Hindawi
Mathematical Problems in Engineering
Volume 2021, Article ID 5558508, 13 pages
https://doi.org/10.1155/2021/5558508

Research Article

GPU Preconditioning for Block Linear Systems Using Block Incomplete Sparse Approximate Inverses

Wenpeng Ma,[1] **Yiwen Hu**,[1] **Wu Yuan**,[2] **and Xiazhen Liu**[2]

[1]*College of Computer and Information Technology, Xinyang Normal University, Xinyang, Henan 464000, China*
[2]*Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China*

Correspondence should be addressed to Wenpeng Ma; mawp@xynu.edu.cn

Received 5 January 2021; Revised 29 March 2021; Accepted 9 May 2021; Published 26 May 2021

Academic Editor: Hussein Abulkasim

Solving sparse triangular systems is the building block for incomplete LU- (ILU-) based preconditioning, but parallel algorithms, such as the level-scheduling scheme, are sometimes limited by available parallelism extracted from the sparsity pattern. In this study, the block version of the incomplete sparse approximate inverses (ISAI) algorithm is studied, and the block-ISAI is considered for preconditioning by proposing an efficient algorithm and implementation on graphical processing unit (GPU) accelerators. Performance comparisons are carried out between the proposed algorithm and serial and parallel block triangular solvers from PETSc and cuSPARSE libraries. The experimental results show that GMRES (30) with the proposed block-ISAI preconditioning achieves accelerations $1.4\times$ –$6.9\times$ speedups over that using the cuSPARSE library on NVIDIA Tesla V100 GPU.

1. Introduction

Sparse triangular solves are essential steps for the incomplete LU- (ILU-) factorized preconditioning [1] in a Krylov subspace solver. However, they are sequentially designed and pose a performance challenge on today's parallel computers. To make full use of the state of art parallel architectures such as multicore CPUs and GPU accelerators, various works have been studied and presented. The well-known algorithm is the level-scheduling scheme [1–5] which groups the possible parallelism in levels of sets each of which can be performed in parallel. The implementations on multicore architectures and GPU accelerators are discussed in [1, 4, 5] and [2, 3], respectively. The NVIDIA's cuSPARSE [6] library offers a function interface for this algorithm which can be called by users directly in practice. However, there are two drawbacks in some of the applications. One is that the extracted parallelism depends on the sparsity pattern of the matrix; thus, the parallelism and performance is highly limited for some cases. The other is that it is usually a time-consuming job to search parallelism (known as preprocessing) before the actual computation is conducted. Liu et al. [7, 8] proposed a method in which the preprocessing step is not necessarily performed when the sparse matrix is expressed in compressed sparse column (CSC) format, and they showed performance improvement on GPU over the level scheduling method in cuSPARSE.

Compared to using exact computation for preconditioning, some inexact preconditioning ideas [9–13] are attractive in recent years because they seek a tradeoff between exactness and parallelism. Chow et al. [12, 13] proposed a fine-grained algorithm to compute ILU factorization asynchronously on Intel MIC architecture [14] and GPUs. They showed that 5 asynchronous sweeps usually make the inexact ILU factorization comparable to the exact one, but with a significant factor of speedups. Anzt et al. [9] implemented an asynchronous method on GPU for solving triangular systems using several number of Jacobi iterations. Although the inexact preconditioning step results in more number of solver iterations, it shows an advantage over the exact preconditioning in terms of the total compute time of the linear solver. Some methods are based on sparse approximate inverses (SAI) [15–19] where the matrix inverse of the triangular factors is estimated through solving least squares problems on a preset pattern. And the resulting inverses can transform the preconditioning step into sparse

matrix-vector multiplications with much more possible of concurrency computing on parallel computers. Most recently, Anzt et al. [10, 11] proposed an incomplete SAI (ISAI) algorithm in which the least square problems are replaced with solving square systems with cheaper computations and faster convergence.

Linear systems arising from block sparse matrices are widely used in scientific computing especially in multiphysics problems. Most of the numerical algorithms for block linear systems are derived from that for scalar systems. Even though they are quite similar to each other in a mathematical formula, the implementation strategies and performance tuning techniques could be very different from each other, especially on GPU accelerators. For example, the block sparse matrix-vector multiplication (BSpMV) on GPU [20] is performed in the multiplications of blocks and vectors, whereas a general SpMV is realized in scalar multiplications. Due to the principles of global memory access and the use of shared memory on the GPU, direct migration of the code from scalar case to block case may hamper the performance much [20]. Therefore, algorithms for block matrices usually require redesigned work. In multiphysics problems, the coupling feature of the physical fields results in block matrices that usually have blocks with a small size such that the inverse of a block can be explicitly expressed. Motivated by this kind of numerical applications and the ISAI preconditioning proposed by Anzt et al. [10, 11], we focus on the GPU preconditioning in a block format in this study. The main contributions of this study are the following.

(1) The GPU preconditioning framework [10, 11] is extended to block matrices with block sizes up to 5.

(2) An efficient, warp-based GPU implementation exploiting fine-grained programming model for block-ISAI preconditioning is proposed and elaborately explained.

(3) Detailed comparisons are made between the proposed algorithm and block triangular solvers from popular libraries, including PETSc [21] and cuSPARSE [6]. On block matrices selected from the SuiteSparse collection [22] and real multiphysics areas, the proposed algorithm shows an advantage over the PETSc's serial and cuSPARSE's parallel implementations of block triangular solvers in terms of total computing time for GMRES (30).

The rest of this study is organized as follows. In Section 2, some backgrounds, including sparse approximate inverse (SAI), ISAI, and block matrices, are introduced. In Section 3, the GPU preconditioning framework for block linear systems is proposed, and the strategy for GPU implementation is introduced and discussed. In Section 4, performance results and comparisons for four testing cases are shown. Concluding remarks are given in Section 5.

2. Background

2.1. Incomplete Sparse Approximate Inverses. For a given sparse matrix $A_{n\times n}$ with n rows and n columns, the SAI algorithm [1, 15–19] gives an approximation of the inverse of A by minimizing the Frobenius norm of $(\mathrm{AW} - I)$ as

$$\min_{W\in S_W} \|\mathrm{AW} - I\|_F^2 = \sum_{j=0}^{n-1} \min_{W_j\in S_W} \left\|\mathrm{AW}_j - I_j\right\|_2^2, \tag{1}$$

where W is the estimated inverse of A for a given sparsity pattern S_W, I is the identity matrix, W_j and I_j represent the j^{th} column of W and I, respectively. Solving (1) is equivalent to solve

$$\sum_{j=0}^{n-1} \min_{W_j(J)} \left\|A(R,J)W_j(J) - I_j(R)\right\|_2^2, \tag{2}$$

where J is the non-zero pattern for W_j, R represents the indices of the rows that contain non-zero values at J columns, and $A(R,J)$ is a submatrix extracted from the R^{th} rows and J^{th} columns of A. Note that $J\subseteq R$ if the diagonal values of A are non-zeros, and (2) can be viewed as the solve of a batch of least-squares problems. The SAI algorithm fits parallel computers because the problems in (2) can be solved independently of each other.

The work in [10, 11] proposed an incomplete-SAI (ISAI) by considering each problem in (2) as

$$\min_{W_j(J)} \left\|A(J,J)W_j(J) - I_j(J)\right\|_2^2. \tag{3}$$

This simplification makes $A(R,J)$ a square matrix, and the least-squares problems in (2) are changed into linear systems in (3). In this way, the ISAI can be easily applied to the matrices having special sparsity patterns. For example, in the case where incomplete LU (ILU) factorizations-based preconditioning is performed, the inverse of the lower triangular factor (L) and upper triangular factor (U) are approximately calculated with high concurrency using (3) [10, 11]. This avoids solving large sparse triangular systems which are regarded as the bottleneck in today's state-of-art parallel computers and accelerators.

