IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2023.0322000

Mixed-Precision Sparse Approximate Inverse Preconditioning Algorithm on GPU

XINYUE CHU
School of Computer and Electronic Information, Nanjing Normal University, Nanjing 210023, China
Corresponding author: Xinyue Chu (e-mail: 2316607219@qq.com).

ABSTRACT In this study, in order to further improve the construction efficiency of sparse approximate inverse (SPAI) preconditioners, we attempt to explore the construction method of SPAI preconditioners in mixed-precision mode from the perspective of single and double precision mixing, and thus propose two mixed-precision SPAI preconditioning algorithms on GPU, abbreviated as MP-SSPAI and MP-HeuriSPAI, respectively. In MP-SSPAI, with original static SPAI preconditioning algorithm as the research object, we mainly consider the following factors to construct its preconditioner in mixed-precision mode: 1) use single precision as much as possible to improve computational efficiency of the preconditioner while ensuring its validity; 2) store certain components in single precision after they have been determined to require single-precision computation to improve read efficiency; and 3) maintain the high-precision output of the preconditioner to ensure that it is computed with high precision when applied to the iterative algorithm. In MP-HeuriSPAI, a mixed-precision heuristic dynamic SPAI preconditioning algorithm on GPU is presented based on the above factors, using HeuriSPAI as the object of study. The experimental results demonstrate the effectiveness and high performance of the proposed MP-SSPAI and MP-HeuriSPAI by comparing them with their respective double-precision versions, single-precision versions, and extended versions.

INDEX TERMS GPU, mixed precision, preconditioning algorithm, sparse approximate inverse.

I. INTRODUCTION

IN general, the large sparse linear systems can be interpreted as follows:

$$Ax = b, \quad x, b \in \mathbb{R}^n, A \in \mathbb{R}^{n \times n}. \tag{1}$$

Here coefficient matrix A is large, sparse, and nonsingular, and x and b are given vector and unknown one, respectively. To address above problem better, preconditioning Krylov iterative methods come into view, which can accelerate convergence and have higher robustness compared with Krylov iterative methods. Using preconditioning techniques, equation (1) can be further transformed into a more tractable form as:

$$MAx = Mb \ \text{ or } \ AMy = b, x = My. \tag{2}$$

Here M is referred to as left (right) preconditioner. A better preconditioner M should satisfy the following three conditions:

1) its operation should be simple and cheap.

2) it is supposed to accelerate convergence of iterative methods.

3) it is effectively computed in parallel.

However, the construction of preconditioners is time-consuming, leading to a significant increase of time cost of seeking the approximate solution ($\hat{x}$). Programmable graphics processing units(GPUs) have the feature of multiple core structures, which makes them powerful for scientific computing and big data processing. And due to easiness of learning and using, and needless of graphics knowledge for developers, the compute united device architecture(CUDA) [1] introduced by NVIDIA is much popular, which supports joint CPU/GPU execution of applications and designs a C-based programming language CUDA C for GPU computing. Therefore, it is utilized in much work [2]–[4] to accelerate the construction of preconditioners.

At present, there are various preconditioners, such as Jacobi preconditioner [5], [6], block−Jacobi preconditioner [7], [8], factorized sparse approximate inverse preconditioner [9]–[11], polynomial preconditioner [12]–[14], incomplete LU decompositions [15]–[17], and sparse approximate inverse (SPAI) preconditioner based on F-norm minimization [18]–[22]. Because of high parallelism and simplicity, the SPAI preconditioner has received widespread attention. And according to the construction method, it is usually classified into static SPAI preconditioning algorithm [23]–[28] and dynamic SPAI preconditioning algorithm [29]–[33].

In addition, with the advancement of technology, GPUs un-

der the CUDA architecture not only support double-precision floating-point operations but also single-precision floating-point operations and even half-precision floating-point operations. Theoretically, single-precision floating-point operations are twice as fast as double-precision floating-point operations and require relatively less memory. As a result, mixed-precision floating-point computations based on single and double precision have been used in multiple research areas [34]–[37]. Inspired by this, in order to further improve the computational efficiency of preconditioning algorithms without losing their effectiveness, some researchers have attempted to construct preconditioners in mixed-precision mode [4], [38]–[43]. However, research on mixed-precision SPAI preconditioning algorithms is scarce.

Therefore, on the basis of precision consideration, we present two mixed-precision SPAI preconditioning algorithms on GPU, abbreviated as MP-SSPAI and MP-HeuriSPAI, respectively. For the construction of the mixed-precision SPAI preconditioner, the following factors are considered: 1) use single precision as much as possible to improve computational efficiency of the preconditioner while ensuring its validity; 2) store certain components in single precision after they have been determined to require single precision computation to improve read efficiency; and 3) maintain the high-precision output of the preconditioner to ensure that it is computed with high precision when applied to the iterative algorithm.

The main contributions in this work are summarized as follows.

- Mixed-precision static SPAI preconditioning algorithm and mixed-precision heuristic SPAI preconditioning algorithm are presented;
- The parallel versions of proposed two mixed-precision SPAI preconditioning algorithms, abbreviated as MP-SSPAI and MP-HeuriSPAI, respectively, are implemented;
- The extended versions of MP-SSPAI and MP-HeuriSPAI are given, abbreviated as MP1-SSPAI and MP1-HeuriSPAI, respectively. Then, a series of experiments demonstrate the effectiveness and high performance of the proposed MP-SSPAI and MP-HeuriSPAI by comparing them with their respective double-precision versions, single-precision versions, and extended versions.

The rest of the paper is organized as follows. In Section 2, sparse approximate inverse (SPAI) preconditioner based on F-norm minimization is summarized. Mixed-precision static SPAI preconditioning algorithm and mixed-precision heuristic SPAI preconditioning algorithm are presented in Section 3. And their parallel implementations on GPU are given in Section 4. Section 5 gives effectiveness analysis and performance evaluation. Finally, Section 6 concludes conclusions and discussions.

II. SPARSE APPROXIMATE INVERSE (SPAI) PRECONDITIONER BASED ON F-NORM MINIMIZATION

The preconditioner M of SPAI preconditioning algorithm is the approximation of A^{-1}. For static SPAI preconditioning algorithm, the sparse pattern of preconditioner M is predetermined, which generally consistents with the sparse pattern of coefficient matrix A or identity matrix E. As shown in [24], preconditioner M is computed by following equation:

$$\min \|AM - \mathcal{I}\|_F^2, \ \ \mathcal{I} \in \mathbb{R}^{n\times n}. \tag{3}$$

Here for M, its columns are independent with each other, thus, equation (3) can be expressed as the following equation:

$$\min \sum_{k=1}^{n} \| Am_k - e_k \|_2^2 = \sum_{k=1}^{n} \min \| Am_k - e_k \|_2^2, \tag{4}$$

where m_k and e_k represent the kth column of preconditioner M and identity matrix E, respectively. Obviously, it can be further decoupled as n least squares problems:

$$\min \|Am_k - e_k\|_2^2, \ \ k = 1, 2, \ldots, n. \tag{5}$$

Observing that, for smaller n, all columns of the preconditioner M can be solved concurrently. This indicates that the SPAI preconditioning algorithm has high degree of parallelism.

In order to solve the preconditioner M easily, its each column will be computed by dimensionality reduction. Taking the kth column of M (m_k) as an example, first, find its row indices of nonzero entries of m_k and save them in set J_k. Second, delete zero rows in matrix $A(., J_k)$ and save its indices of nonzero rows in set I_k, then we can obtain the submatrix $\hat{A}_k$, where $\hat{A}_k = A(I_k, J_k)$. Based on this, equation (5) can be transformed into the following equation:

$$\min \| \hat{A}_k \hat{m}_k - \hat{e}_k \|_2^2, \ \ k = 1, 2, \ldots, n, \tag{6}$$

where $\hat{m}_k$ and $\hat{e}_k$ are the reduced m_k and e_k, respectively. Third, perform QR decomposition on matrix $\hat{A}_k$ with the modified Gram-Schmidt method. Finally, solve the above equation.

The detailed procedure of static SPAI preconditioning algorithm based on double precision (SSPAI for short) is shown as following:

Algorithm 1: Static SPAI preconditioning algorithm (SSPAI)
For each column $m_k, k = 1, 2, \cdots, n$ of M:

1) Set $J_k = \{j | m_k(j) \neq 0\}$, and set its length as $n2$;
2) Construct I_k, where its any element(i) makes $A(i, J_k)$ not all 0, and set its length as $n1$;
3) Construct submatrix $\hat{A}_k$ where $\hat{A}_k = A(I_k, J_k)$ and $\hat{A}_k \in \mathbb{R}^{n1\times n2}$; (double precision)
4) Perform QR decomposition on matrix $\hat{A}_k$, then, the orthogonal matrix $Q_k \in \mathbb{R}^{n1\times n2}$ and the upper triangular matrix $R_k \in \mathbb{R}^{n2\times n2}$ are obtained; (double precision)
5) Set $\hat{A}_k = Q_k R_k$, and then solve $\hat{m}_k$ by (6); (double precision)
6) Scatter $\hat{m}_k$ to m_k; (double precision)

