Mixed Precision Incomplete and Factorized Sparse Approximate Inverse Preconditioning on GPUs

Fritz Göbel[1(✉)], Thomas Grützmacher[1], Tobias Ribizel[1], and Hartwig Anzt[1,2]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
fritz.goebel@kit.edu
[2] University of Tennessee, Knoxville, USA

Abstract. In this work, we present highly efficient mixed precision GPU-implementations of an Incomplete Sparse Approximate Inverse (ISAI) preconditioner for general non-symmetric matrices and a Factorized Sparse Approximate Inverse (FPSAI) preconditioner for symmetric positive definite matrices. While working with full double precision in all arithmetic operations, we demonstrate the benefit of decoupling the memory precision and storing the preconditioner in a more compact low precision floating point format to reduce the memory access volume and therefore preconditioner application time.

Keywords: Mixed precision · Preconditioning · Incomplete Sparse Approximate Inverse · Factorized Sparse Approximate Inverse · GPUs

1 Introduction

The solution of large sparse linear systems $Ax = b$ is a ubiquitous problem in scientific computing applications, with iterative solvers like Krylov subspace methods being often the tool of choice. In practice, these iterative solvers are usually combined with preconditioners to improve their convergence behavior. In particular for ill-conditioned matrices A, transforming the linear system with an appropriate preconditioner M into an equivalent, (left-)preconditioned system $MAx = Mb$ can lead to great convergence and runtime improvements.

An efficient preconditioner is an approximation $M \approx A^{-1}$ such that MA has a lower condition number than A and the Krylov method needs fewer iterations to converge. For this approach to be efficient in the execution-time metric, the overhead of generating and applying the preconditioner has to be smaller than the runtime improvement from accelerated convergence. Most Krylov methods allow for a high degree of parallelization, so considering the increasingly large amount of parallel resources available on modern manycore CPUs and GPUs, efficient preconditioners do not only need to provide good approximation properties, but also take advantage of massively parallel hardware.

The (left) Incomplete Sparse Approximate Inverse (ISAI) preconditioner [7] computes a sparse approximation M on the inverse of A by ensuring the ISAI-property $(MA-I)_{\mathcal{S}} = 0$ holds on a given sparsity pattern $\mathcal{S}$, which is commonly chosen based on the sparsity pattern of A. The ISAI preconditioner offers a lot of parallelism in the generation phase, as every row of the preconditioner can be computed independently, and its application is a simple sparse matrix-vector product (SpMV).

In scientific simulations, very often one wants to solve symmetric positive definite (spd) linear systems. Among the most efficient solvers for spd problems is the Conjugate Gradient (CG) method. Unfortunately, the ISAI preconditioner, as well as other sparse approximate inverse preconditioners generally do not preserve the symmetry of the input matrix [7, 11]. However, a slight modification turns the ISAI preconditioner into the Factorized Sparse Approximate Inverse preconditioner (commonly referred to as FSPAI) as presented in [12]. Being an approximation to the exact Cholesky factorization, the FSPAI preconditioner preserves the spd-ness of the system matrix in the preconditioner, and therewith can be used in conjunction with the CG method to solve the preconditioned system. The downside is that applying the FSPAI preconditioner requires two SpMV operations per iteration instead of one when using ISAI preconditioning.

On modern hardware architectures, the performance of sparse linear algebra kernels is typically limited by memory bandwidth, less so by compute power. That is, the memory access volume in combination with the main memory bandwidth determines the execution time of an algorithm. To address this bottleneck, Anzt et al. [6] proposed to decouple the memory precision format from the arithmetic precision format via a *memory accessor*, practically compressing all data in memory before and after performing arithmetic operations in the processor registers. They propose to perform all arithmetic operations in the natively-supported IEEE 754 double precision format, but to use more compact and imprecise formats to store and read the numerical values. As long as the object can tolerate some compression-induced rounding errors, using this strategy can significantly reduce the runtime of memory-bound algorithms. In this work, we adopt the strategy of decoupling the memory precision from the arithmetic precision and a memory accessor implementation [3] available in Ginkgo [2] for ISAI and FSPAI preconditioning. We always compute and apply the preconditioners in double precision, but utilize the memory accessor to use a lower precision format for the memory operations.

For introducing the idea of mixed precision ISAI and FSPAI preconditioning, presenting the respective software realizations, and demonstrating the practical benefits in a performance analysis on high-end GPUs, the rest of this paper is structured as follows. In Sect. 2 we revisit the ISAI algorithm as presented in [7]. Section 3 describes the extension of this approach to FSPAI preconditioners for spd problems as introduced in [12]. We present an efficient GPU-implementation of the preconditioner generation in Sect. 4.1 before explaining how we make use of mixed precision in the preconditioner application with a memory accessor in Sect. 4.2. In Sect. 5, we demonstrate the practical usability and performance

advantages in an experimental evaluation using various test problems from the SuiteSparse Matrix Collection [1]. We summarize our results and give an outlook on future research in Sect. 6.

