Batched Generation of Incomplete Sparse Approximate Inverses on GPUs

Hartwig Anzt*, Edmond Chow†, Thomas Huckle‡, and Jack Dongarra*§
*Innovative Computing Laboratory, University of Tennessee, {hanzt, dongarra}@icl.utk.edu
†School of Computational Science and Engineering, Georgia Institute of Technology, echow@cc.gatech.edu
‡Department of Informatics, Technical University Munich, huckle@in.tum.de
§University of Manchester; Oak Ridge National Laboratory

***Abstract*—Incomplete Sparse Approximate Inverses (ISAI) have recently been shown to be an attractive alternative to exact sparse triangular solves in the context of incomplete factorization preconditioning. In this paper we propose a batched GPU-kernel for the efficient generation of ISAI matrices. Utilizing only thread-local memory allows for computing the ISAI matrix with very small memory footprint. We demonstrate that this strategy is faster than the existing strategy for generating ISAI matrices, and use a large number of test matrices to assess the algorithm's efficiency in an iterative solver setting.**

***Keywords*-Incomplete Sparse Approximate Inverses, Preconditioning, GPU, Batched routines**

I. Introduction

Preconditioning iterative solvers is a central aspect in the efficient solution of large, sparse linear systems. The underlying concept is to use a preconditioner that improves the conditioning of a linear system such that the convergence of the iterative solver is enhanced. However, a preconditioner is only attractive if the provided convergence improvement compensates for the additional work: the preconditioner generation, and the preconditioner application in the distinct solver iterations. In the context of High-Performance Computing (HPC), the preconditioner efficiency depends on how well these building blocks scale on parallel architectures. Among the most popular preconditioners are the incomplete LU factorizations (ILU [1]). These approximate the factorization of the system matrix on some nonzero pattern, and use sparse triangular solves in the distinct iteration steps for transforming the original problem into the preconditioned one. Unfortunately, the convergence improvement of ILU preconditioning comes at the price of difficult-to-parallelize sparse triangular solves. On parallel architectures, these can become a bottleneck. Given this background, much effort is spent of parallelizing sparse triangular solves [2], [3], and developing techniques that approximate the solution of the sparse triangular systems at a higher concurrency level, see [4], [5], [6]. An interesting approach in this context is the strategy of approximating the inverse of the triangular factors via an Incomplete Sparse Approximate Inverse (ISAI) preconditioner [7]. The ISAI concept arises as a combination of the incompleteness strategies used in Jacobi, ILU, or Sparse Approximate Inverse (SAI) preconditioners, with Frobenius norm minimization [8].

Using the ISAI strategy in the context of incomplete factorization preconditioning has shown to be attractive for several reasons [7]: 1) The preconditioner application boils down to a multiplication with the sparse ISAI matrix (or multiple sparse matrix vector multiplications, in case of using the ISAI matrix for fixed-point steps) — on parallel architectures, this is typically much faster than exact triangular solves via substitution; 2) For many problems, the ISAI matrix succeeds in handling the ill-conditioning of the preconditioner; 3) The generation of the ISAI matrix decomposes into solving a set of small linear systems – an embarrassingly parallel task.

In [7], the generation of the ISAI preconditioner is realized via a set of "batched routines" [9]. These routines are designed to apply a sequence of operations to a large set of data entities, and have proven to be very efficient on streaming architectures like GPUs. In the original paper, the ISAI matrix is generated by a sequence of kernels: first, all the small linear systems are generated in GPU main memory; then the systems are solved via a batched triangular solve routine; and finally, the solutions are inserted into the sparse data structure of the ISAI preconditioner.

In this paper we propose using one single batched routine for the ISAI matrix generation on GPUs. We design a kernel that avoids forming the small linear systems explicitly, but loads (and keeps) the distinct rows only in registers, instead. Starting from there, the solves are realized via thread-to-thread register shuffles, and the solution is inserted immediately into the sparse data structure of the ISAI preconditioner. This strategy dramatically reduces the memory footprint. We show that the proposed kernel outperforms the original strategy, and assess the performance of the implementation in an iterative solver setting.

II. Background and Related Work

A. Batched routines for efficient GPU utilization

The term "batched routines" refers to routines where a certain operation is applied not only to one single data entity, but to a large set of data entities. These data entities are typically small, and the absence of dependencies makes the routines embarrassingly parallel. On parallel architectures, applying the operation to one data entity may not fully utilize the hardware. Scheduling one data entity after another

978-1-5090-5222-6/16 $31.00 © 2016 IEEE
DOI 10.1109/ScalA.2016.11

may leave computational resources unused. A batched implementation applies the operation to all data entities simultaneously, and hence allows for more significant use of the parallel hardware. On streaming processors, like GPUs, an additional advantage comes from the reduced kernel launch overhead from a single batched function call, versus making multiple kernel calls, one for each linear system. Examples of the superiority of batched implementations over baseline implementations often focus on BLAS routines like batched matrix-matrix multiplication [10], batched factorizations [9], and batched triangular solves [11].

