp. The cache misses are normalized to the total number of non-zero entries of the G matrix. The histograms classify each one of the 72 matrices of our experimental setup depending on the number of L1 data cache misses experienced when preconditioning p via G^TGp. We consider three different ways of obtaining the G matrix: First, via the state-of-the-art FSAI preconditioner (category G_{FSAI}); second, via the FSAIE(full) method (category $G_{FSAIE(full)}$); and, third, using a randomly generated extended sparse pattern (category G_{random}). When considering $G_{FSAIE(full)}$ and G_{random}, the amount of new entries present in the extended sparse pattern is the same. These results clearly indicate how our cache-friendly sparse pattern extensions successfully minimize the average data cache misses per G non-zero entry. This cache behaviour makes it possible to translate the iteration

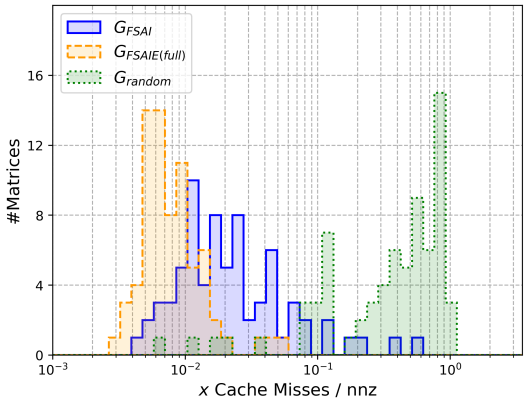


Figure 3: Histogram of L1 data cache misses on p accesses in the preconditioning operation G^TGp normalized to the number of G matrix non-zero entries. Blue columns correspond to state-of-the-art G matrices, orange columns to cache-friendly extended G matrices, and green columns to randomly extended matrices.

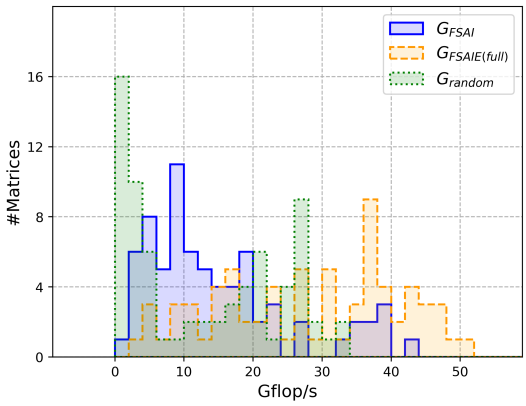


Figure 4: Histogram of the Gflop/s ratio when preconditioning the p vector via the G^TGp operation. Blue columns correspond to state-of-the-art G matrices, orange columns to cache-friendly extended G matrices, and green columns to randomly extended matrices.

count reductions of FSAIE(full) to elapsed time reductions. Random extensions dramatically increase L1 data cache misses, which highlights the quality of the $G_{FSAI(full)}$ sparse pattern.

Figure 4 contains three histograms showing the observed Gflop/s ratios when preconditioning the p vector via the G^TGp operation. These histograms classify each one of the 72 considered matrices depending on the Gflop/s ratios they reach when preconditioning p. We consider the same three different approaches to construct the G matrix as Figure 3: G_{FSAI}, $G_{FSAIE(full)}$, and G_{random}. Figure 4 clearly shows how the cache-aware extensions of the FSAIE(full) method significantly improve the floating-point throughput achieved by the sparse patterns of baseline FSAI. For a significant number of matrices, FSAIE(full) produces sparse patterns able to reach more than 40Gflops/s when performing the two SpMV operations of G^TGp. Since the peak performance of our double socket 24-core Skylake system is 3200GFlop/s, FSAIE(full) makes it possible to achieve more than 1.25% of the peak for a large number of matrices. This is a very large percentage, since the SpMV kernel rarely reaches more than 40Gflop/s on multi-core x86 architectures with 512-bit SIMD extensions [1, 36]. Randomly extending the sparse pattern of G does not produce competitive flop/s ratios in general.