2 Incomplete Sparse Approximate Inverses for General Matrices

A popular technique for preconditioning sparse linear systems is based on incomplete factorizations: We can approximate the factorization $A \approx L \cdot U$ with sparse triangular factors L and U with only a subset of the fill-in that is usually introduced by a full LU factorization. This incomplete factorization is required to be exact only on a certain sparsity pattern $\mathcal{S}$ (e.g. the sparsity pattern of A^k for ILU(k)), meaning

$$(LU)_{ij} = A_{ij} \quad \forall (i,j) \in \mathcal{S}. \tag{1}$$

A bottleneck in using incomplete factorization preconditioners on parallel architectures is that the generation of the triangular factors is an inherently sequential process, with a long critical path and parallelism only existing in the form of smaller sets of unknowns that can be computed independently (level scheduling [14]). Applying an incomplete factorization preconditioner involves solving two triangular systems via forward and backward substitution, which again exposes very limited parallelism. The *Incomplete Sparse Approximate Inverse* (ISAI [7]) algorithm was initially suggested to accelerate the application of incomplete factorization preconditioners. It replaces the forward and backward triangular solves by a matrix-vector multiplication with the approximate inverses of L and U. The approximation is called *incomplete* since we again require the inverse to be exact only on a limited sparsity pattern S. While replacing forward and backward substitution with the ISAI application greatly accelerate the application phase of incomplete sparse approximate inverse preconditioners, the bottleneck of computing the incomplete factorizations in the first place remains. However, the ISAI can also be computed for general matrices. This allows generating the ISAI preconditioner for the system matrix itself, therewith skipping the factorization generation and further reducing the preconditioner application cost to one SpMV per iteration.

Building on the incompleteness property (1), for a given sparsity pattern $\mathcal{S}$, we define the ISAI property as

$$(MA - I)_{\mathcal{S}} = 0 :\Leftrightarrow (MA - I)_{i,j} = 0 \quad \forall (i,j) \in \mathcal{S}. \tag{2}$$

In the following, we will assume that the preconditioner sparsity pattern is given by a power of A, i.e. that $\mathcal{S}(M) = \mathcal{S}(A^k)$ for some $k \in \mathbb{N}$. With $\mathcal{I} := \{j \mid (i,j) \in \mathcal{S}(M)\}$ being the sparsity pattern of the i-th row of M, the ISAI property (2) can be rewritten as

$$\sum_{k \in \mathcal{I}} m_{ik} a_{kj} = \delta_{ij} \quad \text{for all } j \in \mathcal{I}. \tag{3}$$

This equation is equivalent to the linear system $M(i,\mathcal{I})A(\mathcal{I},\mathcal{I}) = I(i,\mathcal{I})$ where I is the $n \times n$ identity matrix. To compute the preconditioner M, we transpose the requirement and solve n (typically small) independent linear systems $\tilde{A}x_i = b$ with $\tilde{A}_i = A^T(\mathcal{I},\mathcal{I})$ and $b = I(\mathcal{I}, i)$, $i = 1, ..., n$. For convenience, we provide the algorithmic description for computing the ISAI preconditioner for a general matrix A by solving n independent linear systems via the Gauss-Jordan-Elimination in Listing 1.1.

Listing 1.1. Algorithm computing the (left) ISAI preconditioner for an $n \times n$ square matrix A on a given sparsity pattern $\mathcal{S}$.

```
% Parallel For-Loop
for i=1:n
    % Extract nonzero indices of i-th row of A.
    I = nonzero_indices(A(i,:));
    small_dense = transpose(A)(I, I);
    % The right hand side is composed of the relevant
    % entries of the i-th unit vector.
    b = identity(n);
    b = b(I, i);
    % Solve small systems with Gauss-Jordan-Elimination.
    x = small_dense \ b;
    M(i,I) = transpose(x);
end
```

Note that for the preconditioner to be well-defined, all $\tilde{A}_i$ need to be non-singular. This is a strong requirement that is not fulfilled in general for non-symmetric matrices A. At the same time, our experimental evaluation in Sect. 5 reveals that in practice, this strategy succeeds for many systems, providing significant convergence improvement.

3 Factorized Sparse Approximate Inverses for SPD Matrices

We have seen that computing the ISAI preconditioner for general matrices requires all $\tilde{A}_i$ systems to be non-singular. While this requirement is fulfilled for an spd matrix A, the preconditioner matrix M resulting from applying the ISAI Algorithm (Listing 1.1) to an spd matrix is generally not again spd. In consequence, it is not possible to use the general ISAI preconditioner in combination with a CG solver that heavily relies on the spd property. A workaround is to compute an approximation to the (unknown) exact Cholesky factor $A = CC^T$. As we will review next, this *Factorized Sparse Approximate Inverse* (FSPAI [12]) preserves the spd-ness of the system matrix in the preconditioner.