2.2. Block Sparse Matrix. Block sparse matrices are widely used in scientific computing applications, especially in multiphysics problems. Figure 1 shows a 3×3 block sparse matrix with a block size of 2. It is neither a general sparse matrix nor a dense matrix, but it falls somewhere in between. It has the sparse property in global but dense features in local blocks.

There are many storage formats for block sparse matrices, such as block compressed sparse row (BCSR) and block compressed sparse column (BCSC) in PETSc [21] and cuSPARSE [6]. Different from the general CSR format for storing scalar values, BCSR format treats each block as a unit and stores all non-zero positions block by block (Figure 1). The values in each block are stored consecutively either in a row-major or column-major format. In the following parts of this study, block matrices are focused on, and column-major format is employed to store non-zero blocks in BCSR or BCSC formats.

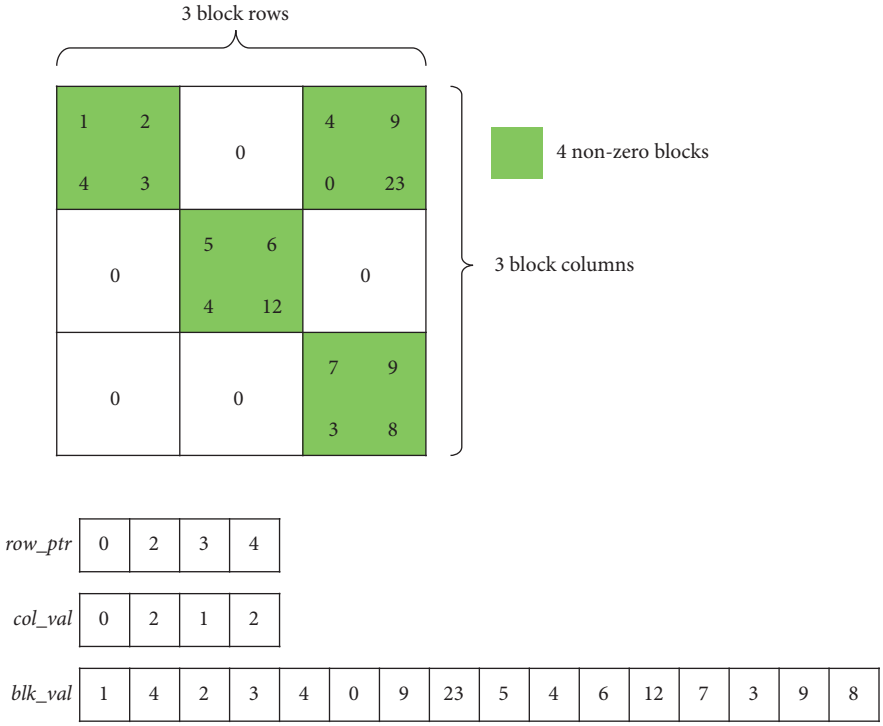

Figure 1: Example of 3×3 block matrix with four non-zero blocks, and the three arrays storing the matrix in BCSR with column-major within block.

3. Block-ISAI Preconditioning on GPUs

To solve an asymmetric block linear system $A_B x = b$, the Krylov subspace-based generalized minimal residual (GRMES) algorithm [23] with right preconditioning is employed as

$$\begin{aligned} A_B M^{-1} x_d &= b, \\ x &= M^{-1} x_d, \end{aligned} \tag{4}$$

where M is a preconditioner.

The well-known method to construct a preconditioner M is to perform a block incomplete LU factorization (BILU) on A_B [21, 24]. Then, the preconditioning step in which the inverse of M applies to a vector v, i.e., $M^{-1}v$, can be executed by

$$M^{-1}v = z \Rightarrow LUz = v, \tag{5}$$

where L and U are the block lower and upper triangular matrices factorized by A_B, respectively, and z is the preconditioned vector.

3.1. Block-ISAI on GPU. Traditionally, the blocked-based forward and backward substitutions are applied to the two triangular systems in (5). However, these are totally sequential operations. Instead of computing (5) accurately, Anzt et al. [10, 11] introduced an inexact idea in which the inverse of the lower and upper factors are estimated by highly parallel ISAI, and (5) is transformed into the problems of ISAI relaxation steps or ISAI SpMV. This method seeks a tradeoff between parallelism and exactness and shows an advantage in terms of total computing time of a linear solver.

Motivated by the scalar version of ISAI on a single GPU [10, 11] and the block version [25] on Intel's MIC (many integrated core) architecture [14], we propose a GPU-enabled block-ISAI algorithm for preconditioning in block matrices. For the convenience of our statement, we assume all the matrices in the following discussion are block matrices.

In the context of incomplete factorization preconditioners, we applied the incompleteness idea of (3) to the block lower factor L and upper factor U to have the estimations of the inverses, NL and NU, for the factors as

$$\begin{aligned} &\min_{NL_j(J_l)} \left\| L(J_l, J_l) NL_j(J_l) - I_j(J_l) \right\|_2^2, \\ &\min_{NU_j(J_u)} \left\| U(J_u, J_u) NU_j(J_u) - I_j(J_u) \right\|_2^2, \end{aligned} \tag{6}$$

where J_l and J_u are the indices of non-zero blocks at the j^{th} block column of L and U, respectively. As $L(J_l, J_l)$ and $U(J_u, J_u)$ are square block matrices, the solve of (6) is equivalent to the solve of a series of triangular systems in the block format independently

$$\begin{aligned} L(J_l, J_l) NL_j(J_l) &= I_j(J_l), \quad \text{and } U(J_u, J_u) NU_j(J_u) \\ &= I_j(J_u) \quad (j = 0, 1, 2, \ldots, n-1). \end{aligned} \tag{7}$$

The sizes of the systems in (7) are determined by the numbers of non-zero blocks in block columns, and they

usually are much smaller than the general large sparse systems. Therefore, the solve of the systems provides relatively fine-grained parallelism that fits the GPU architecture. We show the GPU-enabled block-ISAI in Algorithm 1.

To compute the preconditioned vector z, Algorithm 1 applies the preconditioner M to the input vector v on the GPU. Specifically, the GPU accepts the preconditioner M expressed in two block triangular factors L and U and a vector v as input parameters and outputs the preconditioned vector z. This can be implemented in two phases. The first phase aims to obtain the approximate inverses for L and U (denoted NL and NU, respectively) by the block-ISAI consisting of four main steps. The first step gives the guesses of the sparsity patterns for NL and NU. Following the strategy in [10, 11], we use $S(|L|^k)$ and $S(|U|^k)$ $(k \geq 1)$ as the sparsity patterns for NL and NU, respectively, where $S(|L|^k)$ and $S(|U|^k)$ represent the sparsity patterns for the multiplications of k times of absolute values of L and U, respectively. The second step extracts two block triangular systems, denoted $L(J_l, J_l)$ and $U(J_u, J_u)$, according to the non-zero patterns J_l and J_u at each column of NL and NU. The extraction process is illustrated in Figure 2, which shows how the block lower triangular system corresponding to the fourth column of NL is formed. The non-zero pattern for NL is constructed using $S(|L|^2)$ which is denser than $S(|L|)$. The non-zero pattern $J_l = \{4, 5, 7, 8\}$ for the fourth column of NL provides a row and column set (J_l, J_l) for indexing L, and then, the 4^{th}, 5^{th}, 7^{th}, and 8^{th} block rows and columns are extracted from L as a small block lower triangular matrix. In the third step, looping over all block columns of NL (NU) and storing all corresponding triangular matrices consecutively, we form two groups of small block triangular systems, LGSBTS and UGSBTS. The fourth triangular system in the LGSBTS, also shown in Figure 2, is formed by

$$L(J_l, J_l)\text{Sol}_4 = I_4(J_l), \tag{8}$$

where Sol_4 is the solution for the fourth block column of NL, and $I_4(J_l)$ is the right hand side with the first block being identity matrix. The last step is to solve the LGSBTS and UGSBTS block column by block column for the solutions of NL and NU, respectively. Generally, since the preconditioning matrix $M = \text{LU}$ remains unchanged during the solve of $A_B x = b$, the first phase needs to be performed only once.