7.4 Setup Phase Overhead

While our methods significantly accelerate the solver phase of the conjugate gradient method, the extension of the sparse pattern incurs some overhead in the setup phase with respect to FSAI, mainly due to the higher cost of computing G entries. This overhead is on average 180% for FSAIE(full) with $filter = 0.01$ with respect to FSAI. Table 1 shows the timing cost per matrix of the setup and solve phases for FSAI in its sixth and seventh columns, respectively. The setup and solve costs of FSAIE(sp) are represented in the ninth and tenth columns. These two metrics for the FSAIE(full) appear in the thirteenth and fourteenth columns of Table 1.

Such overhead becomes negligible in a practical numerical simulation context since the setup phase is performed only once while the solve phase is repeated several times for the same matrix. Furthermore, even when the setup is to be repeated on each time step, some of its parts, such as the definition of the final pattern, do not need to be repeated on each time step.

Table 4: Percentage of average iteration and time improvements, highest time improvement, and highest time degradation of FSAIE(sp) and FSAIE(full) for $filter$ values 0.0, 0.001, 0.01, 0.1, and best $filter$ per matrix on POWER9.

	FSAIE(full)			
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.0	18.55	-14.24	52.26	-208.93
0.001	17.96	2.49	54.49	-58.08
0.01	16.90	10.25	56.72	-18.90
0.1	8.99	8.56	42.75	-12.35
Best filter	**15.15**	**12.94**	**56.72**	**-12.35**

7.5 Evaluation on POWER9

We perform numerical experiments on a 40-cores POWER9 system. Our parallel experiments run on all the 40 cores. The experimental setup is described in detail in Section 7.1. To highlight the architecture-independent aspect of our proposals, we use the exact same source code for both POWER9 and Skylake experiments, with the exception of the numerical library used to solve the linear systems associated with each row on the computation of G_{ext}. Table 4 shows a summary of the results obtained considering different $filter$ values. The average improvement for the FSAIE(full) algorithm when using the best $filter$ value per matrix is 12.94%.

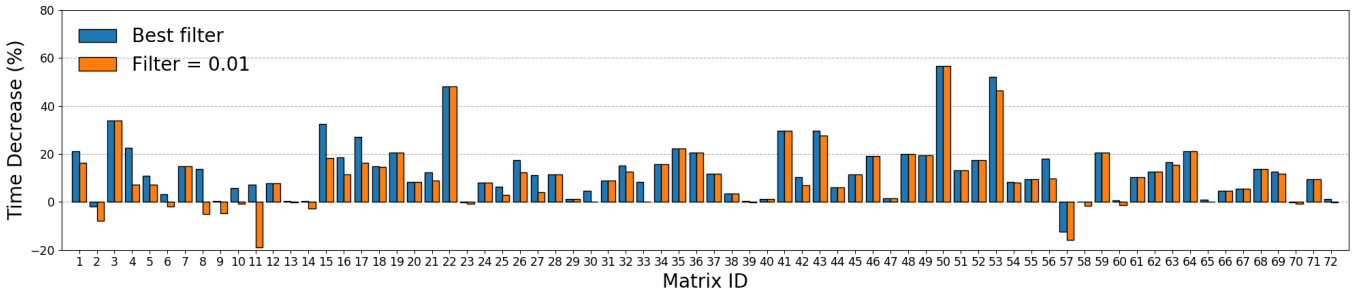


Figure 5: Time decrease of the FSAIE(full) vs FSAI for the best $filter$ value (blue columns) and for the 0.01 $filter$ value (orange columns) on the POWER9 architecture.

Figure 5 shows the improvement obtained on the POWER9 system by the FSAIE(full) method with respect to FSAI for each matrix of the experimental set. In the x-axis we show the Matrix IDs, which are defined in Table 1. In the y-axis we display the performance improvement in terms of time decrease percentage. We show results considering the best performing $filter$ value (blue columns) per matrix, and the best common $filter$ value (orange columns), which is 0.01. The trends we see in Figure 5 are similar to those in Figure 2. For most of matrices $filter = 0.01$ is the best option.

