CCF Transactions on High Performance Computing (2023) 5:160–170
https://doi.org/10.1007/s42514-023-00142-2

REGULAR PAPER

HeuriSPAI: a heuristic sparse approximate inverse preconditioning algorithm on GPU

Jiaquan Gao[1] · **Xinyue Chu**[1] · **Yizhou Wang**[1]

Received: 29 July 2022 / Accepted: 14 March 2023 / Published online: 27 March 2023
© China Computer Federation (CCF) 2023

Abstract
In this study, we present a new heuristic sparse approximate inverse (SPAI) preconditioning algorithm on graphics processing unit (GPU), called HeuriSPAI. For the proposed HeuriSPAI, there are the following novelties: (1) a heuristic method is proposed, which gives the potential candidate indices of the nonzero entries of the preconditioner in advance to guide the selection of the new indices, so as to improve the quality of the obtained preconditioner; and (2) a parallel framework of constructing the heuristic SPAI preconditioner on GPU is presented on the basis of the new proposed heuristic SPAI preconditioning algorithm; and (3) each component of the preconditioner is computed in parallel inside a group of threads. HeuriSPAI fuses the advantages of static and dynamic SPAI preconditioning algorithms, and alleviates the drawback of the existing dynamic SPAI preconditioning algorithms on GPU that are not suitable for large matrices. Experimental results show that HeuriSPAI is effective for large matrices, and outperforms the popular preconditioning algorithms in three public libraries, as well as a recent parallel static SPAI preconditioning algorithm.

Keywords Sparse approximate inverse · Preconditioning · Heuristic · CUDA · GPU

1 Introduction

In the fields of science, engineering and economy etc., many problems can be modeled as the following linear system:

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

Here A is a large, sparse and nonsingular matrix, and x and b are unknown and known vectors, respectively. For solving the linear system in Eq. (1), the iterative methods such as the generalized minimal residual method (GMRES)(Saad and Schultz 1986) and the biconjugate gradient stabilized method (BICGSTAB)(van der Vorst 1992) have been widely applied. Furthermore, with a left or right preconditioner M, the original problem in Eq. (1) can be transformed into a more tractable form as:

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

A good preconditioner M should be easy to be constructed, and effective in reducing the iteration count of iterative methods. Popular preconditioners include the incomplete factorization preconditioners(Saad 2003; Gao et al. 2014; Anzt et al. 2017), the sparse approximate inverse (SPAI) preconditioners based on Frobenius norm minimization(Cosgrove et al. 1992; Grote and Huckle 1997; Chow 2000; Jia and Zhu 2009), the factorized sparse approximate inverse (FSAI) preconditioners(Kolotilina and Yeremin 1993; Benzi et al. 1996, 2000; Ferronato et al. 2014; Bernaschi et al. 2016) and the preconditioners that consist of an incomplete factorization followed by an approximate inversion of the incomplete factors(van der Vorst 1982; Duin 1999.)

The SPAI preconditioner based on Frobenius norm minimization uses approximate A^{-1} as the preconditioner M. As shown in Chow (2000), M is constructed to minimize $\|AM - E\|$ in the Frobenius norm:

$$\min \|AM - E\|_F^2. \tag{3}$$

✉ Jiaquan Gao
springf12@163.com

✉ Xinyue Chu
2316607219@qq.com

Yizhou Wang
1966224230@gmail.com

[1] Jiangsu Key Laboratory for NSLSCS, School of Computer and Electronic Information, Nanjing Normal University, Qixia street, Nanjing 210023, Jiangsu, China

Here E is an $n \times n$ unit matrix. The columns in M are independent of each other, and thus equation (3) can be transferred into the following n independent least squares problems:

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

where m_k and e_k represent the kth column of M and the unit matrix, respectively. Obviously, the construction of SPAI preconditioner is easily parallelized. Compared with the incomplete factorization preconditioners, SPAI preconditioners require only a few sparse matrix-vector multiplication operations instead of triangular solves. Compared with FSAI preconditioners, SPAI preconditioners are suitable for various matrices such as A, not just symmetric positive definite matrices. Therefore, SPAI preconditioners have attracted considerable attention.