For dynamic SPAI preconditioning algorithm, its sparse pattern of preconditioner M is acquired dynamically without a

pre-given. Taking HeuriSPAI [33] as an example, first, solve initial m_k according to Algorithm 1, and then compute initial residual $r_k = e_k - Am_k$. Second, it uses

$$C_k^l = (E + |A|)C_k^{l-1}, \quad l = 1, 2, \cdots, l_{\max} \tag{7}$$

to iteratively generate the candidate indices that might be added to J_k^{l-1}, where l is the internal loop variable, $l_{\max}$ is the maximum iterative number of the heuristic computation, E is identity matrix, and J_k^{l-1} represents the sparse pattern of the kth column of the preconditioner M at the $l-1$st iteration. C_k^0 is equal to initial sparse pattern of the kth column of the preconditioner M (J_k^0). Third, save the indices that appear in C_k^l but not in J_k^{l-1} into set $\tilde{J}_k^l$. Fourth, to avoid excessive computation, the elements in $\tilde{J}_k^l$ need to be reduced. In detail, for each candidate index j ($j \in \tilde{J}_k^l$), consider the following one-dimensional minimization problem:

$$\min_{\mu_j \in R} \|r_k + \mu_j A e_j\| =: \rho_j. \tag{8}$$

Then, ρ_j^2 can be presented by

$$\rho_j^2 = \|r_k\|_2^2 - \left(\frac{r_k^T A e_j}{\|Ae_j\|_2}\right)^2. \tag{9}$$

For each $j \in \tilde{J}_k^l$, if its corresponding ρ_j is smaller, then it will be considered the most profitable index and retained, otherwise it will be deleted. Fifth, utilize the deleted set $\tilde{J}_k^l$, the new row indices set $\tilde{I}_k^l$ is determined, and then execute the QR decomposition of the new submatrix $A(I_k^{l-1} \cup \tilde{I}_k^l, J_k^{l-1} \cup \tilde{J}_k^l)$. Finally, compute new m_k ($m_k(J_k^{l-1} \cup \tilde{J}_k^l)$), r_k, and $||r_k||_2$. If r_k satisfies the loop-stopping condition, the algorithm stops; otherwise, set $l = l + 1$ and then the loop continues. Furthermore, to maintain the sparsity of preconditioner, it sets the filling threshold for each column of M (u_k) by the following equation

$$u_k = \alpha \cdot x_k, \tag{10}$$

where α is a small real number and x_k is the nonzero number of the kth column of A. Algorithm 2 shows its detailed procedure of Heuristic SPAI preconditioning algorithm based on double precision (HeuriSPAI for short), where $|J_k^{l-1}|$ denotes the length of set J_k^{l-1}.

Algorithm 2: Heuristic SPAI preconditioning algorithm (HeuriSPAI)
For every column $m_k, k = 1, 2, \cdots, n$ of M:

1) Choose an initial sparsity $J_k^0 = \{k\}$, set $l = 1$, $C_k^0 = J_k^0$, a suitable tolerance ε, $l_{\max}$, and compute u_k by (10);
2) Solve initial m_k by **Algorithm** 1 and compute r_k with double precision;
   While $\|r_k\|_2 > \varepsilon$ and $l < l_{\max}$ and $|J_k^{l-1}| < u_k$:
3) Compute C_k^l by (7);
4) Save the indices that belong to C_k^l but not in J_k^{l-1} into set $\tilde{J}_k^l$;
5) For every $j \in \tilde{J}_k^l$, compute ρ_j^2 by (9), and delete from $\tilde{J}_k^l$ all but the most profitable indices; (double precision)
6) Determine the new row indices $\tilde{I}_k^l$ and then execute the QR decomposition of the new submatrix $A(I_k^{l-1} \cup \tilde{I}_k^l, J_k^{l-1} \cup \tilde{J}_k^l)$; (double precision)
7) Compute new m_k, r_k, and $||r_k||_2$, then set $J_k^l = J_k^{l-1} \cup \tilde{J}_k^l$, $I_k^l = I_k^{l-1} \cup \tilde{I}_k^l$, $C_k^l = J_k^l$, and $l = l + 1$; (double precision)

III. MIXED-PRECISION SPARSE APPROXIMATE INVERSE PRECONDITIONING ALGORITHM

A. MIXED-PRECISION STATIC SPARSE APPROXIMATE INVERSE PRECONDITIONING ALGORITHM

First, with original double-precision static SPAI preconditioning algorithm shown in Algorithm 1 as the research object, we describe the detailed procedure of the mixed-precision static SPAI preconditioning algorithm (see Algorithm 3). Analyzing Algorithm 3, when constructing the submatrix in the third step, it only involves the assignment of values and does not require inter-valued calculations, thus, single-precision floating-point calculations are used to improve the read efficiency. In the fifth step, the single-precision floating-point computation is still used due to the complexity and time-consuming of QR decomposition. In the sixth and seventh steps, the double-precision floating-point calculation is used to keep the output of the preconditioner with high accuracy, so that when it is applied to the iterative algorithm, the high precision computation is maintained and the accuracy of the solution is not lost.

Algorithm 3: Mixed-precision static SPAI preconditioning algorithm
For every column $m_k, k = 1, 2, \cdots, n$ of M:

1) Set $J_k = \{j | m_k(j) \neq 0\}$, and set its length as $n2$;
2) Construct I_k, where its any element(i) makes $A(i, J_k)$ not all 0, and set its length as $n1$;
3) Construct submatrix $\hat{A}_k$ where $\hat{A}_k = A(I_k, J_k)$ and $\hat{A}_k \in \mathbb{R}^{n1 \times n2}$; (single precision)
4) Perform QR decomposition on matrix $\hat{A}_k$, then, the orthogonal matrix $Q_k \in \mathbb{R}^{n1 \times n2}$ and the upper triangular matrix $R_k \in \mathbb{R}^{n2 \times n2}$ are obtained; (single precision)
5) Set $\hat{A}_k = Q_k R_k$, and then solve $\hat{m}_k$ by (6); (double precision)
6) Scatter $\hat{m}_k$ to m_k; (double precision)

Then, observing that, in Algorithm 3, coefficient matrix A requires single-precision input, while the double-precision coefficient matrix will still be used in iterative solving stage to ensure the robustness and convergence of the iterative algorithm. Therefore, the double-precision coefficient matrix A needs to be converted to a single-precision one on GPU and stored in the array $A1$ before constructing the preconditioner. In addition, the conversion from high precision to low precision may result in numerical overflow, thus, to avoid the situation, we set those numerical overflow values uniformly to half of the maximum value that can be represented by single precision.

In summary, the complete procedure of mixed-precision static SPAI preconditioner applied to the Krylov iterative

algorithm for solving linear systems in (1) will be given below.

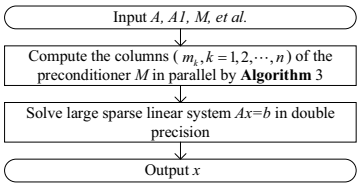

FIGURE 1. Main procedure of Krylov iterative algorithm with mixed-precision static SPAI preconditioner

Finally, based on proposed mixed-precision static SPAI preconditioning algorithm (see Algorithm 3), we give its extended version shown in Algorithm 4 to confirm its high performance. Compare to Algorithm 3, in Algorithm 4, the QR decomposition is performed in double precision, which improves orthogonality but increases time cost. Moreover, it employs single-precision computation in solving m_k, thereby reducing the effectiveness of the preconditioner M.

Algorithm 4: The extended version of mixed-precision static SPAI preconditioning algorithm
For every column $m_k, k = 1, 2, \cdots, n$ of M:

1) Set $J_k = \{j | m_k(j) \neq 0\}$, and set its length as $n2$;
2) Construct I_k, where its any element(i) makes $A(i, J_k)$ not all 0, and set its length as $n1$;
3) Construct submatrix $\hat{A}_k$ where $\hat{A}_k = A(I_k, J_k)$ and $\hat{A}_k \in \mathbb{R}^{n1\times n2}$; (single precision)
4) Perform QR decomposition on matrix $\hat{A}_k$, then, the orthogonal matrix $Q_k \in \mathbb{R}^{n1\times n2}$ and the upper triangular matrix $R_k \in \mathbb{R}^{n2\times n2}$ are obtained; (double precision)
5) Set $\hat{A}_k = Q_k R_k$, and then solve $\hat{m}_k$ by (6); (single precision)
6) Scatter $\hat{m}_k$ to m_k; (single precision)

B. MIXED-PRECISION HEURISTIC SPARSE APPROXIMATE INVERSE PRECONDITIONING ALGORITHM

First, with original double-precision heuristic SPAI preconditioning algorithm shown in Algorithm 2 as the research object, we give the computational procedure of the mixed-precision heuristic sparse approximate inverse preconditioning algorithm, which is given below:

Algorithm 5: Mixed-precision heuristic sparse approximate inverse preconditioning algorithm
For every column $m_k, k = 1, 2, \cdots, n$ of M:

1) Choose an initial sparsity $J_k^0 = \{k\}$, set $l = 1$, $C_k^0 = J_k^0$, a suitable tolerance ε, $l_{\max}$, and compute u_k by (10);
2) Solve initial m_k using **Algorithm** 3 and compute r_k with double precision;
While $\|r_k\|_2 > \varepsilon$ and $l < l_{\max}$ and $|J_k^{l-1}| < u_k$:
3) Compute C_k^l by (7);
4) Save the indices that belong to C_k^l but not in J_k^{l-1} into set $\tilde{J}_k^l$;
5) For every $j \in \tilde{J}_k^l$, compute ρ_j^2 by (9), and delete from $\tilde{J}_k^l$ all but the most profitable indices; (single precision)
6) Determine the new row indices $\tilde{I}_k^l$ and then execute the QR decomposition of the new submatrix $A(I_k^{l-1} \cup \tilde{I}_k^l, J_k^{l-1} \cup \tilde{J}_k^l)$; (single precision)
7) Compute new m_k, r_k, and $||r_k||_2$, then set $J_k^l = J_k^{l-1} \cup \tilde{J}_k^l$, $I_k^l = I_k^{l-1} \cup \tilde{I}_k^l$, $C_k^l = J_k^l$, and $l = l + 1$; (double precision)

Then, observing Algorithm 5, in the initial stage, it computes initial m_k, $k = 1, 2, \cdots, n$ with Algorithm 3, and utilizes double precision to compute r_k and $||r_k||_2$. In the loop finding filling indices stage, it is experimentally found that for different j, their corresponding ρ values are generally different, so that single-precision computing does not affect the final choice of the potential filling indices. In addition, as in Algorithm 3, single precision is used in step 6, while double precision is used in step 7.