For that, let $A = CC^T$ be the exact Cholesky factorization of A. Computing an ISAI of C on the sparsity pattern of the lower triangular part of A, $\mathcal{S}(\text{tril}(A))$ requires computing a matrix L with

$$LC = I \text{ on } \mathcal{S}(\text{tril}(A)). \tag{4}$$

Obviously, with C being unknown, this is not possible. However, multiplying C^T to (4) from the right yields

$$LCC^T = LA = C^T \text{ on } \mathcal{S}(\text{tril}(A)). \tag{5}$$

With $\mathcal{S}(C)$ only containing nonzero entries on the lower triangular part, this only puts restrictions on the main diagonal of L. While the diagonal entries of C are generally not available information, (5) is equivalent to computing an ISAI of A on $\mathcal{S}(C)$ up to diagonal scaling. Imposing the diagonal part of the ISAI condition (2) on the complete spd preconditioned system $LCC^TL^T = LAL^T$ yields the requirements

$$(LA)_{i,j} = 0, (i,j) \in \mathcal{S}(\text{tril}(A)), \quad i \neq j \text{ and} \tag{6}$$

$$(LAL^T)_{i,i} = 1, \quad i = 1, ..., n \tag{7}$$

for computing the FSPAI of A. Given the similarity between the ISAI and the FSPAI conditions, the FSPAI preconditioner can be computed efficiently with a modified ISAI implementation that appends diagonal scaling, see Listing 1.2. The computed triangular matrix L can then be used to transform the original system into a preconditioned system while preserving the spd property:

$$L^TLAx = L^TLb. \tag{8}$$

In practice, the FSPAI preconditioner can thus be implemented in the CG iterative solver as two matrix-vector multiplications with the FSPAI preconditioner L and L^T, respectively.

Listing 1.2. Algorithm computing an FSPAI preconditioner for a $n \times n$ spd matrix A on the sparsity pattern of tril(A).

```
L = isai(A); % on S(tril(A))
D = diag(L);
D = 1 ./ sqrt(D);
L = D * L;
```

4 Mixed Precision Incomplete and Factorized Approximate Inverse Preconditioners on GPUs

4.1 Generating the ISAI and FSPAI Operators

As discussed in Sect. 2, all rows of the preconditioner matrix M can be computed in parallel. For rows containing only a few nonzeros, the corresponding linear systems are of small size and can be solved efficiently via Gauss-Jordan elimination [5]. For rows containing many nonzeros, this approach quickly becomes unattractive due to the exploding computational cost and dense storage requirements of the Gauss-Jordan elimination. Therefore, we distinguish between "short" and "long" rows, treating them differently in the preconditioner generation.

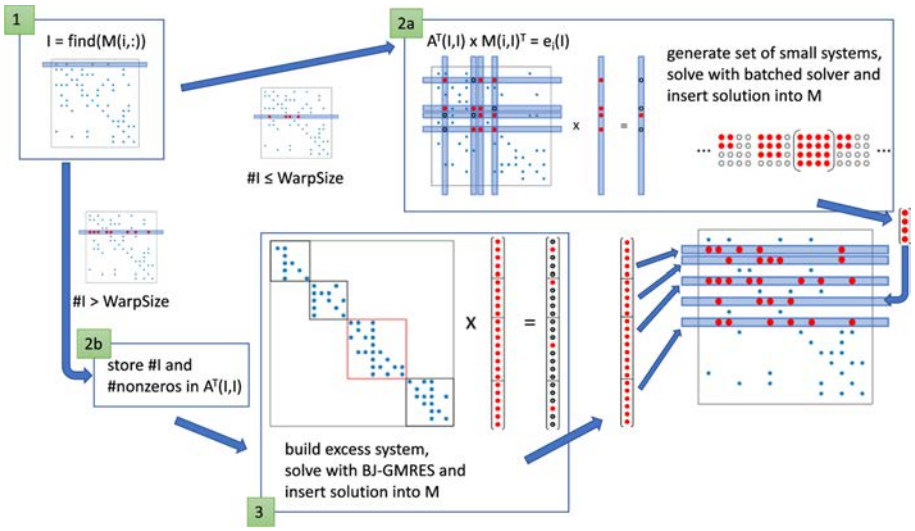

Fig. 1. Preconditioner generation of an ISAI preconditioner matrix M for a square, non-symmetric matrix A. The considered sparsity pattern for M is chosen to equal the sparsity pattern of A. For visualization purposes, the warp size is presumed to be 4.

To compute the ISAI of a matrix A stored in CSR format, the GPU implementation visualized in Fig. 1 launches one warp of size `WarpSize` (32 threads for NVIDIA GPUs) per row of A. The number of nonzeros in each row can simply be derived as the difference of the consecutive row pointer entries. Depending on the nonzero count, the computation of the values in this row is handled via a direct or an iterative method:

(a) $\leq$ `WarpSize` nonzeros: **Batched Gauss-Jordan Elimination with row pivoting**. The warp extracts the small local linear system into shared memory, each thread handling one row. After solving the local system via batched Gauss-Jordan elimination [5], the solution is written out to the preconditioner matrix.

(b) $>$ `WarpSize` nonzeros: **Block-Jacobi preconditioned GMRES.** The warp records the row id and the nonzero count for later processing.