Considering the construction method of M, SPAI can be categorized into static and dynamic types. If the sparsity of M is prescribed a priori, we will have a static SPAI preconditioning procedure. In contrast, the SPAI preconditioning algorithm is a fully dynamic one. Because the SPAI preconditioner construction is generally time-consuming for large matrices, in recent ten years, with the advent of graphics processing units (GPUs), many researchers have attempted to accelerate them on the GPU architecture. There exists some work on accelerating the construction of static SPAI preconditioners with GPU(Dehnavi et al. 2013; Gao et al. 2017; Rupp et al. 2016; He et al. 2020; Gao et al. 2021), However, in many applications, the difficulty lies in determining a good sparsity structure of the approximate inverse in advance when the static SPAI preconditioner is applied. Thus, the dynamic sparse approximate inverse algorithm that aims at searching the sparsity pattern of M automatically is proposed. As compared to the static SPAI preconditioning algorithm, the advantage of the dynamic SPAI preconditioning algorithm is that the sparsity pattern of M is automatically exploited; however, its disadvantage is that a great deal number of iterations is required to explore the nonzero entries of M. Moreover, the research on accelerating the construction of the dynamic SPAI preconditioners with GPU is scarce. Rupp et al. Rupp et al. (2016) present a parallel dynamic SPAI implementation procedure on GPU in the ViennaCL library but it works only for smaller matrices.

Based on the above motivation, in this study, we present a heuristic SPAI preconditioning algorithm on GPU, called HeuriSPAI. In our proposed HeuriSPAI, for each loop, we first present a heuristic method, which gives potential candidate indices of nonzero-entries of M in advance to guide the selection of new indices, and thus improves the quality of obtained M. Second, the loop-stopping condition adds $l < l_{\max}$ ($l_{\max}$ is a small integer) and $J_k \leqslant \alpha * n2_k$ (α is a small real number and $n2_k$ is the nonzero number of the kth column of A) besides $\|r_k\| \leqslant \varepsilon$. This guarantees the sparsity of the preconditioner. Third, a parallel framework of constructing the heuristic SPAI preconditioner is presented. Finally, each component of the preconditioner, such as sparse matrix-matrix multiplication, finding $\tilde{J}$, reducing $\tilde{J}$, determining $\tilde{I}$, QR decomposition, and computing m_k and r_k, is computed in parallel inside a group of threads. HeuriSPAI fuses the advantages of static and dynamic SPAI preconditioning algorithms, and alleviates the drawback of the existing dynamic SPAI preconditioning algorithms on GPU that are not suitable for large matrices. Experimental results show that HeuriSPAI is effective, and outperforms several popular preconditioned algorithms on GPU: CSRILU0 in the CUSPARSE library(NVIDIA 2021), the incomplete SPAI preconditioning algorithm in the MAGMA library(Anzt et al. 2018), a parallel static SPAI preconditioning algorithm and a parallel dynamic SPAI preconditioning algorithm in the ViennaCL library(Rupp et al. 2016), and a recent parallel static SPAI preconditioning algorithm(He et al. 2020).

The rest of this paper is organized as follows. In the second section, a new heuristic SPAI preconditioning algorithm is proposed. In the third section, a heuristic SPAI preconditioning algorithm on GPU is presented. Experimental evaluation and analysis are presented in the fourth section. The fifth section contains our conclusions and points out our future research directions.

2 A Heuristic SPAI algorithm

Assuming that each value of A is greater than or equal to 0, based on the characteristic polynomial for A, we have

$$A^{-1} = \alpha_0 E + \alpha_1 A + \cdots + \alpha_{n-1} A^{n-1}. \tag{5}$$

Therefore, the pattern of A^{-1} (denoted by $S(A^{-1})$) is contained in the pattern $\cup_{j=0}^{j=n-1} S(A^j)$. Thus we can obtain

$$S(A^{-1}) \subseteq S((E + A)^{n-1}). \tag{6}$$

Similarly, the Neumann representation

$$A^{-1} = \beta \sum_{j=0}^{\infty} (E - \beta A)^j \tag{7}$$

for small α shows that numerically $S((E + A)^j)$ is nearly contained in $S(A^{-1})$ for all j. Considering Eqs. (6) and (7), we can obtain

$$S(A^{-1}) \simeq S((E + A)^{n-1}). \tag{8}$$

Let us assume that we have already computed an optimal solution m_k, $k = 1, 2, \ldots, n$, with the residual r_k of the least squares problem relative to an initial index set $J_k^0 = \{k\}$. Next, for each l, $l = 1, 2, \cdots$, utilizing the idea of Eq. (8), we use