In summary, the complete procedure of mixed-precision heuristic SPAI preconditioner applied to the Krylov iterative algorithm for solving linear systems is likewise given below:

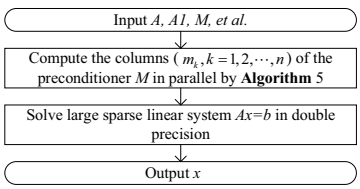

FIGURE 2. Main procedure of Krylov iterative algorithm with mixed-precision heuristic SPAI preconditioner

Finally, in order to prove the high performance of proposed mixed-precision heuristic sparse approximate inverse preconditioning algorithm (see Algorithm 5), we also give its extended version shown in Algorithm 6. Different from Algorithm 5, in Algorithm 6, the extended version of mixed-precision static SPAI preconditioning algorithm (see Algorithm 4) is used to solve initial m_k. The QR decomposition is performed in double precision, which improves orthogonality but increases time cost. Besides that, single precision is utilized to solve m_k, thereby reducing the effectiveness of the preconditioner M.

Algorithm 6: The extended version of of mixed-precision heuristic sparse approximate inverse preconditioning algorithm
For every column $m_k, k = 1, 2, \cdots, n$ of M:

1) Choose an initial sparsity $J_k^0 = \{k\}$, set $l = 1$, $C_k^0 = J_k^0$, a suitable tolerance ε, $l_{\max}$, and compute u_k by (10);
2) Solve initial m_k using **Algorithm** 4 and compute r_k with double precision;
While $\|r_k\|_2 > \varepsilon$ and $l < l_{\max}$ and $|J_k^{l-1}| < u_k$:
3) Compute C_k^l by (7);
4) Save the indices that belong to C_k^l but not in J_k^{l-1} into set $\tilde{J}_k^l$;
5) For every $j \in \tilde{J}_k^l$, compute ρ_j^2 by (9), and delete from $\tilde{J}_k^l$ all but the most profitable indices; (single precision)
6) Determine the new row indices $\tilde{I}_k^l$ and then execute the QR decomposition of the new submatrix $A(I_k^{l-1} \cup \tilde{I}_k^l, J_k^{l-1} \cup \tilde{J}_k^l)$; (double precision)

7) Compute new m_k, r_k, and $||r_k||_2$, then set $J_k^l = J_k^{l-1} \cup \tilde{J}_k^l$, $I_k^l = I_k^{l-1} \cup \tilde{I}_k^l$, $C_k^l = J_k^l$, and $l = l + 1$; (single precision)

IV. PARALLEL IMPLEMENTATION OF MIXED PRECISION SPARSE APPROXIMATE INVERSE PRECONDITIONING ALGORITHM ON GPU

First, the parallel version of mixed-precision static SPAI preconditioning algorithm, called MP-SSPAI, is given as below, which includes three stages:

Pre-MP-SSPAI stage

First, allocate global memory to A on GPU. Second, as mentioned early, preconditioner M is computed in parallel by column, thus, all of A, $A1$ and M are stored in CSC(Compressed Sparse Column) format, which includes three arrays: *A_cData*, *A_cIndex* and *A_cPtr*. Third, to facilitate the calculation of matrix-vector product in iterative process, convert the storage format of A and M into CSR(Compressed Sparse Row), which also includes three arrays: *A_rData*, *A_rIndex* and *A_rPtr*. Fourth, to simplify the accesses of data in memory and enhance the coalescence, the dimensions of all local submatrices (e.g., $\widehat{A}_k$ $(n1_k, n2_k)$) are uniformly defined as $(maxI, maxJ)$, where $maxI = \max_k\{n1_k\}$ and $maxJ = \max_k\{n2_k\}$. Finally, allocate global memory to these arrays used in MP-SSPAI shown in Table 1, where $I = \{I_1, I_2, \cdots, I_k, \cdots, I_n\}$ and $J = \{J_1, J_2, \cdots, J_k, \cdots, J_n\}$.

TABLE 1. Arrays Used in MP-SSPAI

Arrays	Size	Type	Arrays	Size	Type
A_cData	*nonzeros*	single	*jPTR*	n_1 [a]	integer
A_cIndex	*nonzeros*	integer	*J*	$n_1 \times maxJ$ [b]	integer
A_cPtr	$n+1$	integer	*iPTR*	n_1	integer
A1_rData	*nonzeros*	double	*I*	$n_1 \times maxI$ [c]	integer
A1_rIndex	*nonzeros*	integer	$\hat{m}$	$n_1 \times maxJ$	double
A1_rPtr	$n+1$	integer	$\hat{A}(Q)$	$n_1 \times maxI \times maxJ$	single
atom	n	*integer*	*R*	$n_1 \times maxJ \times maxJ$	single

[a] The number of columns executed in parallel at one time.
[b] The maximum of padding toplimits of all columns of M.
[c] The maximum number of row indices of M.

Compute-MP-SSPAI stage

In this stage, a thread group consisted of z threads is used to compute one column of M(e.g., m_k). Thus, it can compute 512/z columns in parallel when a block is assigned 512 threads. And further columns of M can be computed simultaneously by multiple blocks. For z, assume that the number of theads in a block is set to 256, it varies with the value of *maxJ* of sparse matrix. Its principal thought is: if *maxJ* is less than or equal to 2, we set z to 2; if *maxJ* belongs to the right closed interval 2 to 4, z is set to 4; and so on until *maxJ* exceeds upper bound 256, z is set to 256. In addition, one m_k is computed in parallel by z threads. Taking m_k as an example, its specific process is shown below:

1) Determine J_k: Threads within a thread group are assigned to write its row indices of nonzero entries of M into J_k in parallel.
2) Determine I_k: Firstly, for c, the first element of J_k, threads in the thread group load row indices of $A(:, c)$ into I_k in parallel. Then, for other elements of J_k, namely, the corresponding columns of A, row indices of them are compared successively with elements in I_k. Those indices not in I_k will be appended into I_k using the atomic operations. Finally, these elements in I_k are sorted in ascending order in parallel.
3) Construct $\hat{A}_k$: After determining J_k and I_k, a thread group is assigned to construct submatrix $\hat{A}_k = A(I_k, J_k)$. And it includes two steps: firstly, load row indices of I_k in parallel, then, determine the elements of $\hat{A}$ according to column indices of J_k. The Figs. 3 and 4 show the kernel and main procedure of constructing submatrix $\hat{A}_k$, respectively.
4) Decompose $\hat{A}_k$ to Q_kR_k: A thread group is assigned to perform one Q_kR_k decomposition. To be more efficient, shared memory is utilized in this stage. The kernel and main procedure of QR decomposition are shown in Figs. 5 and 6, respectively. As shown in Fig. 6, for each loop i, firstly, read the ith column of $\hat{A}_k$ into Q_k in parallel. Second, compute $R_k(i, i : AN)$ and save them into shared memory *R_s* in parallel. Third, normalize column i of Q_k and compute projection factors $R_k(i, i : AN)$ and the corresponding *R_s* in parallel. Finally, update Q_k using shared memory *R_s*.
5) Compute $\hat{m}_k$: As mentioned in Algorithm 1, $\hat{m}_k = R_k^{-1}Q_k^T\hat{e}_k$. Thus, inside a thread group, firstly, we compute $Q_k^T\hat{e}_k$ in parallel. And then a upper triangular linear system($R_k\hat{m}_k = Q_k^T\hat{e}_k$) is solved to gain $\hat{m}_k$ in parallel. Similarly, we give its kernel and main procedure in Figs. 7 and 8.

```
template <unsigned int WarpSize>
__global__ void Compute_Ahat(float *Ahat, float *A_cData, int *A_cPtr, int
*A_cIndex, int n, int *I, int *iPTR, int *J, int *jPTR, int MAXI, int MAXJ){
  int gid = blockIdx.x * blockDim.x + threadIdx.x;
  int offset = blockDim.x / WarpSize * gidDim.x;
  int Warp_id = gid / WarpSize;
  int lane = gid & ( WarpSize - 1 );
  int col, i, irow, j, jcol, jcol_b, jcol_e, jcol1, AM, AN;
  float idata;
  for(col = warp_id; col < n; col += offset){
    AM = iPTR[col];
    AN = jPTR[col];
    for( i = 0; i < AM; i++){
      irow = I[col * MAXI + i];
      for( j = lane; j < AN; j+= warpSize ){
        jcol = J[col * MAXJ + j];
        jcol_b = A_cPtr[jcol];
        jcol_e = A_cPtr[jcol + 1];
        idata = 0.0;
        for( jcol1 = jcol_b; jcol1 < jcol_e; jcol1++){
          if( A_cIndex[jcol1] == irow){
            idata = A_cData[jcol1];
            break;
          }
        }
        Ahat[col * MAXI * MAXJ + iMAXJ + j] = idata;
      }
      __syncthreads();
    }
  }
}
```