Since both Skylake and POWER9 machines have 64Bytes cache lines, the flexibility of FSAIE(full) for adding additional entries when generating new sparse patterns is the same on the two systems. Therefore, the average iteration decrease achieved by FSAIE(full) is very similar on both systems, as we can see in Tables 2 and 4. The small differences are due to different cache line alignments of vector p, plus small numerical effects due to round-off errors.

Table 5: Percentage of average iteration and time improvements, highest time improvement, and highest time degradation of FSAIE(sp) and FSAIE(full) for $filter$ values 0.0, 0.001, 0.01, 0.1, and best $filter$ per matrix on A64FX.

	FSAIE(full)			
Filter value	**Avg. iterations**	**Avg. time**	**Highest imp.**	**Highest deg.**
0.0	27.81	-17.52	76.99	-575.19
0.001	26.47	14.93	63.47	-54.79
0.01	23.98	20.08	61.38	-6.58
0.1	13.36	13.76	48.03	-3.80
Best filter	**24.91**	**22.85**	**76.99**	**-0.96**

7.6 Evaluation on A64FX

We perform numerical experiments on a 48-cores A64FX system. The experimental setup is described in detail in Section 7.1. We use the same code as the POWER9 and Skylake experiments, with the exception of the numerical library used for the computation of the inverse approximation. A64FX features 256Bytes cache lines, four times larger than Skylake and POWER9 cache lines. This is an important difference as it enables FSAIE(full) to add more cache-friendly entries to the extended sparse patterns, which produces larger iteration count decreases. Indeed, Table 5 clearly shows much larger average iteration decreases than Tables 2 and 4. These large iteration decreases bring significant performance improvement. The FSAIE(full) method brings average performance improvements of 20.08% for $filter = 0.01$, and 22.85% when using the best $filter$ per matrix. Not filtering out any entry, i. e. $filter = 0.0$, brings performance degradation since the iteration cost grows more than the iteration reduction.

Figure 6 shows the improvement obtained on A64FX by the FSAIE(full) method with respect to FSAI for each matrix of the experimental set. In the x-axis we show the Matrix IDs, which are defined in Table 1. In the y-axis we display the performance improvement in terms of time decrease percentage. We show the results of the best performing $filter$ value (blue columns) for each matrix and the results for the best general $filter$ value (orange columns), which is 0.01. Many matrices display much larger performance improvements on A64FX than POWER9 and Skylake.

7.7 Comparing Results on Different Architectures

To clearly illustrate the benefits of FSAIE(full) on Skylake, POWER9 and A64FX, we show in Figure 7 three histograms classifying the 72 matrices of our experimental set in terms of time improvement with respect to FSAI. The red dotted vertical line represents the median improvement. There is a large difference between A64FX and the other two architectures. While results on Skylake and POWER9 show similar trends due to their fundamentally equal pattern extensions, when FSAIE(full) is applied to A64FX it obtains richer sparse patterns, which significantly increase the average improvement for all the matrices in the experimental set. This is a consequence of A64FX having a 4 times larger cache line size than Skylake or POWER9. On average, FSAIE(full) with $filter = 0.01$ generates 61% additional entries to G on Skylake and POWER9, and 93% additional entries on A64FX.

8 RELATED WORK ON APPROXIMATE INVERSE METHODS

There are several previously proposed methods to generate patterns for the sparse approximate inverse problem. They are considered either static or dynamic methods [14], depending on how the sparse pattern of the approximate inverse is evaluated.