To extract the rows and columns belonging to the small local system, we use an in-register implementation of the *MergePath* parallel merging algorithm [10] to find matching entries (i, j) between the current preconditioner row i and all corresponding rows j defining the extracted block. This strategy storing the block in row-major order for efficient processing via the Gauss-Jordan elimination avoids the otherwise necessary transposition of the local system matrix in shared memory.

After this step, all short rows of the preconditioner have been computed, and the algorithm has to process the long rows. Using the row IDs and the nonzero counts, we generate a block-diagonal "excess system" matrix placing

all local systems as a block on the main diagonal. This excess system is then approximately solved via a GMRES solver preconditioned with a block-Jacobi preconditioner, therewith making use of the excess system's inherent block structure. Finally, the solution is again split up and scattered into the corresponding preconditioner rows. Even though this strategy allows for also handling rows with nonzero counts larger than the warp size, this approach becomes unattractive for systems containing many nonzeros in a row, as then, the different blocks contain many redundancies as they were extracted from matching parts of A. We note that when working on triangular sparsity pattern, e.g. when computing an FSPAI preconditioner for an spd matrix, reordering can help to reduce the number of nonzeros accumulated in a single row, see Fig. 2.

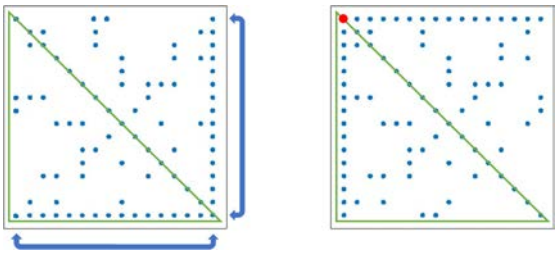

Fig. 2. For the FSPAI operating on the lower triangular part of the system matrix, reordering can help balancing of nonzero entries across the rows.

In general, any sparse matrix format can be used to store the ISAI or FSPAI preconditioners. However, the preconditioners are most efficient if all rows contain a similar number of nonzero elements and no row requires the generation of an excess system. On GPUs, matrices with such a "balanced" nonzero distribution can be handled efficiently via the ELL format. This format enforces all rows to have the same number of nonzeros by explicitly storing zero values for padding. Once a uniform row length is enforced, the ELL format allows for efficient SIMD processing, enabling the ELL SpMV to achieve much higher performance than the CSR-based SpMV [8]. Therefore, after generating the ISAI and FSPAI preconditioners, we convert them to the ELL format to allow for efficient SpMV processing in the preconditioner application.

4.2 Mixed Precision Preconditioner Application

The preconditioner application is among the most expensive operations within most Krylov solvers, aside from the multiplication with the system matrix itself. So to improve the runtime of the preconditioner application, we employ a mixed precision technique that has previously been used successfully for block-Jacobi preconditioning [9]: Since preconditioners are just an approximation to the inverse system matrix, it is often possible to add small rounding errors by

storing their entries in a lower precision format. Due to the typically memory-bound performance characteristics of sparse numerical linear algebra kernels, a reduction in the memory footprint of the preconditioner storage translates almost directly into a performance improvement of the preconditioner application. When using such low precision storage, we have to consider two aspects: 1) The rounding errors introduced by lower storage precision can cause the preconditioner itself to lose regularity, which may prevent convergence of the overall Krylov method; 2) The computations themselves must still be performed in high precision, otherwise we would lose precision in the iteration vectors and, due to the preconditioner application becoming a non-constant linear operator, would require the use of a flexible Krylov solver [4].

Fig. 3. The *memory accessor* converts StPrec values to ArPrec on the fly, reducing memory movement.

The implementation of mixed precision operations in the Ginkgo library makes use of the so-called *memory accessor* [3] to provide a uniform interface decoupling the storage precision from the arithmetic precision used to perform computations. We consider three different configurations for the preconditioner storage and application: While always performing computations in IEEE 754 double precision, we store the preconditioner values in double (FP64), single (FP32) or half precision (FP16). The corresponding preconditioners are denoted by ISAI<ArPrec, StPrec> and FSPAI<ArPrec, StPrec>, where ArPrec indicates the precision format used in the arithmetic operations and StPrec indicates the precision format used for storing the preconditioner. The *memory accessor* reads StPrec values from memory and converts them to ArPrec values on the fly before applying the preconditioner matrix, see Fig. 3.

For the full implementation of the discussed preconditioners, we refer to the Ginkgo github repository[1].

5 Numerical Experiments

In this section, we evaluate the performance of the mixed precision ISAI and mixed precision FSPAI preconditioners storing the numeric values in lower precision while performing all arithmetic in double precision. This means that the

[1] https://github.com/ginkgo-project/ginkgo/pull/719.

preconditioner generation is the same for all precision combinations. The objective of our experiments is to study the effect using a lower precision memory format in the preconditioner application has on the numerical stability and runtime of Krylov methods. Thus, we compare and validate our mixed-precision preconditioners against the full (double) precision versions of the same preconditioners and refrain from repeating experiments showing their general validity (see [7,12]). For the experimental evaluation, we use test matrices of moderate size (more than 20,000 rows; less than 50,000,000 nonzeros) that generally have a low nonzero-per-row ratio. We list the selected non-symmetric and spd test matrices along with some key properties in Tables 1 and 2 in the Appendix. In the experimental evaluation, we first investigate the general ISAI preconditioner in the context of a BiCGSTAB method solving non-symmetric problems. Afterward, we assess an FSPAI-preconditioned CG solver for the spd problems. If not specifically stated otherwise, we use a relative residual stopping criterion of $\frac{\|b-Ax\|}{\|b\|} < 10^{-7}$ and a hard iteration limit of 20,000 iterations for BiCGSTAB and 10,000 iterations for CG, respectively. We always use an all-zero initial guess and a right-hand side of all ones.