Once the approximate solutions of NL and NU are obtained, the second phase can be accomplished by performing two block sparse matrix-vector multiplications, i.e., $y = \text{NL}v$ and $z = \text{NU}y$. These two operations must be conducted in every iteration because v changes at every iteration.

3.2. GPU Implementation. Although the block version of the ISAI looks similar to the scalar version, the computations in the two versions are entirely different. For example, Sol_4 in (8) consists of four blocks instead of four scalars, matrix-matrix multiplications is conducted instead of scalar-scalar multiplications, and the inverse of a non-zero block is computed instead of the inverse of a scalar. Therefore, to make full use of fine-grained features of a GPU, the implementations and tuning strategies for the two versions are also different. In the following discussion, an implementation of Algorithm 1 is introduced by developing several efficient GPU kernels and exploiting the cuSPARSE library [6].

For the first step in S1 of Algorithm 1, the essence of estimating the sparsity patterns of NL and NU is to conduct block matrix-matrix multiplication, which usually involves symbolic multiplication and numerical multiplication. However, since only the non-zero patterns of NL and NU are needed, only the symbolic multiplication of L^k and U^k is employed on the GPU. This is realized by calling the presetup function which estimates the non-zero pattern for the multiplication result of two general sparse matrices.

In the scalar version of the ISAI, there are two strategies introduced in [10, 11] for the GPU implementation, including the following three steps. One can separately design three kernels, each of which is responsible for a single step of Algorithm 1. In this way, the LGSBTS and UGSBTS must be formed explicitly for the solve [10]. An alternative method [11] is to merge all three steps into a single kernel in which the data $L(J_l, J_l)$ (or $U(J_u, J_u)$) extracted from L (or U) are directly used for the partial solution of one system from LGSBTS (UGSBTS). This means each system is formed temporarily, and the data for one system could be overwritten by subsequent systems. The main advantage of the first strategy is that one can explicitly express the LGSBTS (or UGSBTS) in an order that the memory pattern is perfect for coalesced memory accesses on the GPU. However, the drawback is obvious. To solve NL's j^{th} block column with $\text{nnzbl}(j)$ blocks, according to (7), it requires to explicitly storing the corresponding $\text{nnzbl}(j) \times \text{nnzbl}(j)$ lower triangular block matrix that contains $((\text{nnzbl}(j) \times (\text{nnzbl}(j) + 1))/2)$ blocks, i.e., $(s^2 \times \text{nnzbl}(j) \times (\text{nnzbl}(j) + 1)/2)$ scalar elements, where s is the block size. By adding the number of blocks up for all nbr block columns in NL targeting parallel computing, it requires approximately $s^2 \times \sum_{j=1}^{\text{nbr}} ((\text{nnzbl}(j) \times (\text{nnzbl}(j) + 1))/2)$ double precision memory spaces for storing all matrices in (7). It is estimated that explicitly storing LGSBTS for $\text{nbr} = 200,000$, $s = 3$, and $\text{nnzbl}(j) = 10$ requires over 750 MB of memory, and the memory requirement exceeds 2 GB when the block size is 5. The memory requirements will double if the LGSBTS and UGSBTS are both stored. By considering the space complexity of the first strategy and the limited memory resources on the GPU, in the block format, explicitly forming the LGSBTS and UGSBTS in the implementation proposed in this study is avoided.

A key point in S1 is the solving of the small block triangular systems (SBTSs) that fit the GPU architecture very well because they are independent of each other. However, the GPU strategy [10, 11] for the scalar version is not an optimized choice for the block version here because mapping a thread to a matrix block for numerical operations leads to significantly uncoalesced memory accesses on the GPU. Moreover, the experiments on block sparse matrix-vector multiplication in [20] show that using consecutive threads (usually 32 threads in a warp) collaboratively for the

Input:
(1) The BILU triangular factors, L and U ($\mathbf{M} = \mathbf{LU}$);
(2) Input vector, v;
Output:
(1) The preconditioned vector, z;
S1: **setup step**:
S1.1: estimate the sparsity patterns for the inverses of L and U expressed as NL and NU: $S(\mathrm{NL}) = S(|L|^k)$ and $S(\mathrm{NU}) = S(|U|^k)$;
S1.2: extract $L(J_l, J_l)$ and $U(J_u, J_u)$ from L and U, where J_l and J_u are the non-zero patterns at the j^{th} column of NL and NU, respectively, indexing J_l^{th} block rows and columns in L to form $L(J_l, J_l)$ and indexing J_u^{th} block rows and columns in U to form $U(J_u, J_u)$;
S1.3: form two groups of small block triangular systems in (7), LGSBTS: $L(J_l, J_l)\mathrm{NL}_j(J_l) = I_j(J_l)$ and UGSBTS: $U(J_u, J_u)\mathrm{NU}_j(J_u) = I_j(J_u)$, by looping overall J_l and J_u;
S1.4: solve LGSBTS and UGSBTS for the solutions of NL and NU: solve n block lower and upper triangular systems in (7) in parallel.
S2: preconditioning step:
S2.1: perform BSpMV: $y = L^{-1}v = \mathrm{NL}v$;
S2.2: perform BSpMV: $z = U^{-1}y = \mathrm{NU}y$.

Algorithm 1: Block-ISAI preconditioning on GPU: $z = \mathbf{M}^{-1}v$.

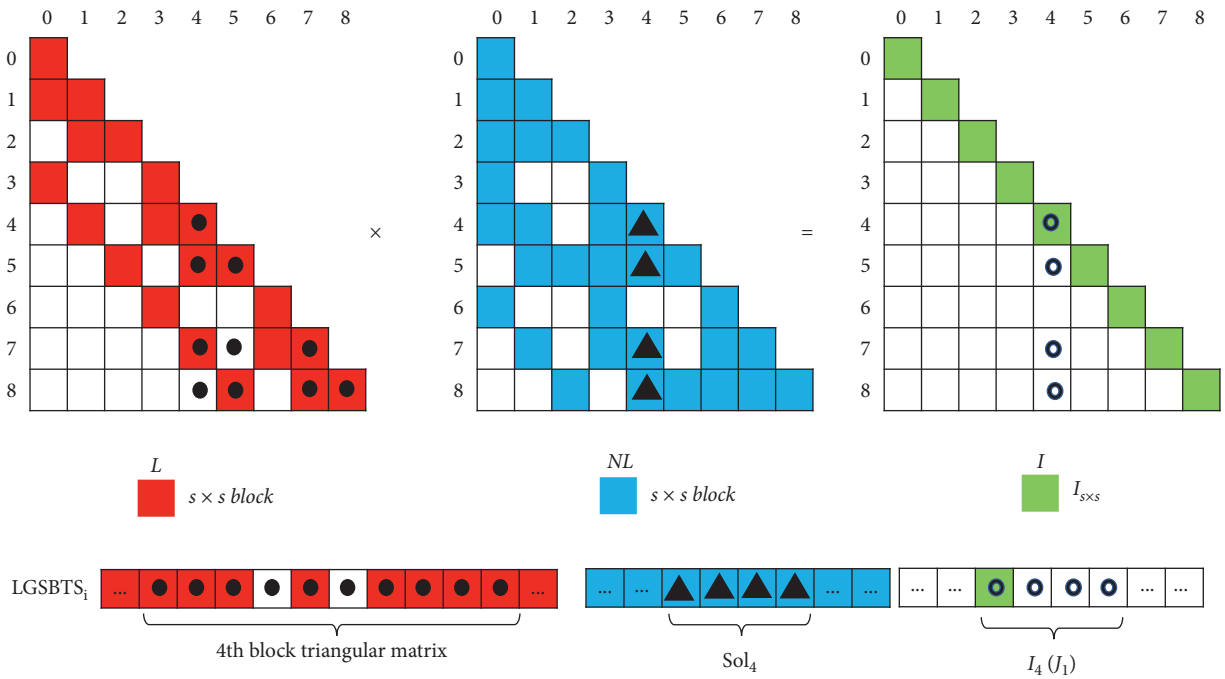

Figure 2: Illustration of extracting block triangular matrix and forming groups of triangular systems for solving NL (s is the block size).