$$C_k^l = (E + |A|)C_k^{l-1} \tag{9}$$

to generate the candidate indices that might be added to J_k^l, where $C_k^0 = J_k^0$, and $|A|$ means to take the absolute value for each value of A. Let $\tilde{J}_k^l$ equal to the set of indices that appear in C_k^l but not in J_k^{l-1}. For each $j \in \tilde{J}_k^l$, we consider the following one-dimensional minimization problem(Grote and Huckle 1997):

$$\min_{\mu_j \in R} \|r_k + \mu_j A e_j\| =: \rho_j. \tag{10}$$

For every j, $\mu_j = -r_k^T A e_j / \|Ae_j\|_2^2$ and thus $\rho_j^2 = \|r_k\|_2^2 - (r_k^T A e_j)^2 / \|Ae_j\|_2^2$.

Obviously, indices with $(r_k^T A e_j)^2 = 0$ lead to no improvement in the one-dimensional minimization. We reduce $\tilde{J}_k^l$ to the set of the most profitable indices j with smallest ρ_j and add it to J_k^l. Using the augmented set of indices J_k^l, we solve the least squares problems in Eq. (4) again. We denote by $\tilde{I}_k^l$ the set of new indices, which correspond to the nonzero rows of $A(., J_k^{l-1} \cup \tilde{J}_k^l)$ not contained in I_k^{l-1}, and by $\tilde{n}_1$ and $\tilde{n}_2$ the number of indices in $\tilde{I}_k^l$ and $\tilde{J}_k^l$, and then have

$$\begin{aligned} A(I_k^{l-1} \cup \tilde{I}_k^l, J_k^{l-1} \cup \tilde{J}_k^l) &= \begin{pmatrix} \hat{A} & A(I_k^{l-1}, \tilde{J}_k^l) \\ 0 & A(\tilde{I}_k^l, \tilde{J}_k^l) \end{pmatrix} \\ &= \begin{pmatrix} Q & \\ & E_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} R & Q_1^T A(I_k^{l-1}, \tilde{J}_k^l) \\ 0 & Q_2^T A(I_k^{l-1}, \tilde{J}_k^l) \\ 0 & A(\tilde{I}_k^l, \tilde{J}_k^l) \end{pmatrix}. \end{aligned} \tag{11}$$

Here $\hat{A} \in R^{n_1 \times n_2}$ is the submatrix of eliminating all zero rows in $A(., J_k^{l-1})$, and Q and R are matrices obtained by the QR decomposition of $\hat{A}$, and Q_1 and Q_2 are the first n_2 columns and the last $(n_1 - n_2)$ columns of Q, respectively. Note that the modified Gram-Schmidt method(Brandes et al. 2012) is utilized to execute the QR decomposition in this study. We require only the computation of the QR decomposition of $B = \begin{pmatrix} Q_2^T A(I_k^{l-1}, \tilde{J}_k^l) \\ A(\tilde{I}_k^l, \tilde{J}_k^l) \end{pmatrix}$. Utilizing the QR decomposition, we can obtain the solution of the least squares problems in Eq. (4). If r_k satisfies the loop-stopping condition, the algorithm stops; otherwise, we set $I_k^l = I_k^{l-1} \cup \tilde{I}_k^l$ and $C^l = J_k^l$ and $l = l + 1$, and continue to execute the loop.

In order to decrease the computational complexity, the loop-stopping condition is set to $\|r_k\| \leqslant \varepsilon$ or $l < l_{\max}$ ($l_{\max}$ is a small integer) or $J_k \leqslant \alpha * n2_k$ (α is a small real number and $n2_k$ is the nonzero number of the kth column of A). We summarize the sequential version of our proposed heuristic SPAI algorithm in the following **Algorithm** 1.