B. Incomplete Sparse Approximate Inverses (ISAI)

The recently proposed Incomplete Sparse Approximate Inverse (ISAI) aims at approximating the inverse of a system matrix A via sparse matrix M such that $M \cdot A \approx I$ [7]. For $\mathcal{J}$ being the set of nonzero indices of the i-th row of M, the ISAI matrix minimizes

$$\min_{M(\mathcal{J},j)} \|A(\mathcal{J},\mathcal{J})M(\mathcal{J},j) - I(\mathcal{J},j)\|_2,$$

respectively

$$A(\mathcal{J},\mathcal{J})M(\mathcal{J},j) = I(\mathcal{J},j). \tag{1}$$

Hence, it can be seen as a simplification of the Sparse Approximate Inverse [12] that minimizes the Frobenius norm considering only the locations included in the sparsity pattern of M [8].

Combining all columns $M(\mathcal{J},j)$ leads to:

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

This implies that the approximation is exact in all locations included in the sparsity pattern $\mathcal{S}$ of M. For $\mathcal{S}$ being the main diagonal this results in $M = diag(A)^{-1}$ and thus links to the Jacobi preconditioner. Similarly, applying this approach to a (block) diagonal pattern results in the (block) Jacobi preconditioner. Hence, the approach (2) is a generalization of the block Jacobi method, allowing for general patterns, and computing the preconditioner columnwise. In the remainder, we use the notation $\mathcal{S}\left(|A|^k\right)$, k integer, to describe a sparsity pattern $\mathcal{S}$ that is defined by the pattern of the k-th power of the component-wise positive system matrix A. This implies that for $\mathcal{S}\left(|A|^k\right)$, the preconditioner leads to zeros of $AM - I$ in the prescribed pattern of $|A|^k$.

The linear systems (1) induced by $\mathcal{S}$, and the resulting preconditioner, are well defined only for $j \in \mathcal{J} \subseteq \mathcal{I}$, and nonsingular $A(\mathcal{J},\mathcal{J})$. This is satisfied, e.g., for A being triangular, and $\mathcal{S}$ containing the diagonal. Sparse triangular systems occur in the context of incomplete factorization preconditioning. In [7], it is shown that incomplete sparse approximate inverses are very attractive for incomplete factorization preconditioning on GPU-accelerated systems. The strategy is to replace the difficult-to-parallelize sparse triangular solves with approximate triangular solves using the ISAI matrix. For many problems, the convergence improvement from ISAI preconditioning is comparable to exact triangular solves, with each preconditioner application being much faster [7].

However, replacing the exact triangular solves with the ISAI preconditioner introduces some overhead in the preconditioner setup as the ISAI matrices have to be generated for the incomplete factors. In this paper we exclusively focus on designing and implementing an algorithm that realizes the ISAI generation efficiently. This is highly relevant as the algorithm can also be used to produce any kind of (block) Jacobi system.

The generation of the ISAI matrix requires solving a set of linear problems (1). More precisely, for each column of the ISAI matrix M, one linear system has to be solved. The size of the system providing the elements for the i-th column of M corresponds to the number of locations included in the pre-set sparsity pattern $\mathcal{S}(:,i)$. Typically, the systems are small for finite element discretizations, but they can become very large for circuit simulation problems, for example.

An important observation is that for approximating the inverse of a sparse triangular matrix, also the systems (1) are of triangular structure. Furthermore, the distinct systems (1) are independent, and do allow for concurrent handling. Hence, the ISAI preconditioner generation can be realized efficiently on parallel architectures.

Consider the derivation of an ISAI preconditioner M for a lower triangular matrix L coming from an incomplete LU factorization. Then, (2) becomes [7]:

$$(LM - I)_{i,j} = 0 \ \forall (i,j) \in \mathcal{S}, \tag{3}$$

and the ISAI matrix can be generated in parallel via Algorithm 1.

Algorithm 1 Algorithm computing ISAI preconditioner M for lower triangular matrix L [7].

```
Choose S(M_L) = S(|L|)
for j = 1 : n do
  m_{j,j} = 1/l_{j,j}
  for k = j + 1 : n and k ∈ S(L(:, j)) do
    m_{k,j} = 0
    for r = j : k − 1 and r ∈ S(L(:, j)) do
      m_{k,j} = m_{k,j} − l_{k,r} m_{r,j}
    end for
    m_{k,j} = m_{k,j} / l_{k,k}
  end for
end for
```

This algorithm can be implemented by generating and solving a set of small triangular systems [7]. All systems can be considered independently and in parallel, which makes this solution process a candidate for a batched implementation. Ultimately, the solutions for the distinct systems have

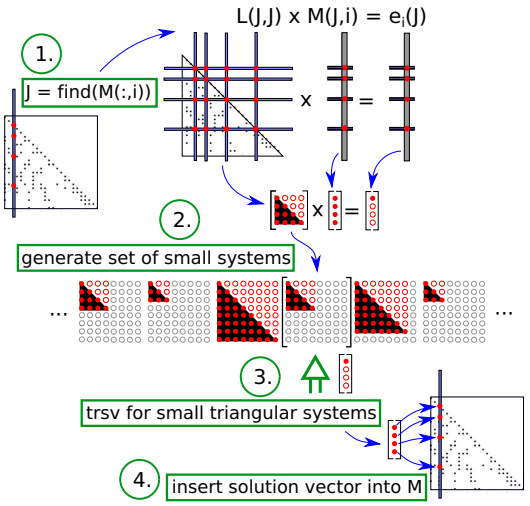

Figure 1. Generation of the ISAI preconditioner matrix M for a lower sparse triangular system L via batched routines [7]. The sparsity structure of the ISAI matrix is chosen to be consistent with the sparsity structure of the lower triangular system L.