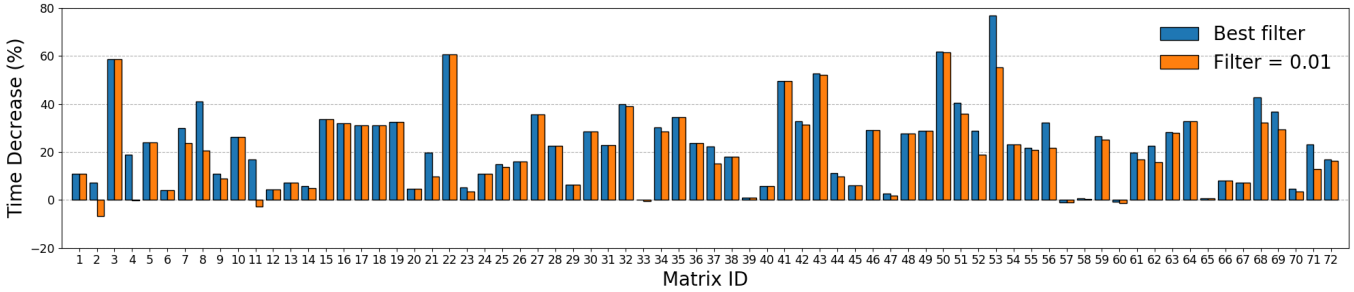
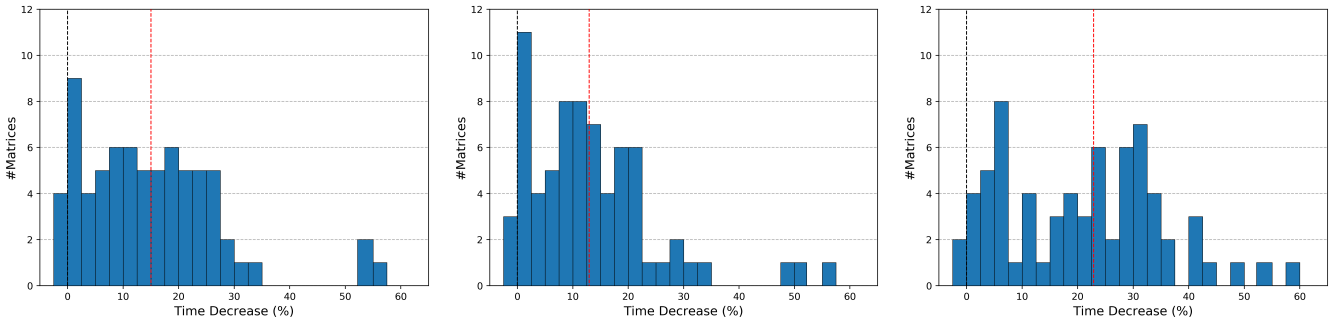


Figure 6: Time decrease of the FSAIE(full) vs FSAI for the best $filter$ value (blue columns) and for the 0.01 $filter$ value (orange columns) on the A64FX architecture.



(a) Skylake **(b) POWER9** **(c) A64FX**

Figure 7: Histograms classifying the 72 matrices of our experimental set in terms of time improvement achieved by FSAIER(full) using the best $filter$ value with respect to FSAI.

For the static approach, the one considered in this paper, the pattern is determined a priori and kept unvaried throughout the computation of the inverse approximation, either M or G for factorized approaches. Various alternatives have been considered. A common one is using the sparse pattern of a power of the A matrix, usually A^2 or A^3 [10, 16, 20]. Other techniques reshape the initial pattern [23]. The resulting patterns can be sparsified through the processes of thresholding and post-filtration or through adaptive entry dropping strategies [5, 6, 11, 15, 27, 29]. Finding optimal thresholding and filtration criteria is generally a challenging task.

Dynamic approximate inverse methods compute the inverse pattern from adaptive procedures that start from an initial guess, for example a diagonal pattern, and enlarge it following some strategy until a specific criteria is fulfilled. An example of a dynamic approach, SPAI, was proposed by Grote and Huckle [17]. There is also a factorized formulation, FSPAI [21]. Other dynamic strategies have been developed more recently, such as, its generalization to block form, BSAI [22], and others like PSAI and RSAI [25, 26]. Typically, dynamic approximate inverses are more powerful than their static counterparts. However, it is not trivial to efficiently parallelize them, and their preprocessing stage is generally much costlier than static approaches. There are implementations of several dynamic and static strategies for computing FSAI preconditioners in shared-memory parallel machines [24]. In addition, techniques to run SAI preconditioners on GPUs have been proposed for both static [3, 7, 31, 35] and dynamic strategies [8].

A common factor of all cited approximate inverse methods, either static or dynamic, is that none of them take into consideration architectural criteria to define the sparse pattern. Based on the concept of cache-friendly pattern extensions, our method is complementary to any of the alternatives mentioned. The most common static method, FSAI, has been used here as a reference. Nonetheless, given any other pattern evaluated with numerical criteria, our approach brings out a potentially significant performance boost.