computing of consecutive blocks is good for bandwidth utilization. Therefore, 32 threads in a warp are employed for the solve of a SBTS. The idea of the algorithm is shown in Figure 3, which shows the first warp in a thread block of size 128 for the solution of (8) with block size 4 in parallel. The small block triangular matrix (SBTM) is extracted from L in columns, so that a warp can handle a column with possible data concurrency. Specifically, when the j^{th} block column is accessed by a warp, the j^{th} block of Sol_4, denoted as $\mathrm{Sol}_4(j)$, is ready to be solved, and once $\mathrm{Sol}_4(j)$ is obtained, it can contribute to the solving of $\mathrm{Sol}_4(k)\,(k > j)$ by performing in parallel

$$I_4(k) = I_4(k) - \mathrm{SBTM}(k, j)\mathrm{Sol}_4(j), \quad (k > j). \tag{9}$$

Taking the SBTS's 0^{th} block column consisting of four blocks as an example, the 0^{th} block is used to solve $\mathrm{Sol}_4(0)$, and this is followed by accumulating $I_4(1)$ and $I_4(2)$ simultaneously. As a warp can only cover two 4×4 blocks, it restarts the computation for the rest (i.e., $I_4(3)$) in the next cycle, until it has gone through all blocks in the column.

The GPU kernel developed by the CUDA C/C++ language for solving the LGSBTS in Algorithm 2 is now described. The one-dimensional thread organization is employed. Since a warp (32 threads) is chosen and implicitly

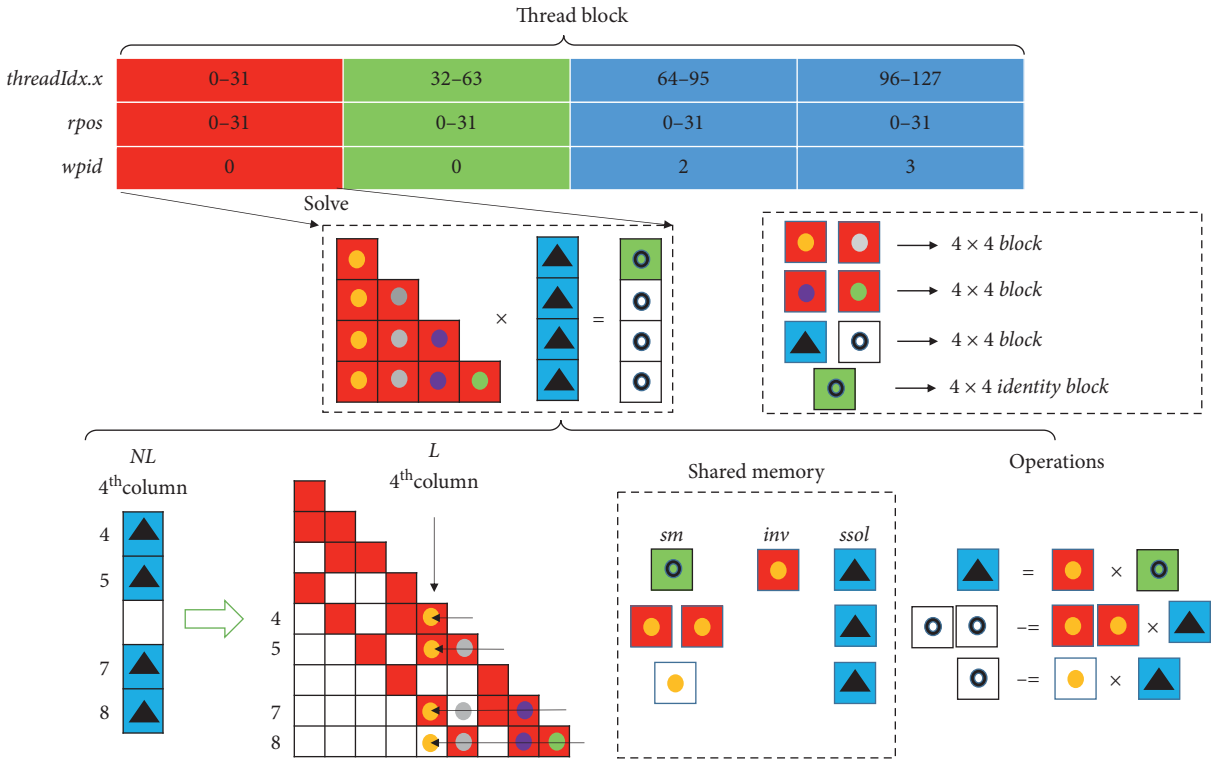

Figure 3: Illustration of warp-based formation of a SBTM and solving a SBTS.

```
(1) __shared__ double sm[warps_per_block][warp_size];
(2) __shared__ double inv[warps_per_block][bsz][bsz];
(3) __shared__ double ssol[warps_per_block][bsz][bsz];
(4) tid = threadIdx.x + blockIdx.x × blockDim.x;
(5) wpid = threadIdx.x/warp_size;
(6) gwpid = tid/warp_size;
(7) rpos = threadIdx.x%warp size;
(8) range = (warp_size/bsz^2) × bsz^2;
(9) if gwpid < nbr then
(10)   if rpos < range then
(11)     r = rpos%bsz;
(12)     c = (rpos/bsz)%bsz;
(13)     InvStIdx = NLColPtr[gwpid];
(14)     InvEdIdx = NLColPtr[gwpid + 1];
(15)     for idx = InvStIdx to InvEdIdx − 1 do
(16)       irow = NLRowVal[idx];
(17)       if rpos < bsz^2 then
(18)         tmpidx = LRowPtr[irow + 1] − 1;
(19)         ssol[wpid][r][c] = L[tmpidx ×bsz^2 + rpos];
(20)         inv[wpid][r][c] = 1.0/det ×inverse(ssol)_{r,c};
(21)         sm[wpid][rpos] = Rhs[idx ×bsz^2 + rpos];
(22)         ssol[wpid][r][c] = ∑_{t=0}^{bsz−1} inv[wpid][r][t] × sm[wpid][c × bsz + t];
(23)         NL[idx ×bsz^2 + rpos] = ssol[wpid][r][c];
(24)       end if
(25)       myidx = idx + 1 + (rpos/bsz^2);
(26)       triEdIdx = triStIdx + (n − icol) ×bsz^2;
(27)       solIdx = solStIdx + bsz^2 + rpos;
(28)       while myidx < InvEdIdx do
(29)           sm[wpid][rpos] = 0.0;
(30)           myrow = NLRowVal[myidx];
```

Algorithm 2: Continued.

```
(31)              tmpidx = LRowPtr[myrow + 1] − 1;
(32)              while tmpidx ≥ LRowPtr[myrow] && LColVal[tmpidx] == irow do
(33)                tmpidx−−;
(34)              end while
(35)              if tmpidx ≥ LRowPtr[myrow] && LColVal[tmpidx] == irow then
(36)                s[wpid][lane] = L[tmpidx ×bsz² + rpos%bsz²];
(37)              end if
(38)            Rhs[myidx ×bsz² + lane%bsz²]− = ∑_{t=0}^{bsz−1} sm[wpid][(lane/bsz²) × bsz² + t × bsz + r] × ssol[wpid][t][c];
(39)            myidx+ = warp_size/bsz²;
(40)          end while
(41)        end for
(42)      end if
(43)  end if
```

Algorithm 2: GPU kernel for solving LGSBTS.

dispatched by CUDA (compute unified device architecture), for solving of a SBTS, the total number of launched threads is $32 \times \text{nbr}$ where nbr is the number of block rows (columns) of the matrix. The total number of CUDA blocks is calculated by ceil $((32 \times \text{nbr})/\text{threads\_per\_block})$. By default, the one-dimensional block organization is employed, but when the number of blocks exceeds the maximum number of blocks along one-dimension provided by the CUDA architecture, the two-dimensional block arrangement is conducted.