to be combined to form the ISAI matrix. Also this step does not require adhering to a specific execution order or synchronizations. In [7], we have shown how a sequence of four batched routines can be used to realize the generation of the ISAI preconditioner. The set of data entities corresponds to the distinct columns of the ISAI matrix M. For each column i, the distinct routines:

1. $\mathcal{J} = find(M(:,i))$
   Collect all nonzero locations in the i-th column of the pre-defined sparsity structure $\mathcal{S}(M)$;
2. **generate** $L(\mathcal{J},\mathcal{J})$
   Generate the small system matrix by extracting the respective entries of the target matrix, (in our case, the lower incomplete factor L);
3. **solve** $L(\mathcal{J},\mathcal{J}) \cdot M(\mathcal{J},i) = I(\mathcal{J},i)$
   Solve the arising small (lower) triangular system;
4. **insert solution** $M(\mathcal{J},i)$ **into preconditioner**
   Back-insert the computed solution into the sparse structure of the ISAI matrix M.

Figure 1 visualizes the generation of the ISAI preconditioner M for a lower triangular factor [7]. In the remainder of the paper, we refer to the implementation generating the ISAI preconditioner via the sequence of batched routines as "baseline ISAI implementation," indicating that we consider this as state-of-the-art.

A disadvantage of the baseline ISAI implementation is the requirement for a pre-defined memory layout that allows handling and accessing the distinct small systems independently, indicated in step 2 in Figure 1. More precisely, the parallel generation of the small linear systems requires the pre-allocated memory to have a uniform size, consistent with the largest system that occurs in the ISAI generation. In [7], a uniform memory layout composing of entities of size 32×32 is chosen. This restricts the scope of the algorithm to ISAI patterns where no column contains more than 32 elements. A study considering a large number of matrices reveals that this setting allows for handling a good portion of the problems [7]. An advantage of imposing this upper bound is that it limits the range of the triangular solve (TRSV) scenarios, and motivates to use size-specific hard-coded implementations for the forward (and backward) substitutions.

The kernel generating the sequence of linear systems in memory is, in most cases, the computationally most expensive kernel. In particular, it puts a lot pressure on the memory bandwidth as it requires scanning the rows and columns of A and M for matches in the sparsity structure. However, using a batched kernel for the generation of all systems can achieve good performance as it allows for coalesced memory access, and benefits from good cache reuse. Also the batched triangular solve and the batched back insertion allow for coalesced memory reads and writes. Combining these batched routines, the algorithm achieves very good performance.

III. Batched ISAI Generation

Despite the good performance achieved by the algorithm presented in the last section, we aim to design and implement an algorithm that generates the incomplete sparse approximate inverses with one single kernel. The idea is to avoid the explicit formulation of the systems (1) in main memory. Instead, we load the elements in the distinct rows into registers of threads being part of the same thread block, and to handle the solution process without global communication. This strategy may sacrifice coalesced memory access, but benefit from reduced data transfer. Furthermore, handling system generation and solution process on-the-fly dramatically reduces the memory footprint of the algorithm, which is the main motivation for the approach. As a result, the scope of the algorithm is no longer limited by the size of the data structure in main memory that contains the batch of linear systems. The new algorithm is capable of generating ISAI preconditioners for patterns where no column of the sparsity structure $\mathcal{S}$ contains more elements than TBS, where TBS denotes the thread block size. For TBS$\leq$ 32, the threads of the thread block are all part of the same warp. Then, the TRSV can realize thread-to-thread communication via CUDA's __SHFL() instruction [13], like proposed in [11]. Figure 2 visualizes the algorithm generating the ISAI preconditioner using one single GPU kernel. Each thread block computes one column of the ISAI matrix, the number of thread blocks needed corresponds to the number of columns. All thread blocks have the same size, if this exceeds the number of nonzeros in the target column, the additional threads are terminated. To distinguish this

algorithm from the baseline ISAI implementation composed of multiple kernels, we denote this algorithm as "batched ISAI implementation."

In the generation of the ISAI matrices, we need to extract the values of the system matrix for the locations that are included in the ISAI sparsity pattern. For M and L stored in ordered CSR format, this can be done efficiently by simultaneously traversing the same rows in L and U, comparing the column-indices, and moving to the next element in the matrix where the index is smaller. The extraction step is completed for a row once the end of this row is reached in either L and U. Hence, to decrease the comparison effort, we want to quickly reach the end of the row in one matrix, while traversing as few elements as possible of the other matrix.