5.1 Hardware Setup

The GPU we use in our experiments is a NVIDIA V100 PCIe with 16 GB of main memory, a memory bandwidth of 900 GB/s, and a theoretical peak performance of 7 TFLOPS in double precision [13]. All code and functionality necessary to reproduce the results are publicly available via the benchmark suite of the Ginkgo open source library [2]. The CUDA toolkit version 10.2 was used to compile Ginkgo and the kernels generating and applying the preconditioners.

5.2 General ISAI

To evaluate the preconditioner quality of the general ISAI algorithm, we use a preconditioned BiCGSTAB solver for the 39 non-symmetric test-matrices listed in Table 1. In all these cases, an ISAI preconditioner stored in double precision is able to reduce the number of BiCGSTAB iterations. To assess the preconditioner quality degradation introduced by rounding the numeric values to lower precision, we visualize in the top plot in Fig. 4 the ratio between BiCGSTAB preconditioned with mixed precision ISAI and BiCGSTAB preconditioned with double precision ISAI. We recall that the mixed precision ISAI preconditioners preserve double precision in all arithmetic operations, but employ either single precision or half precision for the memory operations (denoted with ISAI<FP64,FP32> and ISAI<FP64,FP16>, respectively). The mixed precision ISAI employing single precision for the memory operations preserves the convergence of the BiCGSTAB solver for all problems, introducing a convergence delay of more than 3% only for the EPB3 problem. Conversely, ISAI<FP64,FP16> fails to preserve the preconditioner quality for about half the problems.

Ignoring the potential convergence delay, we visualize in the center of Fig. 4 the mixed precision ISAI speedup over the double precision ISAI. We note that

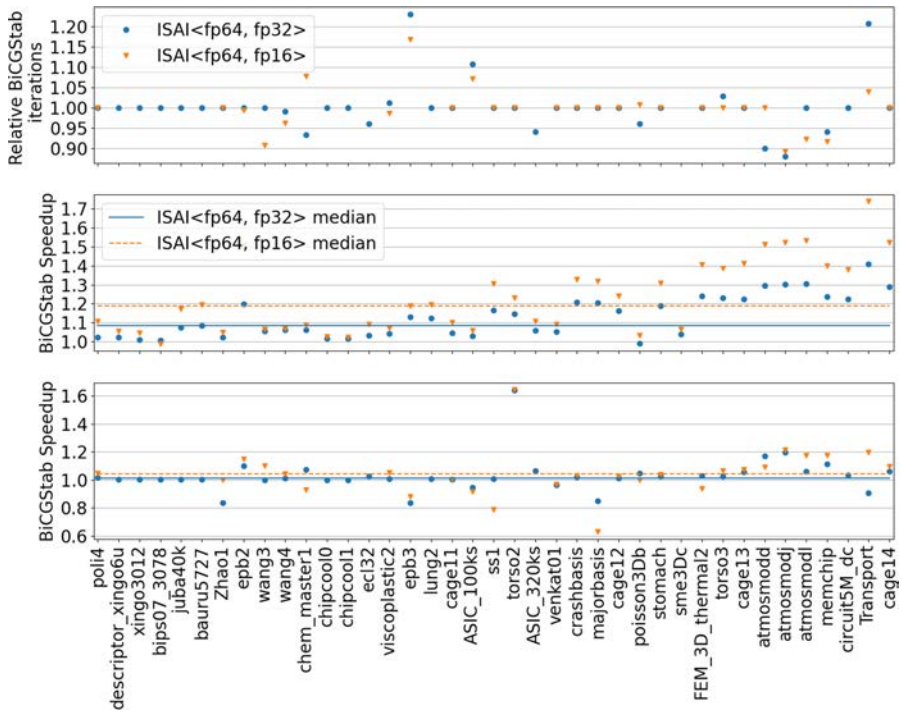

Fig. 4. *Top*: Relative iteration counts for BiCGSTAB using lower precision preconditioner storage compared to double precision. Cases where half precision use results in lower iteration counts are related to rounding effects. *Center*: Speedup of a single mixed precision ISAI application vs. double precision. The speedup ratios ignore the quality degradation of the preconditioner when using low precision storage. *Bottom*: BiCGSTAB speedup when using an ISAI preconditioner stored in lower precision instead of a double precision ISAI. Missing dots indicate the loss of convergence when using the mixed precision ISAI variant. The horizontal lines display the median among all values without loss of convergence. BiCGSTAB runtimes do not include the preconditioner generation. *Note*: The matrices are sorted according to their nonzero count along the x-axis.

the format conversion between memory precision and arithmetic precision is completely hidden behind the memory access, and all speedups are a direct result of the reduced memory footprint. While generally growing with the number of nonzero entries, the speedup reflects the intertwined relation between nonzeros, precision format, cache size, and the interplay of sparsity pattern and vector data reuse. As expected, the speedups are generally larger when using half precision storage, making this an attractive choice if the numerical properties allow for it.