To identify which warp corresponds to the desired column of NL, a global thread identifier tid is first transformed to the global warp identifier gwpid and relative warp identifier wpid in a thread block (lines 5 and 6). Then, the local index within a warp for a thread, denoted rpos, can be obtained by threadId$x.x$%warp_size. For a block size $\text{bsz}\,(\text{bsz} \le 5)$, the maximum number of complete blocks a warp can cover at a time is floor $(\text{warp\_size}/\text{bsz}^2)$, and the maximum number of active threads in a warp is estimated by $\text{range} = \text{floor}\,(\text{warp\_size}/\text{bsz}^2) \times \text{bsz}^2$. As a result, the following task of the kernel (from lines 10 to 42) is limited in range threads. As illustrated in Figure 3, the warp goes through the row indices (J_l) of the gwpid$^{\text{th}}$ block column of NL. For each row index irow, the first task is to extract the blocks at the J_l rows in the irow$^{\text{th}}$ block column of L. Since L is expressed in BCSR format, the irow$^{\text{th}}$ block column cannot be accessed directly and continuously. Moreover, a warp may not be able to cover all blocks in the column simultaneously, and the type of computation on the first (diagonal) block is different from that on the remaining blocks in the column. By considering these issues, the task of going through a column of L is divided into two subtasks.

First, the first bsz^2 threads solve a block of the solution by performing a block-block multiplication (lines 17–24). To exploit the fine-grained feature of the GPU, the computing of only one element for the result of multiplication is assigned to a thread. The thread corresponding to the r^{th} row and c^{th} column of the result is responsible for the inner product between the r^{th} row of the left-hand block and c^{th} column of the right-hand block. Because values in the two blocks are repeatedly visited, the multiplication is implemented through the shared memory, where the two operating blocks are loaded to avoid repeated access from the global memory. Note that the first block (diagonal block of a SBTM) is inversed by using the symbolic expression of the adjoint matrix of a block. For example, letting $B = \{b_{r,c}, r, c = 0, 1, 2, 3\}$ be a 4×4 block, then the r^{th} row and c^{th} column of the adjoint matrix of B, denoted as $B^*_{r,c}$, can be expressed explicitly as

$$B^*_{r,c} = (-1)^{r+c} \begin{vmatrix} b_{(c+1)\%4,(r+1)\%4} & b_{(c+1)\%4,(r+2)\%4} & b_{(c+1)\%4,(r+3)\%4} \\ b_{(c+2)\%4,(r+1)\%4} & b_{(c+2)\%4,(r+2)\%4} & b_{(c+2)\%4,(r+3)\%4} \\ b_{(c+3)\%4,(r+1)\%4} & b_{(c+3)\%4,(r+2)\%4} & b_{(c+3)\%4,(r+3)\%4} \end{vmatrix}, \tag{10}$$

and therefore, each thread of the bsz^2 threads is only computing one element of the adjoint matrix. The symbolic expressions for bsz = 3 and bsz = 5 can be derived in a similar way.

Second, the entire warp executes (9) by going over all remaining blocks in the column (lines 25–40). In the circumstance in which a warp is unable to cover all of the remaining blocks at one time, several cycles are performed until the end block is reached. In each cycle, before loading blocks into the available shared memory, every bsz^2 threads search myrow from right to left to identify whether there is a non-zero block at the irow$^{\text{th}}$ column. The block is set to a zero block if the searching fails. Once the searching is completed, up to floor $(\text{warp\_size}/\text{bsz}^2)$ blocks update the right-hand side by block-block multiplications.

The implementation for the preconditioning step comprises two block matrix-vector multiplications that can be realized by calling *cusparseDbsrmv* in the cuSPARSE library. Compared to performing triangular solves in a traditional way, the block matrix-vector multiplication offers more possible parallelism that may greatly reduce the computing overhead of the preconditioning step in each iteration.

4. Experiments

4.1. Experiment Setup. A computing node of the GPU cluster was used for all experiments. The node is configured with two Intel E5-2640 V4@2.40 GHz CPUs, with 128 GB of memory and four Nvidia Tesla V100 cards. The Tesla V100

card, with a compute capability of 7.5, is configured by 4096-bit HBM2 16 GB memory and has 80 stream multiprocessors (SMs) with 5120 FP32 cores, 2560 FP64 cores, and 640 Tensor cores in total. The peak single-precision and double-precision floating point performance is 15.7 and 7.8 TFLOPS, respectively.

The algorithms presented in this study are implemented based on the PETSc-3.14.2 (portable, extensible toolkit for scientific computation) framework [21] and CUDA 10.0. The GPU version of the GMRES algorithm with restart 30, developed for block matrices based on PETSc's data structure and cuSPARSE library, is used as the test solver.

For each test case, comparisons are made between different implementations for the preconditioning step. In the first implementation, the preconditioning step is realized using PETSc's block triangular solver on the CPU, and the results are copied to the GPU for the remaining part of computing by the GMRES. The second implementation uses parallel block triangular solves on the GPU from the cuSPARSE library. The input for the cuSPARSE's triangular solver is obtained by calling the ILU factorization routine in BCSR format. The last implementation is the block-ISAI preconditioning algorithm on the GPU as proposed in the present study. The lower and upper block triangular factors are provided by the block ILU factorization in PETSc and used as the input of Algorithm 1. For convenience, the wall clock time of computing the preconditioning step for 30 iterations is denoted as T_p, the total computing time until the GMRES converges to the relative tolerance is denoted as T_{total}, and the total number of GMRES iterations is denoted as G_{it}.

4.2. Atmosmodd Case. The first test matrix, atmosmodd, is from the SuiteSparse matrix collection [22]. It consists of 1,270,432 rows and columns and 8,814,880 non-zeros. The matrix is transformed into the BCSR format with block size 4 using cuSPARSE library [6]. As a result, the block matrix consists of 317,608 block rows and columns and 2,190,844 non-zero blocks. The relative tolerance for GMRES convergence is set to 10^{-8}.

Table 1 provides the time costs for 30 preconditioning steps and the GMRES (30) method, as well as the total number of GMRES iterations using different preconditioning methods. It is reported in [3] that the sparsity pattern for atmosmodd provides valid parallelism and make the level-scheduling algorithm on the GPU gain accelerations over serial triangular solvers on the CPU. It is observed from Table 1 that the level-scheduling scheme in block format is also more efficient than the serial block triangular solves implemented in PETSc, and approximately 12.4x speedup can be achieved. The preconditioning costs decrease significantly when block-ISAI ($|X|$) and block-ISAI ($|X|^2$) are applied, which is attributed to the high parallelism provided by the block sparse matrix-vector multiplication in the proposed algorithm. The resulting G_{it}, however, is more than that produced by exact triangular solves using PETSc and cuSPARSE. This is because we only compute an approximate inverse of the lower and upper block triangular matrices based on a guessing sparsity pattern.

Table 1: Atmosmodd case: T_p, T_{total}, and G_{it} for different preconditioning methods.

-	PETSc	cuSPARSE	Block-ISAI ($\|X\|$)	Block-ISAI ($\|X\|^2$)
T_p	1072 ms	86 ms	19.7 ms	44 ms
T_{total}	7654 ms	859 ms	594 ms	685 ms
G_{it}	201	201	295	242

The overhead consisting of three parts for setting up block-ISAI($|X|$) and block-ISAI ($|X|^2$) is shown in Figure 4. As described in Algorithm 1, NL and NU are solved in columns, and thus, the BCSR to BCSC transformations are required for NL and NU after the estimations of sparsity patterns have been made. Once the block values in NL and NU are filled by the GPU kernel, they are transformed into BCSR formats for the block matrix-vector multiplications in the preconditioning step. Therefore, four format transformations in total have to be conducted. Compared to using block-ISAI ($|X|$), using block-ISAI ($|X|^2$) results in a lower number of GMRES iterations but more computing time for the GMRES algorithm and constructing an effective preconditioner. Although the overhead of generating block-ISAI increases with making the sparsity pattern denser, it only costs 2.5% and 5.8% compared to T_{total} for block-ISAI ($|X|$) and block-ISAI ($|X|^2$), respectively.