Algorithm 1: Heuristic SPAI algorithm

Input : A, a tolerance ε, the maximum number of the heuristic computation $l_{\max}$, and α

Output : M For every column m_k of M:

1) Set $l = 1$ and $C_k^0 = \{k\}$, choose an initial sparsity $J_k^0 = \{k\}$.
2) Solve Eq. (4) to obtain m_k, and compute $r_k = e_k - Am_k$. while $\|r_k\|_2 > \varepsilon$ and $l < l_{\max}$ and $J_k \leqslant \alpha \cdot n2_k$:
3) $C_k^l = (E + |A|)C_k^{l-1}$.
4) Let $\tilde{J}_k^l$ equal to the set of indices that appear in C_k^l but not in J_k^{l-1}.
5) For every $j \in \tilde{J}_k^l$, compute $\rho_j^2 = \|r_k\|_2^2 - (r_k^T A e_j)^2 / \|Ae_j\|_2^2$, and delete from $\tilde{J}_k^l$ all but the most profitable indices.
6) Determine the new indices $\tilde{I}_k^l$, and execute the QR decomposition of B.
7) Solve the new least squares problem in Eq. (4) to obtain m_k, and compute the new residual $r_k = e_k - Am_k$.
8) Set $I_k^l = I_k^{l-1} \cup \tilde{I}_k^l$, $J_k^l = J_k^{l-1} \cup \tilde{J}_k^l$, $C^l = J_k^l$, and $l = l + 1$.

It is observed that as compared to the popular dynamic SPAI preconditioning algorithm in Grote and Huckle (1997), our proposed heuristic SPAI algorithm has the following two main difference: (1) a heuristic method is proposed to give potential candidate indices; (2) the loop-stopping condition adds $l < l_{\max}$ and $J_k \leqslant \alpha * n2_k$ besides $\|r_k\| \leqslant \varepsilon$, which can better maintain the sparsity level of the preconditioner. For the proposed heuristic SPAI algorithm, its computational complexity is roughly $O(maxI \times maxJ \times n)$, and the two operations such as the sparse matrix-matrix multiplication and QR decomposition for each iteration are the most time-consuming ones.

In this section, we present a parallel heuristic sparse approximate inverse preconditioning algorithm on GPU, called HeuriSPAI. Table 1 shows the main arrays used in HeuriSPAI. The parallel framework of HeuriSPAI is shown in Fig. 1, which includes three stages: *Init-HeuriSPAI* stage, *Compute-HeuriSPAI* stage, and *Post-HeuriSPAI* stage.

2.1 Init-HeuriSPAI stage

In the *Init-HeuriSPAI* stage, the global memory of GPU to A is first allocated. A is stored in memory using the CSC (Compressed Sparse Column) storage format, and M is also stored in columns. Second, when computing m_k (one column of M), $k = 1, 2, \cdots, n$, the dimensions of local submatrices $\hat{A}_k$ $(n1_k, n2_k)$ are usually distinct for different k. To simplify the accesses of data in memory and enhance the coalescence, the dimensions of all local submatrices are uniformly defined as $(maxI, maxJ)$, where $maxJ = \max_k\{\lceil \alpha \cdot n2_k \rceil\}$ and $maxI = v \cdot maxJ$, where v is an integer. Utilizing $maxI$ and $maxJ$, the main arrays that are used in HeuriSPAI (see Table 1) are defined, and the global memory of GPU to the

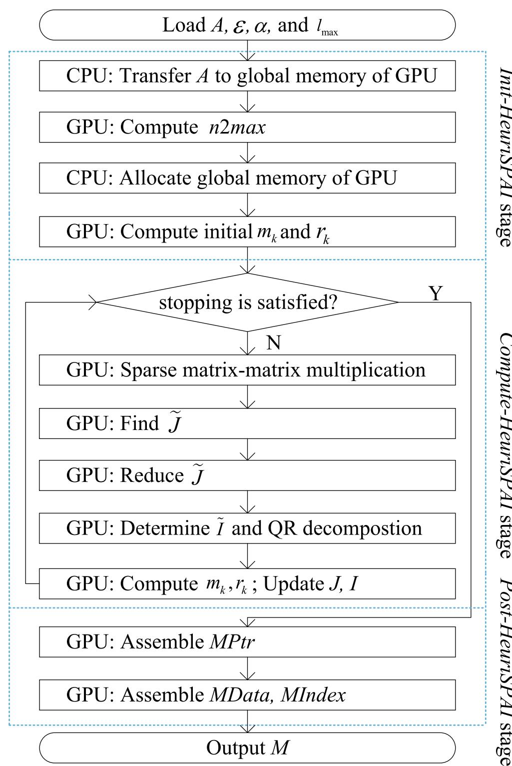

Fig. 1 Parallel framework of HeuriSPAI

Table 1 Arrays used in 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
$\hat{A}$	$n \times maxI \times maxJ$	Double	$\tilde{I}$	$n \times maxI$	Integer
Q	$n \times maxI \times maxJ$	Double	$\tilde{I}PTR$	n	Integer
R	$n \times maxJ \times maxJ$	Double	$\hat{m}$	$n \times maxJ$	Double
atom	n	Integer	$\hat{r}$	$n \times maxI$	Double