The center of Fig. 4 not only ignores the potential numerical breakdown of an ISAI preconditioner when rounding to lower precision, but also the fact that the preconditioner application accounts only for a fraction of a BiCGSTAB solver

iteration. On the bottom of Fig. 4, we quantify the actual savings rendered to the total execution time of a BiCGSTAB iterative solver when replacing a double precision ISAI with its mixed precision variant. We note that additional BiCGSTAB iterations are hardly compensated by faster ISAI application, but for mild convergence variations, the speedups of the mixed precision ISAI preconditioner translate to moderate runtime reduction of the overall BiCGSTAB solver.

5.3 FSPAI

To evaluate the mixed precision FSPAI preconditioner, we use the spd test matrices listed in Table 2. Most of these problems are arising from 2D or 3D finite element discretizations.

The top plot in Fig. 5 indicates that the FSPAI preconditioner quality is barely affected from storing the preconditioner values in lower precision. Using the FSPAI<FP64,FP16> variant, the CG solver fails for the CRYSTM03

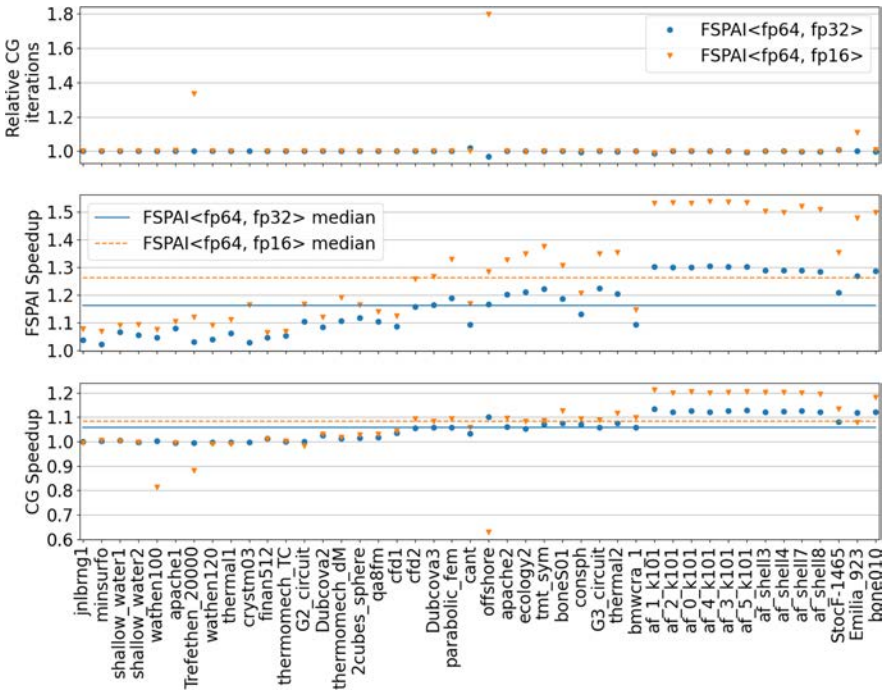

Fig. 5. *Top*: Relative iteration counts for CG using lower precision preconditioner storage compared to double precision. *Center*: Speedup of a single mixed precision FSPAI application vs. double precision. The speedup ratios ignore the quality degradation of the preconditioner when using low precision storage. *Bottom*: CG speedup when using an FSPAI preconditioner stored in lower precision instead of a double precision FSPAI.

problem and the convergence degrades for four additional problems. At the same time, the convergence is only mildly delayed for all other problems. The FSPAI<FP64,FP32> provides the same preconditioner quality as the double precision FSPAI preconditioner.

The center plot of Fig. 5 visualizes the mixed precision FSPAI performance, indicating a clear relation between the nonzero count and the speedup over the double precision FSPAI. For single precision storage, we can accelerate the preconditioner application by up to 1.3×, for half precision storage, this number grows to up to over 1.5×. Across the 46 test problems, we see a median performance improvement of 14% for FSPAI<FP64,FP32> and 27% for FSPAI<FP64,FP16>, respectively.

The bottom plot in Fig. 5 reflects how the mixed precision FSPAI speedups translate into mild but consistent performance benefits for the CG solver. We observe that the overall CG execution time is reduced by up to 20% (median: 8%) when using FSPAI<FP64,FP16> and up to 13% (median: 4%) when using FSPAI<FP64,FP32>, respectively.