FIGURE 3. Kernel of constructing submatrix $\hat{A}$

Post-MP-SSPAI stage

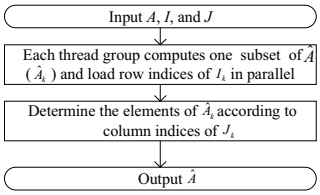

FIGURE 4. **Main procedure of constructing submatrix $\hat{A}$**

```
template <unsigned int WarpSize, unsigned int Size_R_Shared >
__global__ void QRDecomposition with Shared Memory(float *Q, float *R,
 int n, int *iPTR, int *jPTR, int MAXI, int MAXJ ){
__shared__ float R_s[Size_R_Shared];
 int gid = blockIdx.x * blockDim.x + threadIdx.x;
 int offset = blockDim.x /WarpSize * gidDim.x;
 int Warp_id = gid / WarpSize;
 int lane = gid & ( WarpSize - 1 );
 int tid = threadIdx.x/ WarpSize;
 int col, i, j, k, AM, AN;
 float rii, tR;
 int segR = Size_R_Shared * WarpSize / blockDim.x;
 for(col = warp_id; col < n; col += offset ){
  AM = iPTR[col];
  AN = jPTR[col];
  for( i = 0; i < AN; i++){
    for( j = lane + i; j < AN; j+= warpSize ){
      tR = 0.0;
      for( k = 0; k < AM; k++){
        tR += Q[col * MAXI * MAXJ + i + k * MAXJ]
            + Q[col * MAXI * MAXJ + j + k * MAXJ];
      }
      R_s[tid * segR + j - i] = tR;
    }
    __syncthreads();
    rii = sqrt(R_s[tid * segR]);
    for( j = lane; j < AM; j += WarpSize ){
      Q[col * MAXI * MAXJ + j * MAXJ + i] /= rii;
    }
    __syncthreads();
    for( j = lane + i; j < AN; j += WarpSize ){
      R_s[tid * segR + j - i] /= rii;
      R[col * MAXJ * MAXJ + i * MAXJ + j] /= R_s[tid * segR + j - i];
    }
    __syncthreads();
    for( j = lane + i +1; j < AN; j += WarpSize ){
      for( k = 0; k < AM; k++){
        Q[col * MAXI * MAXJ + k * MAXJ + j] -= R_s[tid * segR + j - i]
                                             * Q[col * MAXI * MAXJ + k * MAXJ + i];
      }
    }
    __syncthreads();
  }
 }
}
```

FIGURE 5. **Kernel of *QR* decomposition**

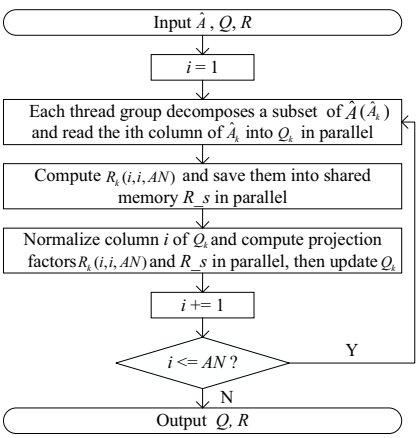

FIGURE 6. **Main procedure of *QR* decomposition**

```
template <unsigned int WarpSize>
__global__ void Solve M (float *Q, float *R, double *X, int *E,
 int n, int *jPTR, int MAXI, int MAXJ ){
 int gid = blockIdx.x * blockDim.x + threadIdx.x;
 int offset = blockDim.x /WarpSize * gidDim.x;
 int Warp_id = gid / WarpSize;
 int lane = gid & ( WarpSize - 1 );
 int col, i, j, AN;
 for(col = warp_id; col < n; col += offset ){
   AN = jPTR[col];
   if(E(col) == -1 ){
     for( i = lane; i < AN; i+= warpSize ){
       X[col * MAXJ + i] = 0.0;
     }
   }else{
     for( i = lane; i < AN; i+= warpSize ){
       X[col * MAXJ + i] = Q[col * MAXI * MAXJ + E[col] * MAXJ + i];
     }
   }
   __syncthreads();
   for( i = AN -1; i ≥ 0; i--){
     if(lane == 0 ){X[col * MAXJ + i] /= R[col * MAXJ * MAXJ + i];}
     __syncthreads();
     for( j = lane; j < i; j+= warpSize ){
       X[col * MAXJ + j] -= R[col * MAXJ * MAXJ + j * MAXJ + i]
                          * X[col * MAXJ + i];
     }
     __syncthreads();
   }
 }
}
```

FIGURE 7. **Kernel of solving upper triangular linear systems**

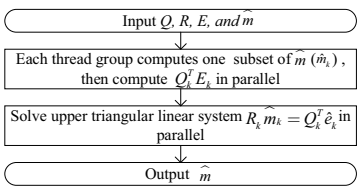

FIGURE 8. **Main procedure of solving upper triangular linear systems**

This stage is to assemble M in the CSC storage format, and store it to the *MPtr*, *MIndex*, and *MData* arrays. it includes the following steps:

1) On the GPU, we assemble *MPtr* using *JPTR*;
2) Utilizing $\widehat{m}_k$ and J_k to assemble *MData* and *MIndex*. Each warp is responsible for assembling one $\widehat{m}_k$ to *MData* and one J_k to *MIndex* in parallel.

Obviously, *MPtr*, *MIndex*, and *MData* arrays are generated on the GPU memory and do not need to be transferred to the CPU.

Then, the parallel version of mixed-precision heuristic SPAI preconditioning algorithm, called MP-HeuriSPAI, is given. It also consists of the following three phases:

Initial-MP-HeuriSPAI stage

In this phase, first, allocate memory for coefficient matrix A on GPU. Second, the upper bounds of the filling non-zero elements in each column are computed in parallel. Then, appropriate memory is allocated for the main arrays (as shown in Table 2). Finally, the parallel implementation of MP-SSPAI is used to compute the initial m_k and r_k.

Compute–MP-HeuriSPAI stage

This stage is basically the same as the computing stage of HeuriSPAI in literature [33], except that single precision com-

TABLE 2. Arrays Used in MP-HeuriSPAI

Arrays	Size	Type	Arrays	Size	Type
AData	*nonzeros*	double	*JPTR*	n	integer
AIndex	*nonzeros*	integer	J	$n \times maxJ$	integer
APtr	$n+1$	integer	*IPTR*	n	integer
CData	$n \times maxI$	double	I	$n \times maxI$	integer
CIndex	$n \times maxI$	integer	$\hat{J}$	$n \times maxJ$	integer
CPtr	n	integer	$\tilde{J}PTR$	n	integer
$\hat{A}$	$n \times maxI \times maxJ$	single	$\tilde{I}$	$n \times maxI$	integer
Q	$n \times maxI \times maxJ$	single	$\tilde{I}PTR$	n	integer
R	$n \times maxJ \times maxJ$	single	$\hat{m}$	$n \times maxJ$	double
atom	n	integer	$\hat{r}$	$n \times maxI$	double

putation will be used in the computation of ρ, the construction of the submatrix $A(I_k \cup \tilde{I}_k, J_k \cup \tilde{J}_k)$, and its QR decomposition, as detailed in literature [33].

Post-MP-SSPAI stage

This stage is also to assemble M in the CSC storage format.

V. EFFECTIVENESS ANALYSIS AND PERFORMANCE EVALUATION

In this section, we evaluate the performance of MP-SSPAI and MP-HeuriSPAI. Table 3 shows the overview of NVIDIA GPUs that are used in the performance evaluation. The test matrices are selected from the SuiteSparse Matrix Collection [47], and have been widely used in some previous work [18], [32], [33], [44]. Table 4 gives the information of the sparse matrices, including the name, kind, number of rows, total number of nonzeros, and positive definiteness. In addition, the constructed preconditioner is applied to GPUPBICGSTAB (a parallel implementation of the preconditioned BICGSTAB on GPU using the CUBLAS [45] and CUSPARSE [46] libraries). And the source codes are compiled and executed using the CUDA toolkit 11.0 [1]. Note that in all experiments below, iteration stops when the residual error is less than $1e^{-7}$ or the number of iterations exceeds 10,000.

TABLE 3. Overview of GPUs

Hardware	GTX1070	TITANXp
Cores	1920	3840
Clock speed (GHz)	1.506	1.480
Memory type	GDDR5	GDDR5X
Memory size (GB)	8	12
Max-bandwidth (GB/s)	256	548
Compute capability	6.1	6.1

A. EFFECTIVENESS ANALYSIS

First of all, we evaluate the effectiveness of MP-SSPAI by comparing it with original static SPAI preconditioning algorithm (SSPAI). The selected test matrices are same as literature [44]. Both of GPUPBICGSTAB with SSPAI and GPUPBICGSTAB with MP-SSPAI are used to solve $Ax = b$. Table 5 gives the comparison results of GPUPBICGSTAB with SSPAI and GPUPBICGSTAB with MP-SSPAI on GTX1070, where "Iters", "preTime" and "allTime" represent the number of iterations, preprocessing time (the execution time of preconditioner), and total runtime(the execution time of preconditioner and iterative algorithm), respectively. In addition, $P_{preTime}$ and $P_{allTime}$ indicate the reduction rate of preprocessing time of MP-SSPAI relative to original SSPAI and total runtime of GPUPBICGSTAB with MP-SSPAI relative to GPUPBICGSTAB with SSPAI, respectively. For all experiments, the minimum value of total runtime is marked in red for all selected sparse matrices.