An interesting aspect in this context is the distribution of the nonzero locations in M and L, and the flexibility to traverse the rows either from left to right or from right to left. For $\mathcal{S}$ being a (block) Jacobi sparsity structure, the nonzeros in M aggregate on and near the main diagonal, and hence the values that are needed from the system matrix are also located in this region (see left and center plot in Figure 3). For this case, the extraction step is completed quickly if we traverse the entries in M and L starting from the main diagonal and moving left: the end of the row in M is reached after comparing only the locations in L that are part of the Jacobi block. Obviously, we should start from the main diagonal and move right if the system matrix has upper triangular structure. Using the ISAI structure $\mathcal{S}\left(|A|^k\right)$, $(k \geq 1)$ for M, all values of the system matrix are needed (see right of Figure 3). For $k = 1$, the nonzero pattern of L and M are identical, the extraction step needs to traverse all elements of the row, and the effort is independent of the scanning direction. For $k > 1$, it is difficult to make general statements, as it depends on where the additional locations in the sparsity structure $\mathcal{S}\left(|A|^k\right)$ of M are generated.

In the numerical experiments in Section IV, we use the generic setting of always traversing the rows from the diagonal outwards, i.e., we scan right-to-left when handling lower triangular factors and left-to-right when handling upper triangular factors.

IV. Numerical Experiments

A. Hardware and Software Environment

The architecture we target is a NVIDIA Tesla K40 GPU (Kepler microarchitecture) with a theoretical peak performance of 1,682 GFlop/s (double precision). The 12 GB of GPU main memory can be accessed at a theoretical bandwidth of 288 GB/s. The kernels generating the ISAI preconditioner are implemented in CUDA version 7.5 [13] and use a default thread block size of 32. All other functionalities, including the Krylov solvers, are taken from the MAGMA-sparse module which is part of the MAGMA open-source software library in version 2.0 [1]. For the generation of the incomplete triangular factors, MAGMA-sparse interfaces to the incomplete factorization routines available in NVIDIA's cuSPARSE library [13]. These exploit parallelism by using level-scheduling strategies [3]. All computations are handled by the GPU in double precision arithmetic.

The test matrices listed in Table I are the ones that have been considered to evaluate the efficiency of ISAI preconditioning in [7] are taken from the University of Florida Sparse Matrix Collection [2]. Although we focus on analyzing the performance of batched ISAI generation, we mention that we symmetrically scale these matrices to have a unit diagonal. This preprocessing has no impact on the performance of the algorithm generating the ISAI preconditioner, but ensures consistency with the data presented in [7]. The original ordering is used except for the AF3 and PAR problems, which are considered in Reverse Cuthill-McKee (RCM [14]) ordering. We generate the incomplete sparse approximate inverses for the triangular factors coming from incomplete LU factorizations without fill-in (ILU(0)).

In Section IV-D a large set of test matrices is used to assess the performance of the batched ISAI implementation. The test suite is composed of all matrices available at UFMC for which an IDR(4) iterative solver preconditioned with an ILU(0) preconditioner converges to a residual norm $\|r\| \leq 10^{-10}\|b\|$ within 10,000 iterations. Also, all problems smaller than 100 rows and problems where the ILU(0)-preconditioned IDR(4) from MAGMA-sparse completes within 0.1 s are considered "too easy" to require the use of a GPU solver. Applying these filters, we obtain a test suite containing 460 "reasonably difficult" problems.

For all experiments where we evaluate the ISAI preconditioner within an iterative solver, we use a right-hand-side $b \equiv 1$, start the iterations with the initial guess $x^0 \equiv 0$, and set the residual stopping criterion to $\|r\| \leq 10^{-10}\|b\|$.

Name	Abbr.	Nonzeros n_z	Size n
AF_SHELL3	AF3	17,562,051	504,855
ECOLOGY2	ECO	4,995,991	999,999
OFFSHORE	OFF	4,242,673	259,789
PARABOLIC_FEM	PARA	3,674,625	525,825
TMT_UNSYM	TMT	4,584,801	917,825
THERMAL2	THM	8,580,313	1,228,045

Table I
TEST MATRICES.

B. Comparing the performance of the ISAI implementations

In a first performance comparison, we focus on the special case of the ISAI matrix having (block-) Jacobi structure with different block sizes. For each of the linear systems, we compute the ILU(0) factorization, and then generate the

[1] MAGMA; http://icl.cs.utk.edu/magma/
[2] UFMC; https://www.cise.ufl.edu/research/sparse/matrices/