main arrays is allocated. Third, the initial values of m_k and r_k are obtained by setting $J_0^k = \{k\}$ on GPU.

2.2 Compute-HeuriSPAI stage

In the *Init-HeuriSPAI* stage, the initial values of m_k and r_k are obtained. The aim of the *Compute-HeuriSPAI* stage is to achieve better values of $m_k, k = 1, 2, \cdots, n$ by the iteration. One m_k vector is computed via one warp (32 threads in a block), many m_k vectors are computed simultaneously via warps executing in parallel. The parallelism is also exploited in a warp by computing one m_k vector in parallel using 32 threads inside a warp.

Sparse matrix-matrix multiplication. This step is to compute $C_k^l = (E + |A|)C_k^{l-1}$, $k = 1, 2, \cdots, n$. In fact, each warp finishes a sparse matrix-sparse vector multiplication. Here we present a novel sparse matrix-matrix multiplication on GPU, and its main procedure is shown in Fig. 2. In a warp, the sparse matrix-sparse vector multiplication, e.g., $(E + |A|)C_k^{l-1}$, is computed as follows. First, the row indices of the first column referenced in $CIndex_k$ are loaded into I_k, and the row index vector of successive columns referenced by $CIndex_k$ are then compared in parallel with values in I_k and new indices are appended to I_k using the atomic operations. Second, each thread computes one row whose indices is in I_k and the values are saved to $\hat{A}_k$. Finally, threads in a warp read $CData_k$ into shared memory $sCData$ in parallel, and then each thread computes one row of $\hat{A}_k \cdot sCData$, and save values to $CData_k$ and the corresponding indices to $CIndex_k$.

Finding $\hat{J}$: Each warp finds a subset of $\tilde{J}$ in this step. In a warp, a subset of $\tilde{J}$, e.g., $\tilde{J}_k$, is computed by the following procedure: the indices in $CIndex_k$ are compared in parallel with values in J_k and the different indices are written into $\hat{J}_k$.

Reducing $\tilde{J}$: In this step, from each subset in $\tilde{J}$ (e.g., $\tilde{J}_k$) all but the most profitable indices are deleted. Each subset of $\tilde{J}$, e.g., $\tilde{J}_k$, is reduced via one warp, which includes the following three stages. In the first stage, the threads in a warp compute ρ_j, $j \in \tilde{J}_k$, in parallel, and save them to shared memory. In the second stage, the values in shared memory are sorted in ascending order. The threads in a warp read ρ_j that is smaller than η from shared memory in parallel and rewrite their corresponding indices to $\tilde{J}_k$ in the third stage.

Determine $\tilde{I}$ and QR decomposition: This step is used to determine $\tilde{I}$ and decomposes the local submatrix into QR using Gram-Schmidt method. Each warp determines one set of $\tilde{I}$, e.g., $\tilde{I}_k$. For each $j \in \tilde{J}_k$, all threads inside a warp search the row indices in the jth column of A in parallel to find indices that are not included in I_k, and then write them to $\tilde{I}_k$ using the atomic operation. In the following, $\tilde{I}_k$ are sorted

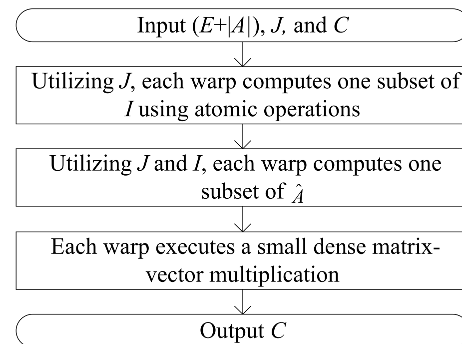

Fig. 2 Main procedure of sparse matrix-matrix multiplication

in parallel in an ascending order. In addition, each warp is also responsible for one QR decomposition in this step, and its main procedure is exhibited in Fig. 3. In a warp, the QR decomposition of the local submatrix, e.g., $\widehat{A}_k$, is composed of three steps at each iteration i. First, all threads compute the ith row of the upper triangle matrix R_k in parallel and put them into shared memory sR. Second, the threads in the warp concurrently normalize column i of Q_k, and compute the projection factors R_k and sR. The values of all columns of Q_k are updated by using shared memory sR and column i of Q_k in parallel in the third step.