6 Summary and Outlook

In this paper, we proposed mixed precision incomplete sparse approximate inverse preconditioners and mixed precision factorized sparse approximate inverse preconditioners that decouple the memory precision from the arithmetic precision and store the preconditioner information in a more compact low precision format. Investigating the numerical effects, we observed that the mixed precision preconditioners are often able to preserve the convergence improvement of their full precision counterparts. We also developed a high performance GPU implementation of the mixed precision preconditioners that achieve speedup ratios corresponding to the memory volume savings. Employing the mixed precision preconditioners into Krylov solvers, we demonstrated that these speedups translate into a mild acceleration of the iterative solution process. Though the runtime savings are incremental, we are convinced that the approach we take can serve as a blueprint for other algorithms, and that the production-ready mixed precision sparse approximate inverse preconditioners we provide can become an attractive building block in the configuration of high performance Krylov solvers. In future work, we will investigate the use of non-standard precision formats that can render higher speedups and focus on designing a cheap mechanism that protects the mixed precision preconditioners from numerical breakdown.

Acknowledgments. This work was supported by the US Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and the "Impuls und Vernetzungsfond" of the Helmholtz Association under grant VH-NG-1241. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Appendix

Table 1. Nonsymmetric test problems along with key properties and BiCGstab iterations. The preconditioners are double precision (<FP64,FP64>) or mixed precision (<FP64,FP32> or <FP64,FP16>). Iteration counts marked with "x" indicate convergence failure.

Non-symmetric test matrices						
Name	#rows	#nonzeros	BiCGSTAB	+ISAI<FP64,FP64>	+ISAI<FP64,FP32>	+ISAI<FP64,FP16>
poli4	33,833	73,249	36	19	19	19
descriptor_xingo6u	20,738	73,916	16,441	17	17	x
xingo3012	20,944	74,386	18,901	17	17	x
bips07_3078	21,128	75,729	5,455	17	17	x
juba40k	40,337	144,945	x	17	17	x
bauru5727	40,366	145,019	x	17	17	x
Zhao1	33,861	166,453	62	25	25	25
epb2	25,228	175,027	672	165	165	164
wang3	26,064	177,168	392	226	226	205
wang4	26,068	177,196	866	206	204	198
chem_master1	40,401	201,201	801	402	375	433
chipcool0	20,082	281,150	x	161	161	x
chipcool1	20,082	281,150	x	151	151	x
ecl32	51,993	380,415	x	486	467	x
viscoplatic2	32,769	381,326	x	1,897	1,919	1,871
epb3	84,617	463,625	x	3,431	4,225	4,009
lung2	109,460	492,564	x	78	78	x
cage11	39,082	559,722	32	14	14	14
ASIC_100ks	99,190	578,890	x	28	31	30
ss1	205,282	845,089	12	7	7	7
torso2	115,967	1,033,473	55	11	11	11
ASIC_320ks	321,671	1,316,085	x	51	48	x
venkat_01	62,424	1,717,792	x	57	57	57
crashbasis	160,000	1,750,416	293	42	42	42
majorbasis	160,000	1,750,416	144	22	22	22
cage12	130 228	2 032 536	25	14	14	14
poisson3Db	85,623	2,374,949	387	275	264	277
stomach	213,360	3,021,648	x	36	36	36
sme3Dc	42,930	3,148,656	x	17,603	x	x
FEM_3D_thermal2	147,900	3,489,300	551	26	26	26
torso3	259,156	4,429,042	309	69	71	69
cage13	445,315	7,479,343	30	15	15	15
atmosmodd	1,270,432	8,814,880	513	268	241	268
atmosmodj	1,270,432	8,814,880	498	276	243	246
atmosmodl	1,489,752	10,319,760	329	155	155	143
memchip	2,707,524	13,343,948	x	407	383	373
circuit5M_dc	3,523,317	14,865,409	x	16	16	x
Transport	1,602,111	23,487,281	4,355	1,215	1,467	1,263
cage14	1,505,785	27,130,349	23	15	15	15

Table 2. Symmetric positive definite test problems along with key properties and CG iterations. The preconditioners are double precision (<FP64,FP64>) or mixed precision (<FP64,FP32> or <FP64,FP16>). Iteration counts marked with "x" indicate convergence failure.