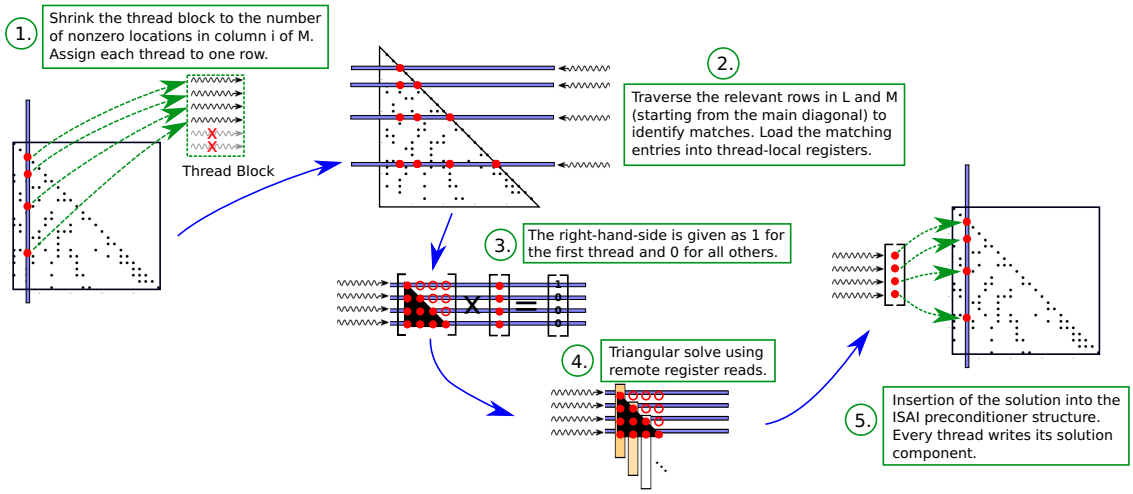

Figure 2. Generation of the ISAI preconditioner matrix M for a lower sparse triangular system L via one single batched routine. In this example, the sparsity structure of the ISAI matrix is chosen to be consistent with the sparsity structure of the lower triangular system L.

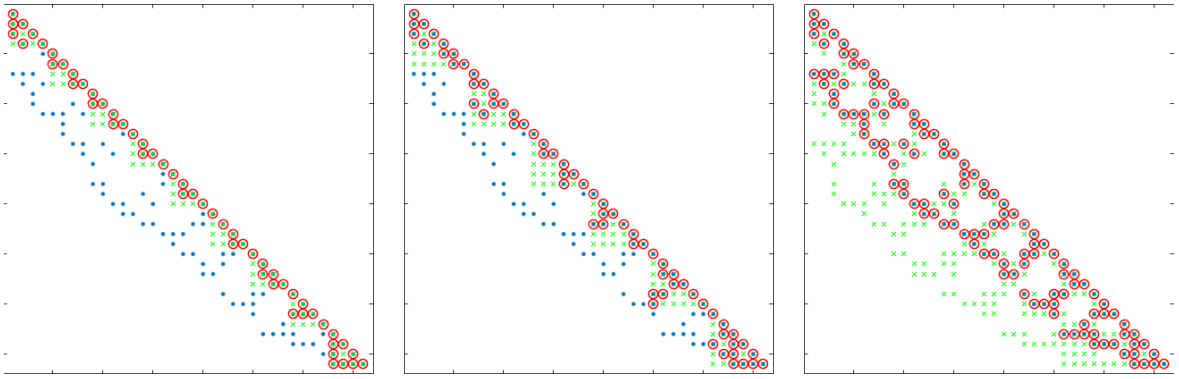

Figure 3. Identifying values that need to be extracted from the system matrix (red circles) by matching the sparsity patterns of the lower triangular ILU(0) factor L (blue dots) and the ISAI sparsity structure (green crosses). The sparsity pattern used for the ISAI matrix are a block Jacobi with block size 4 (left), a block Jacobi with block size 6 (center), and the pattern $\mathcal{S}\left(|A|^2\right)$ (right), respectively.

ISAI matrix M_L for the lower incomplete ILU(0) factor. Figure 4 visualizes the speedup of the batched ISAI implementation over the baseline ISAI implementation. A central observation from Figure 4 is that the speedup decreases with increasing Jacobi block size. For block Jacobi with diagonal blocks of size 32, the speedups mostly range between 2 and 4. For smaller block sizes, the batched ISAI implementation is up to 16 times faster. However, the speedup is very dependent on the linear system properties, and for some matrix structures, the speedup remains smaller.

Next, we compare the generation of ISAI preconditioners for sparsity patterns derived from the system matrix. We generate of the ISAI matrices M_L and M_U for the lower and upper triangular ILU(0) factors, and show in Figure 5 how much faster the batched ISAI implementation realizes this operation. Whenever possible, we consider the sparsity pattern $(|\mathcal{S}(A)|)$, $\left(|\mathcal{S}(A)|^2\right)$, $\left(|\mathcal{S}(A)|^3\right)$, and $\left(|\mathcal{S}(A)|^4\right)$. In the remainder, we use the notation ISAI(k) for the ISAI preconditioner with the nonzero pattern $\left(|\mathcal{S}(A)|^k\right)$. Unfortunately, the baseline ISAI implementation taken from [7] does not allow for the generation of ISAI matrices containing more than 32 nonzeros in one column.