Computing m_k and r_k: This step is used to compute m_k and r_k. As we know, m_k is obtained by scattering $\widehat{m}_k$, and $r_k = e_k - Am_k$. Therefore, the key of this step is to compute $\widehat{m}_k$ by solving $R_k\widehat{m}_k = Q_k^T\widehat{e}_k$. Each warp is responsible for computing one $\widehat{m}_k$. In a warp, computing values of $\widehat{m}_k$ includes two steps. In the first step, all threads inside a thread group compute $Q_k^T\widehat{e}_k$ in parallel and save values to shared memory xE. In the second step, the values of $\widehat{m}_k$ are obtained by solving the upper triangular linear system, $R_k\widehat{m}_k = xE$, in parallel using shared memory.

2.3 *Post-HeuriSPAI* stage

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

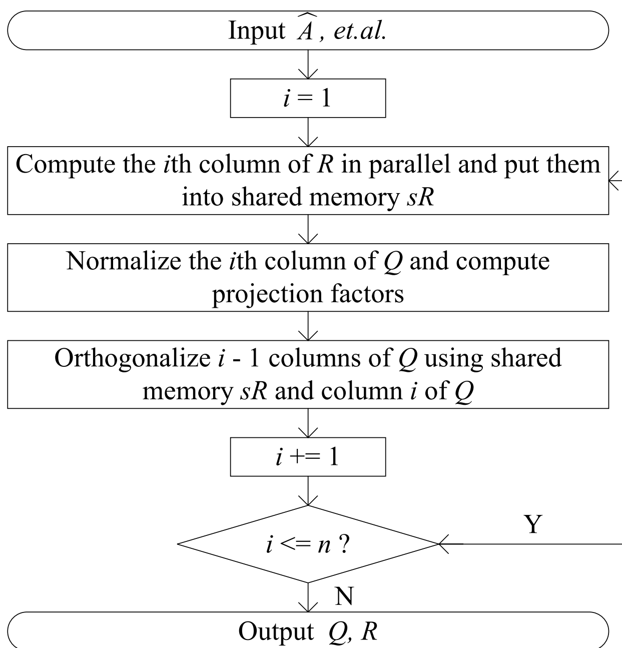

Fig. 3 Main procedure of decomposing $\widehat{A}$ into QR

1) On the GPU, we assemble *MPtr* using *JPTR*, as shown in Fig. 4.
2) Utilizing $\widehat{m}_k$ and J 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.

3 Evaluation and analysis

We evaluate the performance of HeuriSPAI in this section. Table 2 shows the overview of NVIDIA GPUs that are used in the performance evaluation. The test matrices are selected from the SuiteSparse Matrix Collection(Davis and Hu 2011). Table 3 summarizes the information of the sparse matrices, including the name, kind, number of rows, and total number of non-zeros. The test matrices are chosen due to the fact that they have been widely used in some previous work(Grote and Huckle 1997; Dehnavi et al. 2013; He et al. 2020; Gao et al. 2021).The source codes are compiled and executed using the CUDA toolkit 11.1(NVIDIA 2021). Note that in the following experiments, all algorithms use the double-precision floating point numbers in all computations.