By adding the overhead of setting up block-ISAI, the overall times for the above four methods, as well as the speedups over the baseline GPU GMRES (30) with PETSc's preconditioning are shown in Figure 5. A speedup of 12.6x is obtained by block-ISAI ($|X|$) over the baseline GMRES (30), which is the best among the listed methods. Compared to cuSPARSE, both versions of block-ISAI are faster.

4.3. af_shell3 Case. The second matrix, af shell3, is also from the SuiteSparse matrix collection [22]. The size of the matrix is $504,855 \times 504,855$ with 17,588,875 non-zeros. By setting the block size to 5, the matrix is transformed into the block format. The number of block rows and columns is 100,971 and that of non-zero blocks is 703,555. The relative tolerance for GMRES (30) is set to 10^{-8}.

In this case, $S(|X|)$, $S(|X|^2)$, and $S(|X|^3)$ are considered for the estimation the inverses of L and U. Table 2 provides the comparisons of T_p, T_{total}, and G_{it} obtained by the five methods. Note that the PETSc, cuSPARSE, and block-ISAI ($|X|^3$) methods result in less than 30 GMRES iterations, so the corresponding T_p values are reported using the actual number of iterations; otherwise, T_p for 30 iterations is reported. It is observed that the exact triangular solver from cuSPARSE is efficient in extracting possible parallelism from the sparsity pattern, and it runs approximately 2.55x faster than the PETSc's serial block triangular solver. However, the preconditioning overhead of cuSPARSE, T_p, still accounts for 88% of the total computing time. In contrast, the block-ISAI method results in much less time for T_p, but at the sacrifice of increasing the total number of GMRES iterations. In addition, G_{it} decreases when the estimated patterns for inverses become denser.

By counting the overhead of setting up block-ISAI in Figure 6, the overall time and speedups are shown in

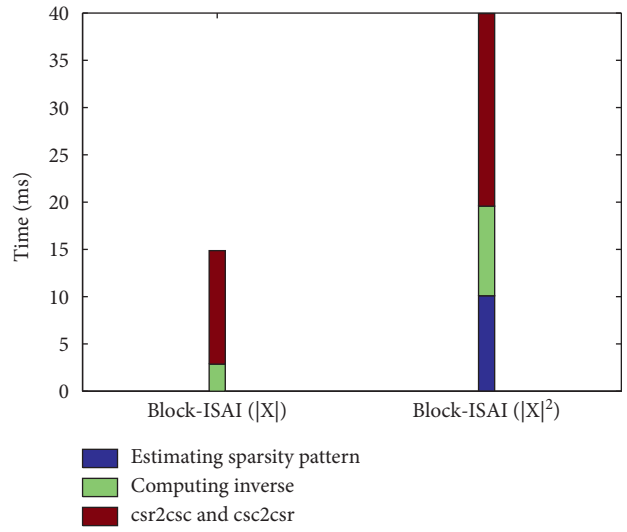

Figure 4: Atmosmodd case: breakdowns of costs for setting up block-ISAI $|X|$ and block-ISAI $|X|^2$.

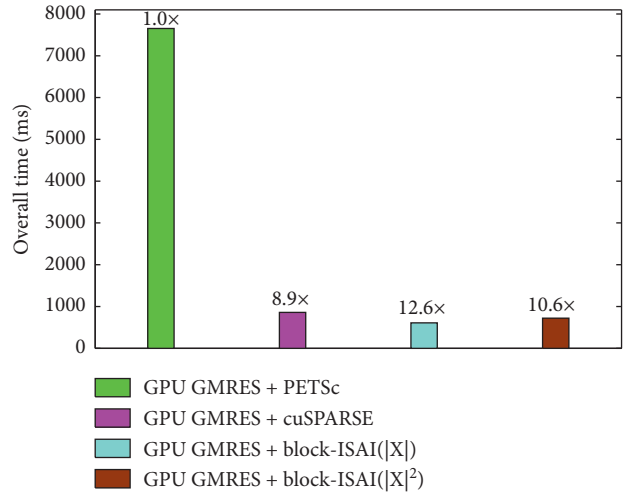

Figure 5: Atmosmodd case: comparisons of overall times between different preconditioning methods.

Table 2: af_shell3 case: T_p, T_{total}, and G_{it} for different preconditioning methods.

-	PETSc	cuSPARSE	Block-ISAI ($\|X\|$)	Block-ISAI ($\|X\|^2$)	Block-ISAI ($\|X\|^3$)
T_p	435 ms	170 ms	9.03 ms	15.1 ms	22.0 ms
T_{total}	470 ms	193 ms	40.1 ms	35.5 ms	39.7 ms
G_{it}	25	25	45	29	26

Figure 7. Although the costs of block-ISAI are comparable to T_{total}, the overall times obtained by block-ISAI methods are still much less than those in the PETSc's and cuSPARSE implementations. Figure 7 indicates that all three block-ISAI with different sparsity patterns can achieve more than 5x over PETSc's implementation. Block-ISAI with $S(|X|)$ is the most efficient in terms of the overall time and is nearly 3.9x faster than the cuSPARSE implementation.

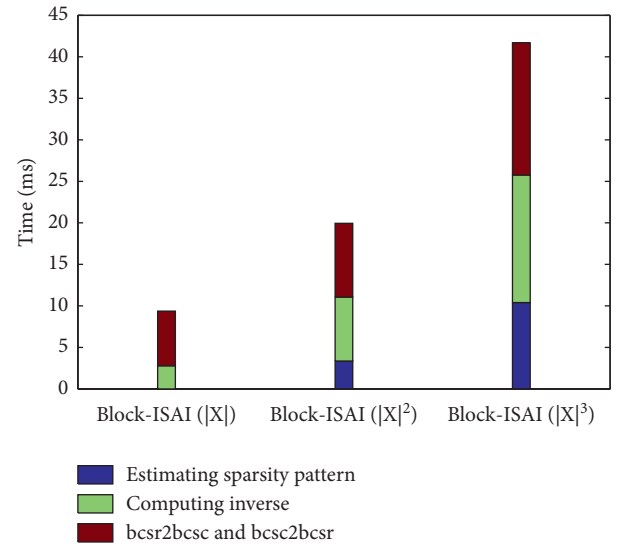

Figure 6: af shell3 case: breakdowns of costs for setting up block-ISAI ($|X|$), block-ISAI ($|X|^2$), and block-ISAI ($|X|^3$).

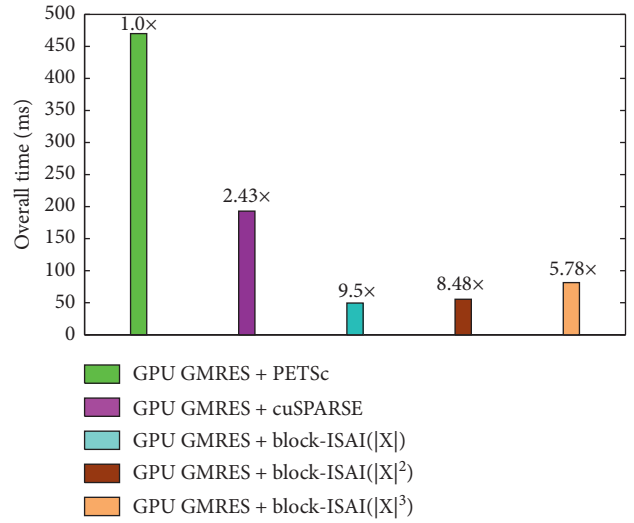

Figure 7: af_shell3 case: comparisons of overall times between different preconditioning methods.