C. ISAI in the iterative solver context

Aside from comparing the batched ISAI implementation to the baseline ISAI implementation, we quantify the absolute performance of the batched ISAI implementation in an iterative solver scenario. For this purpose, we use an Induced Dimension Reduction (IDR(s) [15]) iterative solver with a shadow space dimension $s = 4$. The IDR(s) is a very robust and efficient Krylov solver [16], and the GPU-implementation of the IDR(s) algorithm in MAGMA-sparse has proven to achieve performance close to the theoretical bound [15]. We enhance IDR(4) with an ILU(0) precondi-

Matrix	Preconditioner setup [s]			Preconditioner application [s]			IDR(4) Iterations			Solver execution [s]		
	ILU(0)	ISAI(1)	overhead	trisol	ISAI(1)	speedup	trisol	ISAI(1)	overhead	trisol	ISAI(1)	speedup
AF3	1.3672	0.7279	53.24%	0.6909	0.0129	53.46	329	1588	382.67%	234.2021	49.4370	**4.74**
ECO	0.3122	0.2889	92.56%	0.2116	0.0064	33.16	954	2933	207.44%	216.2284	65.1396	**3.32**
OFF	0.3276	0.2076	63.35%	0.4103	0.0044	92.74	148	440	197.30%	62.2157	5.8897	**10.56**
PARA	0.4857	0.1979	40.77%	0.2687	0.0045	60.09	139	730	425.18%	39.0847	11.0221	**3.55**
THM	0.8296	0.4543	54.77%	0.8084	0.0132	61.27	873	1755	101.03%	722.9693	60.0312	**12.04**
TMT	0.3009	0.2522	83.83%	0.1661	0.0060	27.90	755	1881	149.14%	136.0090	38.9953	**3.49**

Table II
PERFORMANCE OF THE BATCHED ISAI IMPLEMENTATION IN THE CONTEXT OF AN IDR(4) ITERATIVE SOLVER.

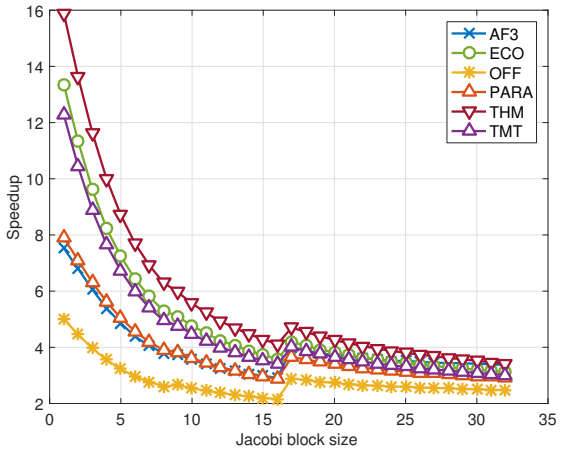

Figure 4. Speedup of the batched ISAI generation over the baseline ISAI implementation for generating the block Jacobi matrix for the incomplete lower ILU(0) factor.

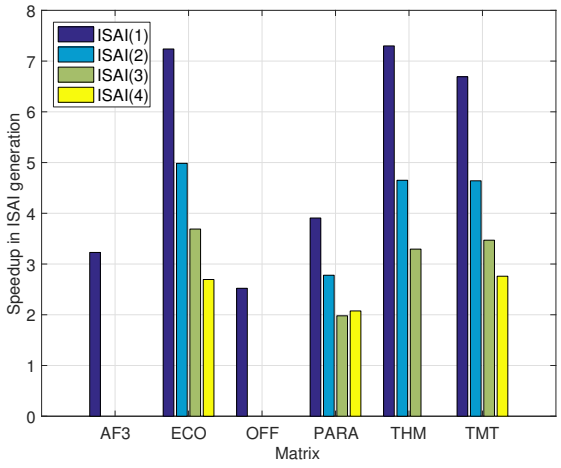

Figure 5. Speedup of the batched ISAI implementation over the baseline ISAI implementation for generating the ISAI matrices for the ILU(0) factors.

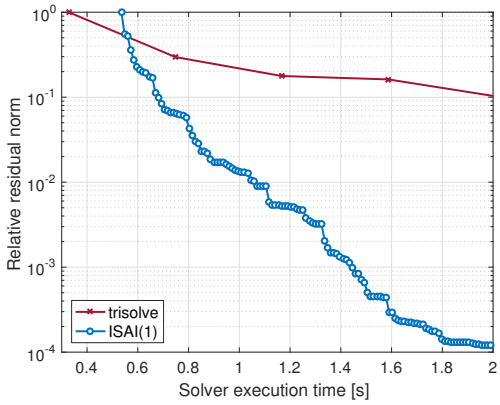

Figure 6. Relative residual norm vs. runtime for of the IDR(4) solver using exact triangular solves and ISAI-based approximate triangular solves. The ISAI matrix generation introduces some overhead to the preconditioner setup. Markers indicate each IDR(4) iteration.

tioner, and either use exact triangular solves with L and U for the preconditioner application, or the multiplication with the ISAI matrices M_L and M_U.