9 CONCLUSIONS

This paper demonstrates the benefits of a FSAI sparse pattern extension based on two fundamental concepts: first, an algorithm able to produce a cache-aware extension of the sparse pattern in a way that the iteration count of CG is significantly reduced with a low time per iteration overhead; second, a robust filtering strategy that maximizes the benefits of the cache-aware extension. Our extensive evaluation campaign considers 72 matrices and cutting-edge high-end hardware. It demonstrates on the Skylake architecture average improvements of 15.02% in terms time to solution, and time reductions of more than 50% for some matrices. Our evaluation shows that our proposals are applicable to any generic multi-core architecture by reporting performance improvements on Skylake,

POWER9, and A64FX. The large 256Bytes cache lines of A64FX produce the best results, namely, average improvements of 22.85% in terms time to solution, and time reductions of more than 75% for some matrices.

While state-of-the-art approaches define sparse patterns exclusively based on numerical considerations, this paper is the first in demonstrating the benefits of taking into account computer architecture concepts. Indeed, the cache-aware pattern extension proposed is complementary to any numerical strategy employed for the definition of the sparsity pattern. Aspects like the design of the cache hierarchy or the information contained in virtual addresses, enable those additional optimizations that are easily applicable to boost applications performance.

ACKNOWLEDGMENTS

This work is partially supported by the Spanish Ministry of Science and Technology through PID2019-107255GB project and by the Generalitat de Catalunya (contract 2017-SGR-1414). Marc Casas has been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship number RYC-2017-23269. It has also been partially supported by the EXCELLERAT project funded by the European Commission's ICT activity of the H2020 Programme under grant agreement number: 823691.

REFERENCES

[1] Christie L. Alappat, Johannes Hofmann, Georg Hager, Holger Fehske, Alan R. Bishop, and Gerhard Wellein. 2020. Understanding HPC Benchmark Performance on Intel Broadwell and Cascade Lake Processors. In *High Performance Computing*, Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief (Eds.). Springer International Publishing, Cham, 412–433.

[2] Guillaume Alléon, Michele Benzi, and Luc Giraud. 1997. Sparse approximate inverse preconditioning for dense linear systems arising in computational electromagnetics. *Numerical Algorithms* 16 (02 1997), 1–15. https://doi.org/10.1023/A:1019170609950

[3] Hartwig Anzt, Edmond Chow, Thomas Huckle, and Jack Dongarra. 2016. Batched Generation of Incomplete Sparse Approximate Inverses on GPUs. 49–56. https://doi.org/10.1109/ScalA.2016.011

[4] Michele Benzi, Carl D. Meyer, and Miroslav Tůma. 1996. A Sparse Approximate Inverse Preconditioner for the Conjugate Gradient Method. *SIAM Journal on Scientific Computing* 17, 5 (1996), 1135–1149. https://doi.org/10.1137/S1064827594271421 arXiv:https://doi.org/10.1137/S1064827594271421

[5] Luca Bergamaschi, Giuseppe Gambolati, and Giorgio Pini. 2007. A numerical experimental study of inverse preconditioning for the parallel iterative solution to 3D finite element flow equations. *J. Comput. Appl. Math.* 210 (12 2007), 64–70. https://doi.org/10.1016/j.cam.2006.10.056

[6] Luca Bergamaschi, Ángeles Martínez, and Giorgio Pini. 2006. Parallel preconditioned conjugate gradient optimization of the Rayleigh quotient for the solution of sparse eigenproblems. *Appl. Math. Comput.* 175, 2 (2006), 1694 – 1715. https://doi.org/10.1016/j.amc.2005.09.015

[7] Massimo Bernaschi, Mauro Bisson, Carlo Fantozzi, and Carlo Janna. 2016. A Factored Sparse Approximate Inverse Preconditioned Conjugate Gradient Solver on Graphics Processing Units. *SIAM Journal on Scientific Computing* 38, 1 (2016), C53–C72. https://doi.org/10.1137/15M1027826 arXiv:https://doi.org/10.1137/15M1027826