TABLE 4. Descriptions of Test Matrices

Name	Kind	Rows	Nonzeros	Positive-Definite
cbuckle	structural	13,681	676,515	yes
gyro_m	duplicate model reduction	17,361	340,431	yes
venkat01	CFD sequence	62,424	1,717,792	no
2cubes_sphere	electromagnetics	101,492	1,647,264	yes
imagesensor	semiconductor device	118,758	1,446,396	no
cfd2	CFD	123,440	3,085,406	yes
power9	semiconductor device	155,376	1,887,730	no
majorbasis	optimization	160,000	1,750,416	no
stomach	2D/3D	213,360	3,021,648	no
CurlCurl_1	model reduction	226,451	2,472,071	no
offshore	electromagnetics	259,789	4,242,673	yes
ASIC_320ks	circuit simulation	321,671	1,316,085	no
test1	semiconductor device	392,908	9,447,535	no
msdoor	structural	415,863	19,173,163	yes
CoupCons3D	structural	416,800	17,277,420	no
Fault_639	structural	638,802	27,245,944	yes
apache2	structural	715,176	4,817,870	yes
t2em	electromagnetics	921,632	4,590,832	no
thermal2	thermal	1,228,045	8,580,313	yes
atmosmodd	CFD	1,270,432	8,814,880	no
Geo_1438	structural	1,437,960	60,236,322	yes
G3_circuit	circuit simulation	1,585,478	7,660,826	yes
af23560	CFD	23,560	460,598	no
FEM_3D_thermal2	thermal	147,900	3,489,300	no
cage13	directed weighted graph	445,315	7,479,343	no
af_shell3	subsequent structural	504,855	17,562,051	yes
parabolic_fem	CFD	525,825	3,674,625	yes
ecology2	2D/3D	999,999	4,995,991	yes

Observing Table 5, compared to SSPAI, firstly, MP-SSPAI has shorter execution time for all test matrices. Then, from the analysis of iterations, for cbuckle, gyro_m, cfd2, CurlCurl_1, ASIC_320ks, msdoor, apache2, t2em, thermal2, Geo_1438, and G3_circuit, GPUPBICGSTAB with MP-SSPAI reduces their number of iterations. In particular, for matrices cfd2, msdoor, and apache2, their number of iterations are significantly reduced. After that, for matrices venkat01, 2cubes_sphere, power9, majorbasis, stomach, offshore, CoupCons3D, Fault_639, and atmosmodd, GPUPBICGSTAB with MP-SSPAI keeps their number of iterations unchanged. Finally, GPUPBICGSTAB with MP-SSPAI also has shorter total execution time for all test matrices except for imagesensor. In addition, compared to SSPAI, for all

TABLE 5. Comparison Results of GPUPBICGSTAB with SSPAI and GPUPBICGSTAB with MP-SSPAI on GTX1070

Matrices	SSPAI			MP-SSPAI			$P_{preTime}$	$P_{allTime}$
	Iters	preTime	allTime	Iters	preTime	allTime		
cbuckle	96	8.009	8.395	94	7.802	**8.107**	2.6%	3.4%
gyro_m	180	0.818	1.189	178	0.704	**1.153**	1.2%	3.0%
venkat01	35	1.177	1.440	35	0.969	**1.300**	1.9%	9.7%
2cubes_sphere	4	0.851	1.172	4	0.734	**1.029**	13.7%	12.2%
imagesensor	52	0.343	**0.692**	976	0.274	1.259	20.1%	-81.9%
cfd2	1583	2.209	4.345	1375	1.823	**3.822**	38.6%	12.0%
power9	37	4.524	5.032	37	4.413	**5.014**	2.5%	0.4%
majorbasis	20	0.390	0.721	20	0.326	**0.635**	16.4%	12.0%
stomach	24	0.847	1.183	24	0.705	**1.149**	16.8%	2.9%
CurlCur_1	266	0.425	1.069	245	0.356	**0.946**	16.2%	11.5%
offshore	5	2.216	2.551	5	1.891	**2.180**	14.7%	14.5%
ASIC_320ks	10	4.918	5.269	8	4.629	**4.941**	5.9%	6.2%
test1	14	21.150	21.497	57	19.848	**20.817**	6.2%	3.2%
msdoor	892	61.099	66.456	626	57.854	**60.379**	5.3%	9.1%
CoupCons3D	52	77.494	78.102	52	73.377	**73.879**	5.3%	5.4%
Fault_639	1226	190.716	200.646	1226	183.426	**192.767**	3.8%	3.9%
apache2	1090	0.237	3.697	996	0.155	**2.759**	34.6%	25.4%
t2em	755	0.079	2.793	673	0.064	**2.398**	19.0%	14.1%
thermal2	2086	0.374	11.508	1920	0.281	**9.659**	24.9%	16.1%
atmosmodd	135	0.402	1.408	135	0.244	**1.228**	39.3%	12.8%
Geo_1438	339	148.765	154.977	330	133.783	**139.548**	10.1%	10.0%
G3_circuit	468	0.150	3.032	455	0.118	**2.903**	21.3%	4.3%

matrices, the preprocessing time of MP-SSPAI can be reduced by up to 39.3%, with an average reduction of 14.6%, while the total runtime of GPUPBICGSTAB with MP-SSPAI can be reduced by up to 25.4% relative to GPUPBICGSTAB with SSPAI, with an average reduction of 9.1% (except for imagesensor). To further demonstrate the superiority of MP-SSPAI performance, Fig. 9 shows the ratio of execution time of SSPAI to MP-SSPAI and total runtime of GPUPBICGSTAB with SSPAI to GPUPBICGSTAB with MP-SSPAI. Based on above analysis, MP-SSPAI is effective and widely applicable.

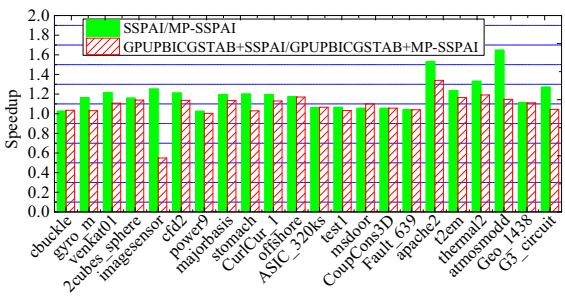

FIGURE 9. Ratio of execution time of SSPAI to MP-SSPAI and total runtime of GPUPBICGSTAB with SSPAI to GPUPBICGSTAB with MP-SSPAI

Then, to test the effectiveness of MP-HeuriSPAI, it was compared with HeuriSPAI [33]. The selected test matrices are same as literature [33]. The comparison results are shown in Table 6, where "Iters", "preTime", "allTime", $P_{preTime}$, and $P_{allTime}$ are the same as in Table 5.

Observing Table 6, firstly, we can see that the execution time of MP-HeuriSPAI is shorter than that of HeuriSPAI for all test matrices. Next, compared to GPUPBICGSTAB with HeuriSPAI, for gyro_m, af23560, af_shell3, and parabolic_fem, the number of iterations of GPUPBICGSTAB with

TABLE 6. Comparison Results of the GPUPBICGSTAB with HeuriSPAI and GPUPBICGSTAB with MP-HeuriSPAI on GTX1070

Matrices	HeuriSPAI			MP-HeuriSPAI			$P_{preTime}$	$P_{allTime}$
	Iters	preTime	allTime	Iters	preTime	allTime		
gyro_m	96	2.598	2.956	89	1.753	**2.078**	32.5%	29.7%
af23560	291	1.565	1.995	290	0.997	**1.414**	36.3%	29.1%
wenkat01	25	2.323	2.676	25	1.605	**1.942**	30.9%	27.4%
imagesensor	22	0.778	1.122	22	0.638	**1.057**	19.3%	5.8%
FEM_3D_thermal2	9	0.859	1.194	9	0.622	**0.924**	27.6%	22.6%
ASIC_320ks	8	7.675	8.020	8	5.369	**5.587**	30.0%	30.3%
cage13	8	0.922	1.241	8	0.664	**0.951**	28.0%	23.4%
af_shell3	441	37.873	52.634	421	34.308	**48.891**	9.4%	7.1%
parabolic_fem	288	0.883	2.354	279	0.649	**2.085**	26.5%	11.4%
apache2	694	0.975	3.634	697	0.727	**3.320**	25.4%	8.6%
t2em	574	0.659	3.253	583	0.511	**3.216**	22.4%	1.1%
ecology2	2665	0.701	**12.531**	2700	0.558	13.001	20.4%	-3.8%
thermal2	1449	2.681	12.179	1449	1.954	**12.063**	27.1%	1.0%
atmosmodd	117	0.991	1.976	117	0.722	**1.675**	27.1%	15.2%
G3_circuit	330	1.189	3.791	330	0.917	**3.593**	22.9%	5.2%

MP-HeuriSPAI is smaller, while it keeps unchanged for venkat01, imagesensor, FEM_3D_thermal2, ASIC_320ks, cage13, thermal2, atmosmodd, and G3_circuit. Moreover, for matrices apache2 and t2em, although GPUPBICGSTAB with MP-HeuriSPAI increases their number of iterations, it decreases their total execution time. And for all matrices except ecology2, the total execution time of GPUPBICGSTAB with MP-HeuriSPAI is less than that of the GPUPBICGSTAB with HeuriSPAI. In addition, for all matrices, the preprocessing time of MP-HeuriSPAI can be reduced by up to 36.3% relative to HeuriSPAI, with an average reduction of 25.7%, while the total runtime of GPUPBICGSTAB with MP-HeuriSPAI can be reduced by up to 30.3% relative to GPUPBICGSTAB with HeuriSPAI, with an average reduction of 14.5% (except for ecology2). To further prove the superiority of MP-HeuriSPAI performance, Fig. 10 shows the ratio of execution time of HeuriSPAI to MP-HeuriSPAI and total runtime of GPUPBICGSTAB with HeuriSPAI to GPUPBICGSTAB with MP-HeuriSPAI. The above analysis shows that MP-HeuriSPAI is effective.