3.1 Effectiveness analysis

First, we test the effectiveness of the approximate inverse matrices that are obtained by HeuriSPAI. For each matrix, both GPUBICGSTAB and GPUPBICGSTAB are called to

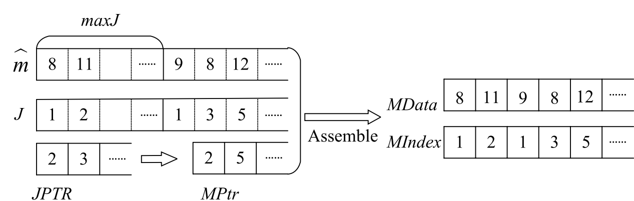

Fig. 4 Assemble M

Table 2 Overview of GPUs

Hardware	GTX1070	RTX3090
Cores	1920	10496
Clock speed (GHz)	1.56	1.70
Memory type	GDDR5	GDDR6X
Memory size (GB)	8	24
Max-bandwidth (GB/s)	256	384
Compute capability	6.1	8.6

Table 3 Descriptions of test matrices

Name	Kind	Rows	Nonzeros
Orsreg_1	CFD	2,205	14,133
Orsirr_1	CFD	1,030	6,858
Orsirr_2	CFD	886	5,970
Sherman1	CFD	1,000	3,750
Sherman2	CFD	1,080	23,094
Sherman3	CFD	5,005	20,033
Sherman4	CFD	1,104	3,786
Sherman5	CFD	3,312	20,793
Pores_2	CFD	1,224	9,613
Pores_3	CFD	532	3,474
Saylr4	CFD	3,564	22,316
Af23560	CFD	23,560	460,598
Zhao2	Electromagnetics	33,861	166,453
Venkat01	CFD sequence	62,424	1,717,792
Imagesensor	semiconductor device	118,758	1,446,396
Apache2	structural	715,176	4,817,870
T2em	electromagnetics	921,632	4,590,832
Thermal2	thermal	1,228,045	8,580,313
Atmosmodd	CFD	1,270,432	8,814,880
Nv2	semiconductor device	1,453,908	37,475,646
G3_circuit	circuit simulation	1,585,478	7,660,826
Ss	semiconductor process	1,652,680	34,753,577
Stokes	semiconductor process	11,449,533	349,321,980

solve $Ax = b$, where all elemental values of b are 1, and the produced M is used as the preconditioner, and the initial $x_0 = b$. GPUBICGSTAB and GPUPBICGSTAB are the parallel implementations of BICGSTAB and preconditioned BICGSTAB on GPU using the CUBLAS 11.1(NVIDIA 2021) and CUSPARSE 11.1(NVIDIA 2021) libraries, respectively, and stop when the residual error, which is defined as $\frac{\|b-Ax\|_2}{\|b-Ax_0\|_2}$, is less than $1e^{-7}$ or the number of iterations exceeds 10,000. Tables 4 and 5 show the number of iterations and execution time after the convergence of GPUBICGSTAB and GPUPBICGSTAB on GTX1070 and RTX3090, respectively. The time unit is second (s). Note that the execution time of GPUPBICGSTAB in Tables 4 and 5 includes the execution time of HeuriSPAI that is used to obtain the preconditioner; and "/" means that the execution time of the algorithm is not counted because its iterations exceed 10,000.

From Tables 4 and 5, we can observe that on two GPUs, without the preconditioner, for af25600, Zhao2, imagesensor, venkat01, nv2, G3_circuit, ss, and stokes, GPUBICGSTAB cannot converge to the 10^{-7} residual error in 10,000 iterations while GPUPBICGSTAB with HeuriSPAI can. For apache2, t2em, thermal2, and atmosmodd, GPUBICGSTAB can converge under 10,000 iterations, but the number

Table 4 Iterations and execution time of two algorithms on GTX1070

Matrix	GPUBICGSTAB		GPUPBICGSTAB	
	Iter	Exe time	Iter	Exe time
Af23560	> 10000	/	291	1.959
Zhao2	> 10000	/	1269	0.746
Venkat01	> 10000	/	21	1.656
Imagesenor	> 10000	/	31	1.069
Apache2	4207	8.852	566	3.855
T2em	1581	3.861	573	3.387
Thermal2	6620	23.947	1399	13.417
Atmosmodd	241	1.621	78	1.652
Nv2	> 10000	/	38	26.444
G3_circuit	> 10000	/	324	3.376
Ss	> 10000	/	81	31.663
Stokes	> 10000	/	775	288.501

of iterations decreases dramatically using the preconditioner. GPUPBICGSTAB has smaller execution time than GPUBICGSTAB for these matrices except for atmosmodd. These observations validate the effectiveness of the approximate inverse matrices that are obtained by HeuriSPAI.