[8] Massimo Bernaschi, Mauro Carrozzo, Andrea Franceschini, and Carlo Janna. 2019. A Dynamic Pattern Factored Sparse Approximate Inverse Preconditioner on Graphics Processing Units. *SIAM Journal on Scientific Computing* 41, 3 (2019), C139–C160. https://doi.org/10.1137/18M1197461 arXiv:https://doi.org/10.1137/18M1197461

[9] Daniele Bertaccini and Salvatore Filippone. 2016. Sparse approximate inverse preconditioners on high performance GPU platforms. *Computers & Mathematics with Applications* 71, 3 (2016), 693 – 711. https://doi.org/10.1016/j.camwa.2015.12.008

[10] Edmond Chow. 2000. A Priori Sparsity Patterns for Parallel Sparse Approximate Inverse Preconditioners. *SIAM Journal on Scientific Computing* 21, 5 (2000), 1804–1822. https://doi.org/10.1137/S106482759833913X arXiv:https://doi.org/10.1137/S106482759833913X

[11] Edmond Chow. 2001. Parallel Implementation and Practical Use of Sparse Approximate Inverse Preconditioners With a Priori Sparsity Patterns. *International Journal of High Performance Computing Applications* 15 (05 2001). https://doi.org/10.1177/109434200101500106

[12] Edmond Chow and Yousef Saad. 1998. Approximate Inverse Preconditioners via Sparse-Sparse Iterations. *SIAM Journal on Scientific Computing* 19, 3 (1998), 995–1023.

[13] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[14] Massimiliano Ferronato. 2012. Preconditioning for Sparse Linear Systems at the Dawn of the 21st Century: History, Current Developments, and Future Perspectives. *ISRN Applied Mathematics* 2012 (12 2012). https://doi.org/10.5402/2012/127647

[15] Massimiliano Ferronato, Carlo Janna, and Giorgio Pini. 2012. Shifted FSAI preconditioners for the efficient parallel solution of non-linear groundwater flow models. *Internat. J. Numer. Methods Engrg.* 89 (03 2012), 1707–1719. https://doi.org/10.1002/nme.3309

[16] John R. Gilbert. 1994. Predicting Structure in Sparse Matrix Computations. *SIAM J. Matrix Anal. Appl.* 15, 1 (1994), 62–79. https://doi.org/10.1137/S0895479887139455 arXiv:https://doi.org/10.1137/S0895479887139455

[17] Marcus J. Grote and Thomas Huckle. 1997. Parallel Preconditioning with Sparse Approximate Inverses. *SIAM Journal on Scientific Computing* 18, 3 (1997), 838–853. https://doi.org/10.1137/S1064827594276552

[18] Wolfgang Hackbusch. 1985. *Multi-Grid Methods and Applications*. Vol. 4. https://doi.org/10.1007/978-3-662-02427-0

[19] Guixia He, Renjie Yin, and Jiaquan Gao. 2019. An efficient sparse approximate inverse preconditioning algorithm on GPU. *Concurrency and Computation: Practice and Experience* 32 (12 2019). https://doi.org/10.1002/cpe.5598

[20] Thomas Huckle. 1999. Approximate sparsity patterns for the inverse of a matrix and preconditioning. *Applied Numerical Mathematics* 30, 2 (1999), 291 – 303. https://doi.org/10.1016/S0168-9274(98)00117-2

[21] Thomas Huckle. 2003. Factorized Sparse Approximate Inverses for Preconditioning. *Journal of Supercomputing* 25, 2 (2003), 109–117.