4.4. Lid-Driven Cavity Case. In the third experiment, a linear system with the block matrix from a lid-driven cavity case is solved on a 300×300 mesh in the velocity-vorticity formulation using the five-point finite difference method. The block matrix, with a block size of 3, is extracted from the first Newton step of the nonlinear solver. The number of block rows and columns is 90,000 and that of non-zero blocks is 448,800. The relative tolerance is set to 10^{-5} for GMRES (30).

It is observed from Table 3 that cuSPARSE on the Tesla V100 fails to accelerate the preconditioning step that is composed of two block triangular systems. This is due to the strong data dependency in the sparsity pattern. Block-ISAI, in contrast, can offer a better solution in this circumstance. Eventhough the number of GMRES iterations using block-ISAI is 14%, 34%, and 75% more than that using exact

Table 3: Lid-driven cavity case: T_p, T_{total}, and G_{it} for different preconditioning methods.

-	PETSc	cuSPARSE	Block-ISAI ($\|X\|$)	Block-ISAI ($\|X\|^2$)	Block-ISAI ($\|X\|^3$)
T_p	130 ms	129 ms	3.72 ms	5.20 ms	6.88 ms
T_{total}	3150 ms	3074 ms	560 ms	468 ms	443 ms
G_{it}	623	623	1092	836	711

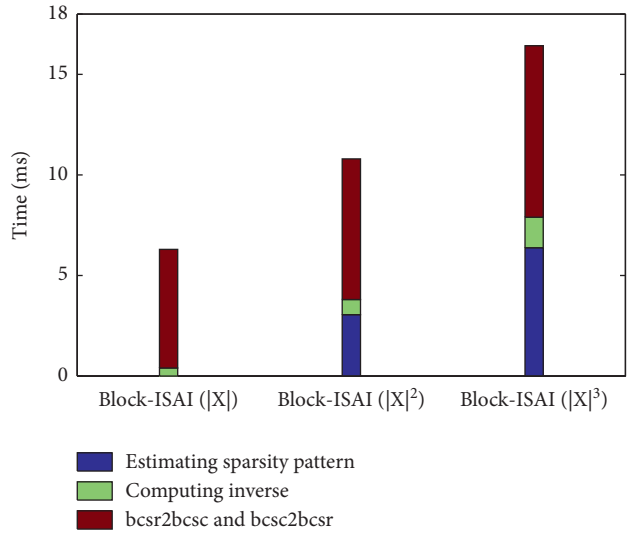

Figure 8: Lid-driven cavity case: breakdowns of costs for setting up block-ISAI ($|X|$), block-ISAI ($|X|^2$), and block-ISAI ($|X|^3$).

triangular solves, block-ISAI preconditioning is still more efficient in terms of T_{total}. In Figure 8, the breakdown costs of block-ISAI with different sparsity patterns indicates that compared to T_{total}, the overhead is only up to 4%, which keeps the cost from affecting the overall time.

Figure 9 shows the comparisons of overall times obtained by PETSc, cuSPARSE, and block-ISAI. Block-ISAI preconditioning shows an obvious advantage over the serial and parallel preconditioning from PETSc and cuSPARSE. It is further observed that both T_{total} and G_{it} obtained by block-ISAI preconditioning decreases upon increasing the density level of the sparsity pattern. GMRES (30) using block-ISAI ($|X|^3$) preconditioning is approximately 6.9x faster than that using PETSc or cuSPARSE preconditioning.

4.5. Venkat50 Case. The last matrix is from the unstructured two-dimensional Euler solver provided by Venkatakrishnan at the SuiteSparse matrix collection [22]. The fluid fields associated with mesh points are ordered continuously so that the resulting matrix has a block structure naturally, with a block size of 4, which can be viewed in Figure 10. The matrix has 15,606 block rows and columns with 107,362 non-zero blocks in total. The relative tolerance is 10^{-8} for the GMRES (30) solver.

It is given in Table 4 that the preconditioning (triangular solver) using cuSPARSE is nearly 4 times slower than the serial ones provided by PETSc. This is normal because the level scheduling-based scheme cannot always be efficient on

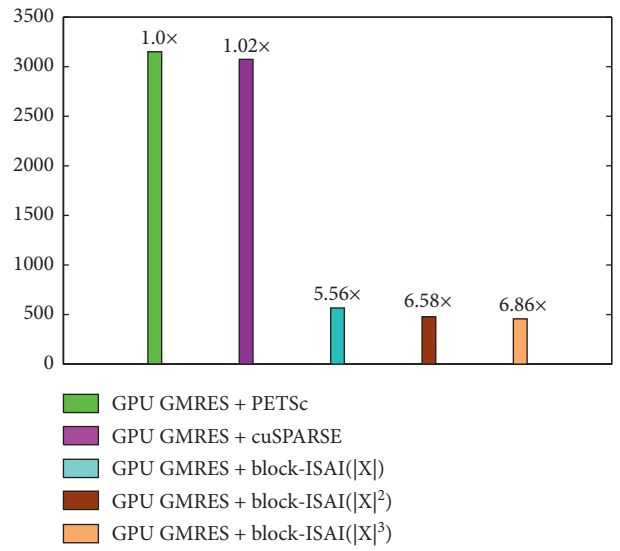

Figure 9: Lid-driven cavity case: comparisons of overall times between different preconditioning methods.

the GPU when slight parallelism is extracted from a given sparsity pattern [3]. Compared to T_{total} obtained by PETSc, T_{total} using block-ISAI ($|X|$) does not show an obvious advantage because the approximate inverses lack of accuracy and result in 6 times more GMRES iterations in G_{it}. Upon increasing the density of the sparsity pattern, the decrease of both G_{it} and T_{total} is observed, and block-ISAI ($|X|^3$) is

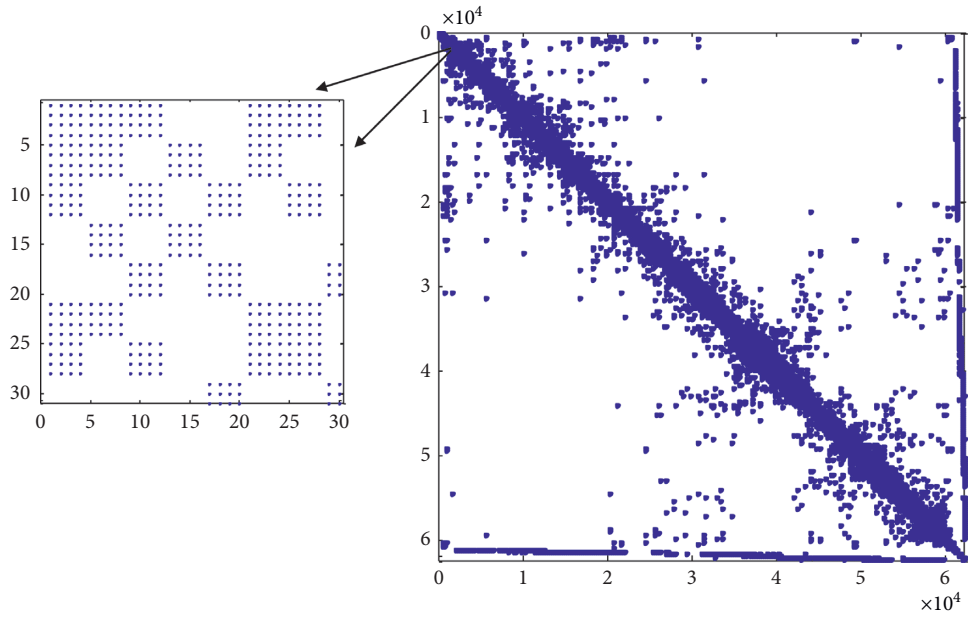

Figure 10: Sparsity pattern for venkat50 matrix with natural block structure.

Table 4: Venkat50 case: T_p, T_{total}, and G_{it} for different preconditioning methods.