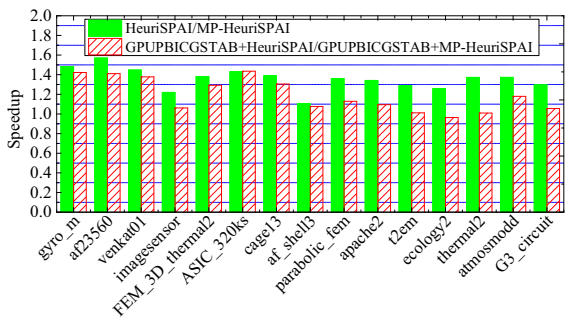

FIGURE 10. Ratio of execution time of HeuriSPAI to MP-HeuriSPAI and total runtime of GPUPBICGSTAB with HeuriSPAI to GPUPBICGSTAB with MP-HeuriSPAI

B. PERFORMANCE EVALUATION

In this subsection, firstly, using SSPAI as the standard, we compare MP-SSPAI with the single-precision version of the

static SPAI preconditioning algorithm (denoted as S-SSPAI), and its extended version(denoted as MP1-SSPAI) shown in Algorithm 4. In addition, this experiment will be performed on both NVIDIA GTX1070 and TITANXp GPUs, and test matrices are same as Table 5. The results are shown in Tables 7 and 8. In Tables 7 and 8, for each matrix, the first row is the number of iterations when GPUPBICGSTAB stops, the second row is the execution time of preconditioners, and the third row is the total execution time, which includes the execution time of preconditioner and iterative algorithm. In addition, for all experiments, the minimum value of total execution time is marked in red for all selected sparse matrices.

From Table 7, we can see that on GTX1070 GPU, compared with SSPAI, for all matrices except test1, S-SSPAI can effectively reduce their execution time. However, for cbuckle, inagesensor, cfd2, ASIC_320ks, msdoor etc. 12 matrices, the number of iterations of GPUPBICGSTAB with S-SSPAI is increased, and its total execution time is also increased for matrices inagesensor, cfd2, apache2, t2em, thermal2, atmosmodd, and G3_circuit. In particular, for matrix test1, GPUPBICGSTAB with S-SSPAI does not converge under the iterative stopping condition. For MP1-SSPAI, it effectively reduces the execution time for most matrices. However, for cbuckle, gyro_m, inagesensor, cfd2, CurlCurl_1, etc. 11 test matrices, the number of iterations of GPUPBICGSTAB with MP1-SSPAI is increased, and for inagesensor, cfd2, thermal2, etc. 9 test matrices, the total execution time is also increased. In particular, for test1, it does not converge under the iteration stopping condition. For MP-SSPAI, the analysis of Table 5 shows that it not only has high effectiveness and computational efficiency, but also is more stable and applicable. Further, on TITANXp GPU, analyzing Table 8, we can see that the performance of MP-SSPAI is also better than that of SSPAI, S-SSPAI, and MP1-SSPAI.

Then, using HeuriSPAI as the standard, this subsection compares MP-HeuriSPAI with a single precision version of HeuriSPAI (denoted as S-HeuriSPAI) and its extended version(denoted as MP1-HeuriSPAI) shown in Algorithm 6. Tables 9 and 10 provide their comparison results on GTX1070 and TITANXp, respectively.

From Tables 9 and 10, we can see that on both of GTX1070 and TITANXp, firstly, the execution time of S-HeuriSPAI is shorter than that of HeuriSPAI for all test matrices except imagesensor. However, for af23560, ASIC_320ks, af_shell3, parabolic_fem, apache2, t2em, ecology2, and thermal2, the number of iterations of GPUPBICGSTAB with S-HeuriSPAI is significantly higher than that of GPUPBICGSTAB with HeuriSPAI, especially for imagesensor, GPUPBICGSTAB with S-HeuriSPAI does not converge under the iteration stopping condition. Considering the total execution time, for parabolic_fem, t2em, ecology2 and thermal2, GPUPBICGSTAB with S-HeuriSPAI has longer total execution time than that of GPUPBICGSTAB with HeuriSPAI. The above analysis shows that S-HeuriSPAI does not improve the performance of HeuriSPAI. For MP1-HeuriSPAI, the analysis shows that overall, its performance is comparable to that of S-

TABLE 7. Comparison Results of the GPUPBICGSTAB with SSPAI, GPUPBICGSTAB with S-SSPAI, GPUPBICGSTAB with MP1-SSPAI, and GPUPBICGSTAB with MP-SSPAI on GTX1070

Matrices	SSPAI	S-SSPAI	MP1-SSPAI	MP-SSPAI
cbuckle	96 8.009 8.395	98 7.798 8.137	106 7.975 8.401	94 7.802 **8.107**
gyro_m	180 0.818 1.189	180 0.706 1.161	181 0.810 1.174	178 0.704 **1.153**
venkat01	35 1.177 1.440	35 0.982 1.405	35 1.036 1.470	35 0.969 **1.300**
2cubes_sphere	4 0.851 1.172	4 0.735 1.036	4 0.771 1.067	4 0.734 **1.029**
imagesensor	52 0.343 **0.692**	2709 0.268 2.469	1304 0.319 1.528	976 0.274 1.259
cfd2	1583 2.209 4.345	1715 1.814 5.071	1670 1.897 4.831	1375 1.823 **3.822**
power9	37 4.524 5.032	37 4.436 5.018	37 4.530 5.034	37 4.413 **5.014**
majorbasis	20 0.390 0.721	20 0.327 0.637	20 0.410 0.717	20 0.326 **0.635**
stomach	24 0.847 1.183	24 0.705 1.152	24 0.864 1.174	24 0.705 **1.149**
CurlCurl_1	266 0.425 1.069	251 0.353 0.965	281 0.376 1.024	245 0.356 **0.946**
offshore	5 2.216 2.551	5 1.877 **2.178**	5 1.991 2.285	5 1.891 2.180
ASIC_320ks	10 4.918 5.269	33 4.621 5.049	14 4.957 5.276	8 4.629 **4.941**
test1	14 21.150 21.497	>10000 / /	>10000 / /	57 19.848 **20.817**
msdoor	892 61.099 66.456	980 57.903 65.749	775 59.887 64.562	626 57.854 **60.379**
CoupCons3D	52 77.494 78.102	52 73.453 74.054	52 75.923 76.450	52 73.377 **73.879**
Fault_639	1226 190.716 200.646	1282 182.187 194.881	1249 186.141 196.242	1226 183.426 **192.767**
apache2	1090 0.237 3.697	1199 0.151 3.916	1152 0.166 3.796	996 0.155 **2.759**
t2em	755 0.079 2.793	775 0.062 2.837	825 1.005 3.023	673 0.064 **2.398**
thermal2	2086 0.374 11.508	2922 0.280 16.529	2433 0.467 13.344	1920 0.281 **9.659**
atmosmodd	135 0.402 1.408	140 0.349 1.434	135 0.288 1.231	135 0.244 **1.228**
Geo_1438	339 148.765 154.977	372 133.779 140.537	411 136.510 143.933	330 133.783 **139.548**
G3_circuit	468 0.150 3.032	470 0.116 3.172	464 0.130 2.916	455 0.118 **2.903**

TABLE 8. Comparison Results of the GPUPBICGSTAB with SSPAI, GPUPBICGSTAB with S-SSPAI, GPUPBICGSTAB with MP1-SSPAI, and GPUPBICGSTAB with MP-SSPAI on TITANXp