[22] Carlo Janna and Massimiliano Ferronato. 2011. Adaptive Pattern Research for Block FSAI Preconditioning. *SIAM Journal on Scientific Computing* 33, 6 (2011), 3357–3380. https://doi.org/10.1137/100810368 arXiv:https://doi.org/10.1137/100810368

[23] Carlo Janna, Massimilano Ferronato, and Giuseppe Gambolati. 2010. A Block FSAI-ILU Parallel Preconditioner for Symmetric Positive Definite Linear Systems. *SIAM Journal on Scientific Computing* 32, 5 (2010), 2468–2484. https://doi.org/10.1137/090779760 arXiv:https://doi.org/10.1137/090779760

[24] Carlo Janna, Massimiliano Ferronato, Flavio Sartoretto, and Giuseppe Gambolati. 2015. FSAIPACK: A Software Package for High-Performance Factored Sparse Approximate Inverse Preconditioning. *ACM Trans. Math. Softw.* 41, 2, Article 10 (Feb. 2015), 26 pages. https://doi.org/10.1145/2629475

[25] Zhongxiao Jia and Wenjie Kang. 2017. A residual based sparse approximate inverse preconditioning procedure for large sparse linear systems. *Numerical Linear Algebra with Applications* 24, 2 (2017), e2080. https://doi.org/10.1002/nla.2080 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/nla.2080 e2080 nla.2080.

[26] Zhongxiao Jia and Baochen Zhu. 2009. A Power Sparse Approximate Inverse Preconditioning Procedure for Large Sparse Linear Systems. *Numerical Linear Algebra with Applications* 16 (04 2009), 259 – 299. https://doi.org/10.1002/nla.614

[27] Liliya Yu. Kolotilina, Andy A. Nikishin, and Alex Yu. Yeremin. 1999. Factorized Sparse Approximate Inverse Preconditionings. IV: Simple Approaches to Rising Efficiency. *Numerical Linear Algebra With Applications - NUMER LINEAR ALGEBR APPL* 6 (10 1999), 515–531. https://doi.org/10.1002/(SICI)1099-1506(199910/11)6:73.0.CO;2-0

[28] Liliya Yu. Kolotilina and Alex Yu. Yeremin. 1993. Factorized Sparse Approximate Inverse Preconditionings I. Theory. *SIAM J. Matrix Anal. Appl.* 14, 1 (1993), 45–58. https://doi.org/10.1137/0614004 arXiv:https://doi.org/10.1137/0614004

[29] Jiří Kopal, Miroslav Rozložník, and Miroslav Tůma. 2015. Approximate inverse preconditioners with adaptive dropping. *Advances in Engineering Software* 84 (2015), 13 – 20. https://doi.org/10.1016/j.advengsoft.2015.01.006 CIVIL-COMP.

[30] Mark [D. Kremenetsky], John Richardson, and Horst [D. Simon]. 1995. - Parallel preconditioning for CFD problems on the CM-5. In *Parallel Computational Fluid Dynamics 1993*, A. Ecer, J. Hauser, P. Leca, and J. Periaux (Eds.). North-Holland, Amsterdam, 401 – 410. https://doi.org/10.1016/B978-044481999-4/50173-0

[31] I.B. Labutin and I.V. Surodina. 2013. Algorithm for sparse approximate inverse preconditioners in the conjugate gradient method. 19 (01 2013), 120–126.

[32] G. Oyarzun, R. Borrell, A. Gorobets, and A. Oliva. 2014. MPI-CUDA sparse matrix–vector multiplication for the conjugate gradient method with an approximate inverse preconditioner. *Computers & Fluids* 92 (2014), 244 – 252. https://doi.org/10.1016/j.compfluid.2013.10.035

[33] Yousef Saad. 2002. Preconditioned Krylov Subspace Methods for CFD Applications. *Proceedings of the international workshop on solution techniques for large-scale CFD problems* (02 2002).

[34] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2nd ed.). Society for Industrial and Applied Mathematics, USA.

[35] K. Xu, D.Z. Ding, Z.H. Fan, and R.S. Chen. 2011. FSAI preconditioned CG algorithm combined with GPU technique for the finite element analysis of electromagnetic scattering problems. *Finite Elements in Analysis and Design* 47, 4 (2011), 387 – 393. https://doi.org/10.1016/j.finel.2010.11.005

[36] Hong Zhang, Richard T. Mills, Karl Rupp, and Barry F. Smith. 2018. Vectorized Parallel Sparse Matrix-Vector Multiplication in PETSc Using AVX-512 *(ICPP 2018)*. Association for Computing Machinery, New York, NY, USA, Article 55, 10 pages. https://doi.org/10.1145/3225058.3225100