-	PETSc	cuSPARSE	Block-ISAI ($\|X\|$)	Block-ISAI ($\|X\|^2$)	Block-ISAI ($\|X\|^3$)
T_p	50 ms	192 ms	1.96 ms	3.02 ms	4.25 ms
T_{total}	734 ms	2556 ms	678 ms	458 ms	369 ms
G_{it}	367	367	2208	1324	948

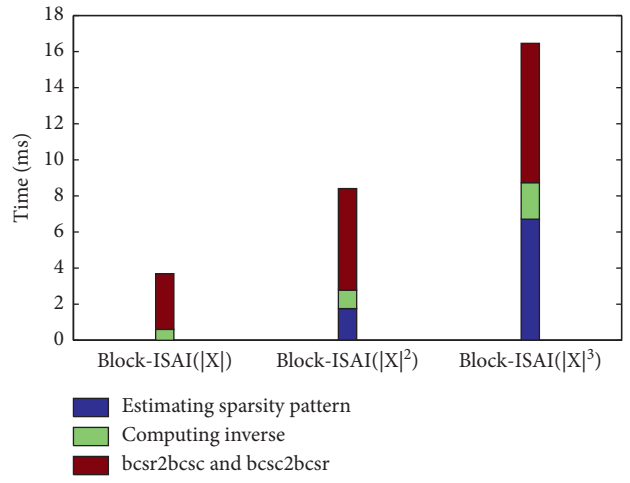

Figure 11: Venkat50 case: breakdowns of costs for setting up block-ISAI ($|X|$), block-ISAI ($|X|^2$), and block-ISAI ($|X|^3$).

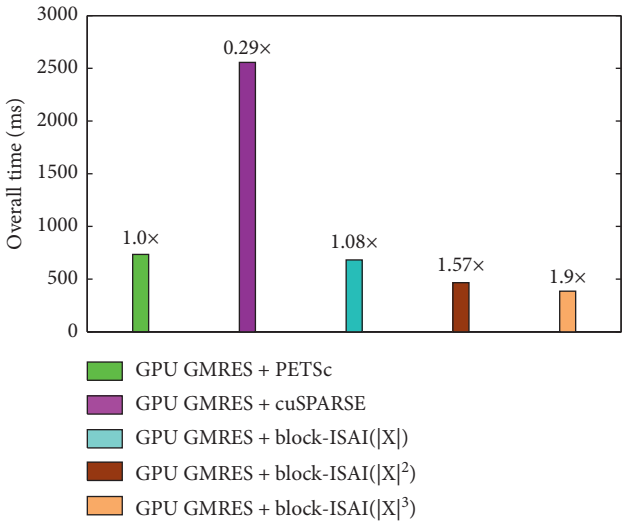

Figure 12: Venkat50 case: comparisons of overall times between different preconditioning methods.

shown to be the best preconditioner in term of T_{total}. The breakdown of costs for computing block-ISAI upon increasing the dense levels is shown in Figure 11. The overall speedups on GMRES are calculated and shown in Figure 12 by counting the overhead for computing block-ISAI, and speed improvements of approximately 1.92x and 6.69x are achieved compared to PETSc's and cuSPARSE's preconditioning, respectively.

5. Conclusions

In this study, block-ISAI preconditioning for block sparse matrices is investigated by proposing an efficient, warp-based algorithm to approximate the inverses of block triangular factors on a Tesla V100 GPU. Instead of solving triangular systems globally for preconditioning, a group of

small triangular systems with very high concurrency is solved to approximate the inverses and conduct block SpMV in the preconditioning step. The results on four cases from a matrix collection website and multiphysics areas show that the proposed GPU algorithm outperforms the serial and parallel block triangular solver-based preconditioning from the PETSc and cuSPARSE libraries in terms of the total computing time of the GMRES (30) algorithm. It is noted that all comparisons are performed under the condition that the L and U factors are provided because the target for comparison is the block triangular solves in the preconditioning step. Planned future work includes extending the algorithm to multiple GPUs that are connected by the NVLink technology using block-Jacobi or the additive Schwarz method (ASM) preconditioners, as well as the coupling of sparsity patterns and relaxation steps [10] for extreme cases.

Data Availability

The main matrix data used to support the findings of this study are available from the SuiteSparse matrix collection (https://sparse.tamu.edu/).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was supported by a grant from the National Key R&D Program of China (2019YFB1704202), the National Natural Science Foundation of China (61702438), and the Nanhu Scholar Program of XYNU and the Innovation Team Support Plan of University Science and Technology of Henan Province (19IRTSTHN014). The authors thank LetPub for its linguistic assistance during the preparation of this manuscript.

References

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

[2] M. Naumov, *Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU*, NVIDIA Technical Report NVR-2011-001, NVIDIA Corporation, Santa Clara, CF, USA, 2011.

[3] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.

[4] H. Kabir, J. D. Booth, G. Aupy, A. Benoit, Y. Robert, and P. Raghavan, "STS-K: a multilevel sparse triangular solution scheme for NUMA multicores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin, TX, USA, November 2015.

[5] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying synchronization for high-performance shared-memory sparse triangular solver," in *Proceedings of International Supercomputing Conference*, Springer International Publishing, Leipzig, Germany, June 2014.

[6] NVIDIA cuSPARSE, https://docs.nvidia.com/cuda/archive/10.0/cusparse/index.html. (accessed on 30 December 2020).

[7] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A synchronization-free algorithm for parallel sparse triangular solves," *Euro-Par 2016: Parallel Processing*, vol. 9833, pp. 617–630, 2016.

[8] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter, "Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides," *Concurrency and Computation: Practice and Experience*, vol. e4244, pp. 1–18, 2017.

[9] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *Proceedings of European Conference on Parallel Processing*, Springer, Berlin, Heidelberg, August 2015.

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

[11] H. Anzt, E. Chow, T. Huckle, and J. Dongarra, "Batched generation of incomplete sparse approximate inverses on GPUs," in *Proceedings of Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, IEEE, Salt lake, UT, USA, November 2016.

[12] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015.

[13] E. Chow, H. Anzt, and J. Dongarra, "Asynchronous iterative algorithm for computing incomplete factorizations on GPUs," *Lecture Notes in Computer Science*, vol. 9137, pp. 1–16, 2015.

[14] Intel Many Integrated Core Architecture (Intel MIC Architecture)-Advanced. https://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html. (accessed on 31 December 2020).

[15] 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.

[16] A. C. N. V. Duin, "Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 4, 1999.

[17] D. Bertaccini and S. Filippone, "Sparse approximate inverse preconditioners on high performance GPU platforms," *Computers & Mathematics with Applications*, vol. 71, no. 3, pp. 693–711, 2016.

[18] G. He, R. Yin, and J. Gao, "An efficient sparse approximate inverse preconditioning algorithm on GPU," *Concurrency and Computation Practice and Experience*, vol. 32, no. 3, 2019.

[19] J. Gao, Q. Chen, and G. He, "A thread-adaptive sparse approximate inverse preconditioning algorithm on multi-GPUs," *Parallel Computing*, vol. 101, no. 3, Article ID 102724, 2021.

[20] R. Eberhardt and M. Hoemmen, "Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures," in *Proceedings of 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, Chicago, IL, USA, May 2016.

[21] Balay S.; Abhyankar S.; Adams M. F.; et al. PETSc Web Page. https://www.mcs.anl.gov/petsc.(accessed on 30 December 2020).

[22] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.

[23] Y. Saad, M. H. Schultz, and M. H. Schultz, "GMRES: a generalized minimal residual algorithm for solving non-symmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, 1986.

[24] B. Yang and H. Liu, "Accelerating the GMRES solver with block ILU(k) preconditioner on GPUs in reservoir simulation," *Journal of Geology and Geophysics*, vol. 4, no. 2, pp. 1–7, 2015.

[25] A. Kashi and S. Nadarajah, *Fine-grain Parallel Smoothing by Asynchronous Iterations and Incomplete Sparse Approximate Inverses for Computational Fluid Dynamics*, AIAA, Orlando, FL, USA, 2020.