Matrices	SSPAI	S-SSPAI	MP1-SSPAI	MP-SSPAI
cbuckle	96 4.886 5.272	100 4.798 5.014	105 4.672 5.278	95 4.341 **4.884**
gyro_m	180 0.577 0.946	180 0.465 0.917	183 0.569 0.931	178 0.457 **0.910**
venkat01	35 1.118 1.216	35 0.923 1.181	35 0.977 1.246	35 0.910 **1.076**
2cubes_sphere	4 0.573 0.942	4 0.457 0.806	4 0.493 0.837	4 0.439 **0.799**
imagesensor	52 0.332 **0.571**	2531 0.256 2.048	1130 0.318 1.107	823 0.251 1.038
cfd2	1601 1.512 3.647	1723 1.384 4.218	1690 1.577 4.311	1392 1.385 **3.462**
power9	37 3.388 4.107	37 3.300 4.003	37 3.394 4.110	37 3.277 **3.916**
majorbasis	20 0.371 0.683	20 0.309 0.541	20 0.396 0.679	20 0.301 **0.524**
stomach	24 0.840 1.153	24 0.699 1.144	24 0.854 1.167	24 0.681 **1.139**
CurlCurl_1	266 0.301 0.827	261 0.229 0.720	276 0.245 0.813	225 0.214 **0.708**
offshore	5 1.752 2.081	5 1.413 **1.700**	5 1.527 1.815	5 1.414 1.704
ASIC_320ks	10 2.791 4.397	35 2.216 4.192	19 2.573 4.406	8 2.203 **3.826**
test1	14 14.297 16.754	>10000 / /	>10000 / /	118 9.741 **11.562**
msdoor	1029 37.801 41.375	1183 30.616 36.102	823 33.890 38.241	697 29.946 **34.335**
CoupCons3D	52 48.979 49.658	52 44.938 45.610	52 47.408 48.006	52 44.862 **45.435**
Fault_639	1149 123.937 129.867	1226 115.408 127.102	1172 119.362 125.463	1149 116.647 **121.988**
apache2	1190 0.230 3.751	1223 0.144 3.970	1198 0.159 3.853	1030 0.146 **3.664**
t2em	824 0.057 1.952	844 0.040 1.996	893 0.983 2.182	742 0.041 **1.557**
thermal2	2086 0.363 9.541	2735 0.276 14.211	2107 0.415 11.632	1918 0.268 **7.890**
atmosmodd	135 0.400 1.403	140 0.343 1.429	135 0.267 1.337	135 0.241 **1.205**
Geo_1438	339 101.544 109.010	365 91.925 97.251	415 94.645 99.323	334 91.617 **96.515**
G3_circuit	468 0.148 3.161	489 0.113 3.167	460 0.127 2.875	451 0.109 **2.736**

TABLE 9. Comparison Results of the GPUPBICGSTAB with HeuriSPAI, GPUPBICGSTAB with S-HeuriSPAI, GPUPBICGSTAB with MP1-HeuriSPAI, and GPUPBICGSTAB with MP-HeuriSPAI on GTX1070

Matrices	HeuriSPAI	S-HeuriSPAI	MP1-HeuriSPAI	MP-HeuriSPAI
gyro_m	96 2.598 2.956	96 2.083 2.509	99 2.428 2.782	89 1.753 **2.078**
af23560	291 1.565 1.995	292 1.030 1.446	290 1.513 1.941	290 0.997 **1.414**
venkat01	25 2.323 2.676	25 1.618 1.943	25 2.041 2.403	25 1.605 **1.942**
imagesensor	22 0.778 1.122	/ / /	2670 0.801 2.472	22 0.638 **1.057**
FEM_3D_thermal2	9 0.859 1.194	9 0.657 0.958	9 0.778 1.112	9 0.622 **0.924**
ASIC_320ks	8 7.675 8.020	13 5.361 5.681	8 7.454 7.799	8 5.369 **5.587**
cage	8 0.922 1.241	8 0.749 1.031	8 0.952 1.268	8 0.664 **0.951**
af_shell3	441 37.873 52.634	502 34.373 52.108	449 36.446 53.340	421 34.308 **48.891**
parabolic_fem	288 0.883 2.354	301 0.619 2.422	288 0.809 2.200	279 0.649 **2.085**
apache2	694 0.975 3.634	702 0.683 3.334	714 0.897 3.621	697 0.727 **3.320**
t2em	574 0.659 3.253	662 0.439 3.362	634 0.586 3.418	583 0.511 **3.219**
ecology2	2665 0.701 **12.531**	2717 0.509 13.123	2910 0.613 14.206	2700 0.558 13.001
thermal2	1449 2.681 12.179	2012 1.835 14.862	1635 2.181 12.855	1449 1.954 **12.063**
atmosmodd	117 0.991 1.976	117 0.716 **1.672**	117 0.880 1.862	117 0.722 1.675
G3_circuit	330 1.189 3.791	327 0.854 **3.407**	332 1.067 3.690	330 0.917 3.593

HeuriSPAI. And for MP-HeuriSPAI, compare to HeuriSPAI, Table 6 shows it effectively improves the validity of preconditioners and the computational efficiency for most matrices on GTX1070. Further, on TITANXp, analysis of Table 10 shows that this conclusion still holds. In summary, MP-HeuriSPAI is effective and superior to HeuriSPAI, S-HeuriSPAI, and MP1-HeuriSPAI.

The above experiments show that the proposed MP-SSPAI and MP-HeuriSPAI can improve the computational efficiency without increasing the number of iterations for most test matrices. why does the change in computational accuracy improve the convergence for most test matrices? In the transformation of the coefficient matrix A from double precision to single precision, although each data has only a small change, there is more data for large sparse matrices, and it involves complex calculations in multiple steps in the construction

TABLE 10. Comparison Results of the GPUPBICGSTAB with HeuriSPAI, GPUPBICGSTAB with S-HeuriSPAI, GPUPBICGSTAB with MP1-HeuriSPAI, and GPUPBICGSTAB with MP-HeuriSPAI on TITANXp

Matrices	HeuriSPAI	S-HeuriSPAI	MP1-HeuriSPAI	MP-HeuriSPAI
	96	96	99	89
gyro_m	2.082	1.837	2.016	1.376
	2.384	2.213	2.365	**1.716**
	291	293	290	290
af23560	1.456	0.921	1.404	0.888
	1.868	1.337	1.814	**1.287**
	25	25	25	25
venkat01	1.774	1.132	1.492	1.066
	1.629	1.018	1.356	**0.903**
	37	/	1320	37
imagesensor	0.768	/	0.803	0.627
	1.096	/	2.191	**1.024**
	9	9	9	9
FEM_3D_thermal2	0.817	0.608	0.765	0.601
	1.1753	0.936	1.101	**0.911**
	8	11	8	8
ASIC_320ks	5.132	3.418	5.015	3.441
	6.768	4.571	6.307	**4.492**
	8	8	8	8
cage	0.910	0.741	0.947	0.655
	1.227	1.019	1.254	**0.938**
	441	457	445	438
af_shell3	26.283	23.778	25.189	23.031
	41.377	38.769	39.162	**37.462**
	288	300	288	280
parabolic_fem	0.872	0.611	0.802	0.633
	2.217	2.364	2.171	**2.026**
	641	712	725	655
apache2	0.971	0.680	0.892	0.683
	2.314	2.287	2.769	**2.250**
	574	663	637	583
t2em	0.632	0.416	0.579	0.418
	2.543	2.736	2.603	**2.482**
	2802	2816	2937	2810
ecology2	0.540	0.429	0.537	0.430
	13.312	13.638	14.506	13.277
	2248	2811	2434	2248
thermal2	2.678	1.823	2.185	1.820
	13.168	14.975	14.239	**12.766**
	103	103	103	103
atmosmodd	0.546	0.314	0.401	0.312
	1.729	**1.604**	1.656	1.613
	330	327	332	330
G3_circuit	1.079	0.742	1.003	0.945
	3.691	**3.378**	3.529	3.423

of preconditioners. Therefore, these can cause error accumulation and alter its effectiveness. The experimental results demonstrate that the error accumulation in the proposed two mixed accuracy models improves or maintains the validity of the constructed preconditioners for most test matrices.

VI. CONCLUSIONS AND DISCUSSIONS

Based on the construction method of sparse approximate inverse(SPAI) preconditioners in mixed precision mode from the perspective of single and double precision mixing, two mixed precision sparse approximation inverse preconditioning algorithms, MP-SSPAI and MP-HeuriSPAI, are given in this paper, and their parallel implementations are also given. A series of experiments show that MP-SSPAI and MP-HeuriSPAI are effective and applicable to a wide range of applications. In the future, we will research on the error analysis of MP-SSPAI and MP-HeuriSPAI in theory to further confirm their high performance.