Table II shows the runtime needed to generate the ILU(0) preconditioner, and the runtime needed to generate the ISAI(1) matrices for the incomplete factors. Depending on the problem, adding the ISAI generation increases the preconditioner setup cost by a factor of up to 2. At the same time, the preconditioner application becomes much cheaper as the multiplication with the ISAI(1) matrix is typically much faster than the exact triangular solves. Depending on the specific problem characteristics, the preconditioner application is accelerated by up to two orders of magnitude. Replacing the exact triangular solves with sparse approximate inverse multiplications can diminish the preconditioner quality. This is reflected in the IDR(4) iteration count. The preconditioner quality can be improved by using the ISAI matrix in fixed-point iterations [7]. In this paper, we refrain from considering this strategy and always apply the ISAI preconditioner as a sparse approximate inverse. The trade-off between the overhead introduced to the preconditioner setup, the reduced preconditioner quality, and the faster preconditioner application determines whether replacing exact triangular solves with the ISAI preconditioner pays off. If

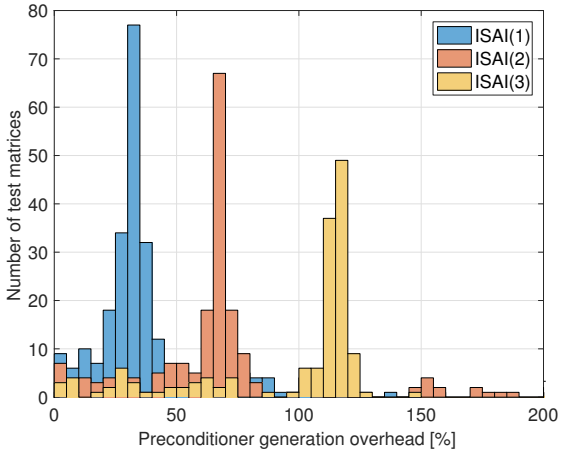

Figure 7. Histogram of the overhead of the ISAI generation for the incomplete triangular factors over the ILU(0) preconditioner setup.

only few iterations are needed, the faster preconditioner application may not compensate for the overhead in the preconditioner setup. Figure 6 visualizes the convergence with respect to solver execution time for the OFF problem. For the residual stopping criterion $\|r\| \leq 10^{-10}\|b\|$, the ISAI preconditioner is superior for all the test cases we consider in Table II. Depending on the problem, the total solver execution time is reduced by a factor as high as 12.

D. Batched ISAI performance case study

	Preconditioner generation			ILU(0) comparison		
Pattern	#	converges	overhead	faster	slower	speedup
ISAI(1)	251	219 (87%)	37.8%	187	32	7.71
ISAI(2)	192	171 (89%)	77.4%	158	13	10.56
ISAI(3)	164	146 (89%)	113.9%	136	10	11.49

Table III
ISAI STATISTICS ON THE TEST SUITE CONTAINING 460 UFMC TEST MATRICES: # OF SYSTEMS THE ISAI GENERATION SUCCEEDS RESPECTIVELY THE IDR(4) CONVERGES, THE ASSOCIATED OVERHEAD IN THE PRECONDITIONER GENERATION, AND THE COMPARISON AGAINST AN ILU(0) COMBINED WITH EXACT TRIANGULAR SOLVES. OVERHEAD AND SPEEDUP ARE AVERAGE VALUES.

To assess the performance of the batched ISAI generation, we use the test suite described in IV-A. Out of the 460 test matrices, the batched ISAI implementation succeeds in generating the ISAI(1) matrices for the ILU(0) factors of 251 matrices, see Table III. The remaining 209 matrices contain columns with more than 32 nonzero locations. These can easily identified in a preprocessing step. Also, as previously elaborated, the batched ISAI implementation can easily be extended to cover a larger portion of the test matrices. For the denser pattern ISAI(2) and ISAI(3), the generation succeeds for 192 and 164 systems, respectively. If the ISAI generation is successful, it induces overhead to the preconditioner setup. The statistics visualized in Figure 7 reveal that this overhead is typically about 40% for the ISAI(1) preconditioner, and increases to 70% and 120% when generating ISAI(2) and ISAI(3) matrices, respectively.

The successful preconditioner generation alone does not imply that replacing the exact triangular solves with the ISAI(1) preconditioner is recommended: the preconditioner quality might be diminished, making it possible that the combination ILU(0)+ISAI diverges, or converges slower than when using exact triangular solves. Both issues can be addressed by using the ISAI preconditioner for fixed-point iterations [7], which, however, remains outside the focus of this paper. The strategy of applying the ISAI as sparse approximate inverse succeeds for about 90% of the test-cases, see Figure 7. For these, IDR(4) typically converges more quickly when replacing the exact triangular solves with the ISAI preconditioner. ISAI(1), for example, is in 3 of 4 cases faster for the systems where the generation is successful, and considering also the 64 systems where exact triangular solves are the better choice, the ISAI(1) is on average still 7.71 times faster.

Finally, we assess the impact of the preconditioner setup time on the best choice of solver. On the left-hand side of Figure 8 we use the height of the bars to indicate for how many problems a certain configuration is the performance-winner ignoring the preconditioner setup time. As expected, there exists a number of test matrices where the ILU(0) has to be combined with exact triangular solves because the batched ISAI implementation fails to generate the sparse approximate inverse matrices, and an un-preconditioned or Jacobi-preconditioned IDR(4) fails to converge (faster) within the iteration limit of 1,000,000 iterations. We also recognize the embarassingly parallel Jacobi-preconditioner being the winner for a good portion of the problems. Although the ISAI preconditioner can only be generated for 219 of the 460 systems, it wins most cases.