Symmetric positive definite test matrices						
Name	#rows	#nonzeros	CG	+FSPAI<FP64,FP64>	+FSPAI<FP64,FP32>	+FSPAI<FP64,FP16>
jnlbrng1	40,000	199,200	100	54	54	54
minsurfo	40,806	203,622	59	39	39	39
shallow_water1	81,920	327,680	12	7	7	7
shallow_water2	81,920	327,680	23	13	13	13
wathen100	30,401	471,601	228	22	22	22
apache1	80,800	542,184	1,550	1,025	1,025	1,029
Trefethen_20000	20,000	554,466	770	3	3	4
wathen120	36,441	565,761	229	27	27	27
thermal1	82,654	574,458	1,384	666	666	666
crytm03	24,696	583,770	91	9	9	x
finan512	74,752	596,992	37	8	8	8
thermomech_TC	102,158	711,558	60	7	7	7
G2_circuit	150,102	726,674	5,198	682	682	682
Dubcova2	65,025	1,030,225	167	121	121	121
thermomech_dM	204,316	1,423,116	60	7	7	7
2cubes_sphere	101,492	1,647,264	x	6	6	6
qa8fm	66,127	1,660,579	59	10	10	10
cfd1	90,656	1,825,580	1,727	874	874	873
cfd2	123,440	3,085,406	7,083	1,991	1,992	1,991
Dubcova3	146,689	3,636,643	170	118	118	118
parabolic_fem	525,825	3,674,625	1,366	1,177	1,177	1,177
cant	62,451	4,007,383	9,452	5,423	5,525	5,440
offshore	259,789	4,242,673	x	311	302	559
apache2	715,176	4,817,870	4,458	1,868	1,868	1,868
ecology2	999,999	4,995,991	5,896	3,167	3,167	3,166
tmt_sym	726,713	5,080,961	3,389	2,067	2,067	2,067
boneS01	127,224	5,516,602	2,331	1,012	1,013	1,013
consph	83,334	6,010,480	x	1,618	1,609	1,618
G3_circuit	1,585,478	7,660,826	8,442	824	824	824
thermal2	1,228,045	8,580,313	5,079	2,560	2,559	2,561
bmwcra_1	148,770	10,641,602	x	6,709	6,725	6,700
af_0_k101	503,625	17,550,675	x	8,739	8,739	8,742
af_1_k101	503,625	17,550,675	x	7,495	7,413	7,427
af_2_k101	503,625	17,550,675	x	9,051	9,052	9,052
af_3_k101	503,625	17,550,675	x	7,099	7,101	7,094
af_4_k101	503,625	17,550,675	x	9,335	9,360	9,334
af_5_k101	503,625	17,550,675	x	9,184	9,148	9,145
af_shell3	504,855	17,562,051	1,967	882	882	879
af_shell4	504,855	17,562,051	1,967	882	882	879
af_shell7	504,855	17,579,155	1,963	882	881	881
af_shell8	504,855	17,579,155	1,963	882	881	881
StocF-1465	1,465,137	21,005,389	x	5,062	5,109	5,089
Emilia_923	923,136	40,373,538	x	5,004	5,014	5,547
bone010	986,703	47,851,783	x	8,437	8,421	8,508

References

1. Suitesparse matrix collection. https://sparse.tamu.edu
2. Anzt, H., et al.: Ginkgo: a high performance numerical linear algebra library. J. Open Source Softw. **5**(52), 2260 (2020). https://doi.org/10.21105/joss.02260
3. Anzt, H., Cojean, T., Grützmacher, T.: Technical report: Design of the accessor. LLNL Report LLNL-SR-818775, January 2021
4. Anzt, H., Dongarra, J., Flegar, G., Higham, N.J., Quintana-Ortí, E.S.: Adaptive precision in block-jacobi preconditioning for iterative sparse linear system solvers. Concurrency Comput. Practice Exp. **31**(6), e4460 (2019)
5. Anzt, H., Dongarra, J., Flegar, G., Quintana-Ortí, E.S.: Batched gauss-jordan elimination for block-jacobi preconditioner generation on gpus. In: Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2017, pp. 1–10. Association for Computing Machinery, New York (2017). https://doi.org/10.1145/3026937.3026940
6. Anzt, H., Flegar, G., Grützmacher, T., Quintana-Ortí, E.S.: Toward a modular precision ecosystem for high-performance computing. Int. J. High Performance Comput. Appli. **33**(6), 1069–1078 (2019). https://doi.org/10.1177/1094342019846547
7. Anzt, H., Huckle, T.K., Bräckle, J., Dongarra, J.: Incomplete sparse approximate inverses for parallel preconditioning. Parallel Comput. **71**, 1–22 (2018). https://doi.org/10.1016/j.parco.2017.10.003
8. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. SC 2009. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1654059.1654078, https://doi.org/10.1145/1654059.1654078
9. Flegar, G., Anzt, H., Cojean, T., Quintana-Ortí, E.S.: Customized-precision Block-Jacobi preconditioning for Krylov iterative solvers on data-parallel manycore processors. ACM Trans. Math. Softw. (2020). under review. Available from the authors
10. Green, O., McColl, R., Bader, D.A.: GPU merge path: A GPU merging algorithm. In: Proceedings of the 26th ACM International Conference on Supercomputing, ICS 2012, pp. 331–340. ACM. https://doi.org/10.1145/2304576.2304621
11. Grote, M.J., Huckle, T.: Parallel preconditioning with sparse approximate inverses. SIAM J. Sci. Comput. **18**(3), 838–853 (1997). https://doi.org/10.1137/S1064827594276552
12. Kolotilina, L.Y., Yeremin, A.Y.: Factorized sparse approximate inverse preconditionings i. theory. SIAM J. Matrix Anal. Appl. **14**(1), 45–58 (1993). https://doi.org/10.1137/0614004
13. NVIDIA Corp.: Whitepaper: NVIDIA TESLA V100 GPU ARCHITECTURE (2017)
14. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. SIAM (2003)