REFERENCES

[1] NVIDIA, "CUDA C Programming Guide," Version 11.1, 2021. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide
[2] M. Bernaschi, M. Carrozzo, A. Franceschini, and C. Janna, "A dynamic pattern factored sparse approximate inverse preconditioner on graphic processing units," *SIAM J. Sci. Comput.*, vol. 41, no. 3, pp. C139–C160, Jan. 2019.
[3] H. Liu, Z. X. Chen, and B. Yang, "Accelerating preconditioned iterative linear solvers on GPU," *Int. J. Numer. Anal. Mod.*, vol. 5, no. 1-2, pp. 136–146, Jan. 2014.
[4] Z. Xiao, T.X. Gu, Y.X. Peng, X.G. Ren, and J. Qi, "Mixed precision in CUDA polynomial precondition for iterative solver," *IEEE Int. Conf. Comput. Commun. Eng. Technol.*, Beijing, China, pp. 186–192, 2018.
[5] K. K. Phoon, F. H. Lee, and S. H. Chan, "Iterative solution of intersecting tunnels using the generalised Jacobi preconditioner," Proc. Int. Conf. Numerical Simulation of Construction Processes in Geotechnical Eng. for Urban Environment - Numerical Modelling of Construction Processes in Geotechnical Eng. for Urban Environment, Luniver Press, UK, 2008.
[6] S. H. Chan, K. K. Phoon, and F. H. Lee, "A modified Jacobi preconditioner for solving ill-conditioned Biot's consolidation equations using symmetric quasi-minimal residual method," *Int. J. Numer. Anal. Methods Geomech.*, vol. 25, no. 10, pp. 1001–1025, Aug. 2001.
[7] H. Anzt *et al.,* "Variable-size batched GaussÍCJordan elimination for block-Jacobi preconditioning on graphics processors," *Parallel Comput.*, vol. 81, pp. 131–146, Jan. 2019.
[8] H. Anzt *et al.,* " Batched Gauss-Jordan elimination for Block-Jacobi preconditioner generation on GPUs," International Workshop on Programming Models and Applications for Multicores and Manycores, 2017.
[9] M. Ferronato, C. Janna, and G. Gambolati, "A novel factorized sparse approximate inverse preconditioner with supernodes," presented at the 30th *Int. Symp. High Perform. Parallel Distrib. Comput.*, 2020.
[10] L. Grigori, Q. Niu, and Y. X. Xu, "Stabilized dimensional factorization preconditioner for solving incompressible Navier-Stokes equations," *Appl. Numer. Math.*, vol. 146, pp. 309–327, Dec. 2019.
[11] S. Laut, R. Borrell, and M. Casas, "Cache-aware sparse patterns for the factorized sparse approximate inverse preconditioner," *Adv. Eng. Softw.*, vol. 113, pp. 19–24, Jun. 2017.
[12] L. E. Carr III, C. F. Borges, and F. X. Giraldo, "Matrix-free polynomial-based nonlinear least squares optimized preconditioning and its application to discontinuous Galerkin discretizations of the Euler equations," *J. Sci. Comput.*, vol. 66, pp. 917–940, Jun. 2015.
[13] J. Cerdán, J. Marín, and A. Martínez, "Polynomial preconditioners based on factorized sparse approximate inverses," *Appl. Math. Comput.*, vol. 133, no. 1, pp. 171–186, Nov. 2002.
[14] M. B. van Gijzen, "A polynomial preconditioner for the GMRES algorithm," *J. Comput. Appl. Math.*, vol. 59, no. 1, pp. 91–107, Nov. 1993.
[15] E. Coleman and M. Sosonkina, "Self-stabilizing fine-grained parallel incomplete LU factorization," *Sustain. Comput. -Infor.*, vol. 19, pp. 291–304, Sep. 2018.
[16] M. M monga Made and H. A. van der Vorst, "A generalized domain decomposition paradigm for parallel incomplete LU factorization preconditionings," *Future Gener. Comp. Sy.*, vol. 17, no. 8, pp. 925–932, Jun. 2001.
[17] T. N. Phillips, "On methods of incomplete LU decompositions for solving Poisson's equation in annular regions," *Appl. Numer. Math.*, vol. 8, no. 6, pp. 515–531, Dec. 1991.
[18] J. Gao, Q. Chen, and G. He, "A thread-adaptive sparse approximate inverse preconditioning algorithm on multi-GPUs," *Parallel Comput.*, vol. 101, pp. 102724, Nov. 2021, [Online]. Available: https://doi.org/10.1016/j.parco.2020.102724
[19] L. González and A. Suárez, "Improving approximate inverses based on Frobenius norm minimization," *Appl. Math. Comput.*, vol. 219, no. 17, pp. 9363–9371, May. 2013.
[20] P. Tarazaga and D. Cuellar, "Preconditioners generated by minimizing norms," *Comput. Math. Appl.*, vol. 57, no. 8, pp. 1305–1312, Apr. 2009.
[21] B. Carpentieti, I. S. Duff, and L. Giraud, "Sparse pattern selection strategies for robust Frobenius-norm minimization preconditioners in electromagnetism," *Numer. Linear Algebr.*, vol. 7, no. 7-8, pp. 667–685, Otc. 2000.

[22] T. Hucle, "Approximate sparsity patterns for the inverse of a matrix and preconditioning," *Appl. Numer. Math*., vol. 30, no. 2-3. pp. 291–303, Jun. 1999.
[23] G. X. He, R. J. Yin, and J. Q. Gao, "An efficient sparse approximate inverse preconditioning algorithm on GPU," *Concurr. Comp. Pract. E*., 2019, [Online]. Available: https://doi.org/10.1002/cpe.5598
[24] E. Chow, "A priori sparsity patterns for parallel sparse approximate inverse preconditioners," *SIAM J. Sci. Comput*., vol. 21, no. 5, pp. 1804–1822, Apr. 2000.
[25] D. Bertaccini and S. Filippone, "sparse approximate inverse preconditioners on high performance GPU platforms," *Comput. Math. Appl*., vol. 71, no. 3, pp. 693–711, Feb. 2016.
[26] G. Oyarzun *et al.,* "MPI-CUDA sparse matrix-vector multiplication for the conjugate gradient method with an approximate inverse preconditioner," *Comput. Fluids*, vol. 92, pp. 244–252, Mar. 2014.
[27] M. M. Dehnavi, D. M. Fernández *et al.,* "Parallel sparse approximate inverse preconditioning on graphic processing untits," *IEEE Trans. Parallel Distrib. Syst*., vol. 24, no. 9, pp. 1852–1861, Sep. 2013.
[28] M. Lukash, K. Rupp, and S. Selberherr, "Sparse approximate inverse preconditioners for iterative solvers on GPUS," *Proc. Symp. High Perform. Comput*., Society for Computer Simulation: San Diego, CA, USA, 2012.
[29] Z. X. Jia and Q. Zhang, "Robust dropping criteria for F-norm minimization based sparse approximate inverse preconditioning," *BIT Numer. Math*., vol. 53, no. 4, pp. 959–985, Jun. 2013.
[30] M. J. Grote and T. Huckle, "Parallel preconditioning with sparse approximate inverses," *SIAM J. Sci. Comput*., vol. 18, no. 3, pp. 838–853, Jan. 1997.
[31] Z. Jia and B. Zhu, "A power sparse approximate inverse preconditioning procedure for large sparse linear systems," *Numer. Linear Algebr*., vol. 16, no. 4, pp. 259–299, Jul. 2009.
[32] J. Gao, X. Chu, X. Wu, J. Wang, and G. He, "Parallel dynamic sparse approximate inverse preconditioning algorithm on GPU," *IEEE T. Parall. Distr*., vol. 33, no. 12, pp. 4723–4737, Dec. 2022.
[33] J. Q. Gao, X. Y. Chu, and Y. Z. Wang, HeuriSPAI: "A heuristic sparse approximate inverse preconditioning algorithm on GPU," *CCF Trans. High Perform. Comput*., 2023, [Online]. Available: https://doi.org/10.1007/s42514-023-00142-2
[34] M. Baboulin *et al.,* "Accelerating scientific computations with mixed precision algorithms," *Comput. Phys. Commun*., vol. 180, no. 12, pp. 2526–2533, Dec. 2009.
[35] J. Kurzak and J. Dongarra, "Implementation of mixed precision in solving systems of linear equations on the Cell processor," *Concurr. Comp. -Pract. E*., vol. 19, no. 10, pp. 1371–1385, Jul. 2010.
[36] H. Anzt, B. Rocker, and V. Heuveline, "Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms," *Comput. Sci.-Res. Dev*., vol. 25, no. 3-4, pp. 141–148, Aug. 2010.
[37] A. Abdelfattah *et al.,* "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *Int. J. High Perform*., vol. 35, no. 4, pp. 344–369, Mar. 2021.
[38] A. R. Khaz'ali, M. R. Rasaei, and J. Moghadasi, "Iterative methods with mixed-precision preconditioning for ill-conditioned linear systems in multiphase CFD simulations," 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems(ScalA), pp. 1–8, 2021.
[39] F. Göbel, T. Grützmacher, T. Ribizel, and H. Anzt, "Mixed precision incomplete and factorized sparse approximate inverse preconditioning on GPUs," In Euro-Par 2021: Parallel Processing: 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, Proceedings 27, pp. 550–564, 2021.
[40] G. Flegar, H. Anzt, T. Cojean, and E. S. Quintana-Ortí, "Adaptive precision Block-Jacobi for high performance preconditioning in the Ginkgo Linear Algebra Software," *ACM Trans. Math. Softw*., vol. 47, no. 2, pp. 1–28, Apr. 2021.
[41] D. Kressner, Y. Ma, and M. Shao, "A mixed precision LOBPCG algorithm," *Numer. Algorithms* pp. 1–19, May 2023.
[42] N. Lindquist, P. Luszczek, and J. Dongarra, "Accelerating restarted GMRES with mixed precision arithmetic," *IEEE Trans. Parallel Distrib. Syst*., vol. 33, no. 4, pp. 1027–1037, Apr. 2022.
[43] H. Zhang, W. Ma, W. Yuan, J. Zhang, and Z. Lu, " Mixed-precision block incomplete sparse approximate preconditioner on Tensor core," *CCF Trans. High Perform. Comput*., pp. 1–14, Apr. 2023.
[44] X. Y. Chu, Y. Z. Wang, Q. Chen, and J. Q. Gao, "Optimizing the sparse approximate inverse preconditioning algorithm on GPU," *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 2, no. 4, pp. 100087, Mar. 2023.
[45] NVIDIA, "CUBLAS Library," 2022. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html
[46] NVIDIA, "CUSPARSE Library," 2022. [Online]. Available: https://docs.nvidia.com/cuda/cusparse/index.html
[47] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM T. Math. Software*, vol. 38, no. 1, pp. 1–25, Nov. 2011.

XINYUE CHU is currently a PH.D candidate of the School of Computer and Electronic Information at the Nanjing Normal University in Nanjing, China, and her current research interests include high-performance computing (HPC), parallel algorithms.