On the right-hand side we include the preconditioner setup cost in the performance evaluation. We use migration bars to visualize for how many systems the optimal choice changes. Accounting for the preconditioner setup cost makes the un-preconditioned IDR(4) more attractive. Also the light-weight Jacobi preconditioner wins more cases. For ILU combined with the ISAI preconditioner, many systems migrate from the ISAI(3) to the ISAI(2) configuration. The share of the ISAI preconditioner in the total wins decreases, however, not by making the exact triangular solves more attractive: not a single test case migrates from the ILU+ISAI strategy to the plain ILU using exact triangular solves. This reveals that the cost of generating the ISAI matrices is not a crucial factor, and shows that the batched ISAI implementation we propose in this paper is sufficiently efficient for being used in an iterative solver setting.

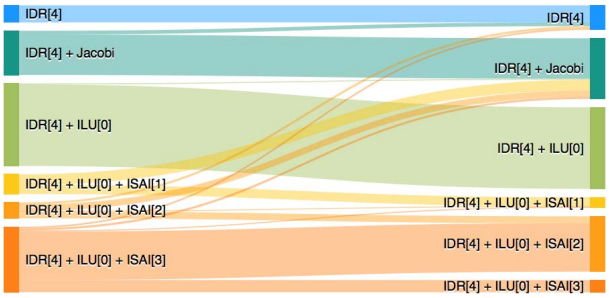

Figure 8. Choosing the runtime-optimal preconditioner: the height of the bars indicates for how many of the 460 test matrices a certain configuration is the fastest. The results on the left do not consider the preconditioner setup time; the results on the right consider the total solver execution time.

V. Conclusion

We have proposed a batched algorithm that generates an incomplete sparse approximate inverse preconditioner for sparse triangular systems via a single GPU kernel. Experimental results show that the new algorithm is typically faster than the previous strategy which was based on generating the ISAI matrices via a sequence of batched routines. For a test suite containing a large number of matrices, generating the ISAI(1) matrices for the ILU(0) factors makes the preconditioner setup phase about 40% more expensive. Experiments show that this overhead is typically easily compensated by the cheaper preconditioner application. We have shown that the choice of the triangular solver in an ILU(0) context is independent of the preconditioner setup cost. This indicates that the batched ISAI generation satisfies the demand of not impacting the iterative solver efficiency. Future research will extend the batched ISAI implementations enabling the generation of ISAI matrices with more than 32 elements in a column.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC0016564 and DE-SC0016513, and NVIDIA.

References

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[2] M. Benzi, W. Joubert, and M. G., "Numerical experiments with parallel orderings for ILU preconditioners," *Electronic Transactions on Numerical Analysis*, vol. 8, pp. 88–114, 1999.

[3] M. Naumov, P. Castonguay, and J. Cohen, "Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU," NVIDIA, Tech. Rep., 2015.

[4] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM Journal on Scientific Computing*, vol. 37, pp. C169–C193, 2015.

[5] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *Euro-Par 2015: Parallel Processing*, ser. Lecture Notes in Computer Science, J. L. Träff, S. Hunold, and F. Versaci, Eds. Springer Berlin Heidelberg, 2015, vol. 9233, pp. 650–661.

[6] E. Chow and J. Scott, "On the use of iterative methods and blocking for solving sparse triangular systems in incomplete factorization preconditioning," Rutherford Appleton Laboratory, Tech. Rep. Technical Report RAL-P-2016-006, 2016.

[7] H. Anzt, T. Huckle, J. Bräckle, and J. Dongarra, "Incomplete Sparse Approximate Inverses for Parallel Preconditioning," *SIAM Journal on Scientific Computing*, submitted.

[8] M. W. Benson, "Iterative solution of large scale linear systems," Master's thesis, Lakehead University, Thunder Bay, 1973.

[9] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance tuning and optimization techniques of fixed and variable size batched cholesky factorization on GPUs," *Procedia Computer Science*, vol. 80, pp. 119–130, 2016, international Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[10] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Performance, design, and autotuning of batched GEMM for gpus," in *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, 2016, pp. 21–38.

[11] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra, "Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1045-9219, no. 1045-9219, 2015.

[12] L. Y. Kolotilina and A. Y. Yeremin, "Factorized sparse approximate inverse preconditionings i. theory," *SIAM Journal on Matrix Analysis and Applications*, vol. 14, no. 1, pp. 45–58, 1993.

[13] NVIDIA Corp., *CUDA C Programming Guide, v7.5*, September 2015.

[14] I. S. Duff and G. A. Meurant, "The effect of ordering on preconditioned conjugate gradients," *BIT*, vol. 29, no. 4, pp. 635–657, 1989.

[15] H. Anzt, M. Kreutzer, E. Ponce, G. D. Peterson, G. Wellein, and J. Dongarra, "Optimization and performance evaluation of the IDR iterative Krylov solver on GPUs," *International Journal of High Performance Computing*, 2016.

[16] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. Koehler, "Efficiency of general krylov methods on gpus – an experimental study," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 683–691.