Second, we test the effectiveness of HeuriSPAI by comparing it with a recent static SPAI algorithm suggested in He et al. (2020)(denoted by SSPAI) and a popular dynamic SPAI algorithm by Grote and Huckle(Grote and Huckle 1997) (denoted DSPAI) from the viewpoint of accelerating the convergence. The first eleven small matrices in Table 3 are used in this test. The small matrices are chosen for the following two reasons: (1) DSPAI is not suitable for large matrices; (2) they are the same as those in Grote and Huckle (1997). Similar to Grote and Huckle (1997), the preconditioned BICGSTAB is called to solve $Ax = b$, and stops when the residual error is less than $1e^{-8}$ or the number of iterations

Table 5 Iterations and execution time of two algorithms on RTX3090

Matrix	GPUBICGSTAB		GPUPBICGSTAB	
	Iter	Exe time	Iter	Exe time
Af23560	> 10000	/	291	0.902
Zhao2	> 10000	/	1269	0.352
Venkat01	> 10000	/	21	0.902
Imagesenor	> 10000	/	31	0.439
Apache2	4207	2.885	566	1.349
T2em	1581	1.285	573	1.084
Thermal2	6620	7.743	1399	4.424
Atmosmodd	241	0.621	78	0.653
Nv2	> 10000	/	38	14.813
G3_circuit	> 10000	/	324	1.206
Ss	> 10000	/	81	17.676
Stokes	> 10000	/	775	170.154

exceeds 10,000. Table 6 shows the convergence results of four algorithms. The second, third, fourth, and fifth columns are the convergence results without the preconditioner, and with the preconditioner that is obtained by SSPAI, and with the preconditioner that is obtained by DSPAI, and with the preconditioner that is obtained by HeuriSPAI, respectively.

As compared to SSPAI, for all test cases, the preconditioned BICGSTAB with the preconditioner that is obtained by HeuriSPAI has smaller number of iterations than that with preconditioner that is obtained by SSPAI. Especially, for sherman2 and pores_2, the preconditioned BICGSTAB with SSPAI cannot converge to the 10^{-8} residual error in 10,000 iterations while the preconditioned BICGSTAB with HeuriSPAI can. This verifies that HeuriSPAI is better than SSPAI. As compared to DSPAI, the preconditioned BICGSTAB with the preconditioner that is obtained by HeuriSPAI has smaller number of iterations than that with preconditioner that is obtained by DSPAI for all test matrices except for sherman3 and pores_2. Especially, for sherman2, the preconditioned BICGSTAB with DSPAI cannot converge to the 10^{-8} residual error in 10,000 iterations while the preconditioned BICGSTAB with HeuriSPAI can. This means that HeuriSPAI is effective.

3.2 Performance analysis

In this section, we first take GTX1070 to investigate the fraction of the total time spent in the *Init-HeuriSPAI*, *Compute-HeuriSPAI*, *Post-HeuriSPAI* stages in Fig. 5. We can observe that for all the matrices, the fractions of the *Init-HeuriSPAI* and *Post-HeuriSPAI* stages are at most $\frac{1}{10}$ and $\frac{1}{20}$, respectively. This further verifies that the time of HeuriSPAI is mainly attributed to the cost of the *Compute-HeuriSPAI* stage. Second, we take the *Compute-HeuriSPAI* stage to explore the ratio of its execution time on the CPU to its execution time on the GPU, as shown in Fig. 6. It can be seen that the ratios of the execution time on the CPU to the execution time on the GTX1070 range roughly from 41.38 to 63.33 for the 12 test matrices, and the average ratio is 51.05; the ratios of the execution time on the CPU to the execution time on the RTX3090 range roughly from 53.43 to 79.44 for the 12 test matrices, and the average ratio is 63.89. These results show that computing the preconditioner for our proposed HeuriSPAI has higher parallelism.

Table 6 Convergence results of all algorithms

Matrix	No precond	SSPAI	DSPAI	HeuriSPAI
Orsreg_1	347	94	47	27
Orsirr_1	1671	118	55	29
Orsirr_2	1039	163	45	30
Sherman1	391	54	41	26
Sherman2	> 10000	> 10000	> 10000	37
Sherman3	> 10000	235	72	108
Sherman4	100	34	28	26
Sherman5	2021	48	41	37
Pores_2	> 10000	> 10000	78	212
Pores_3	1597	141	118	47
Saylr4	4055	1474	285	163

3.3 Performance comparison

We evaluate the performance of HeuriSPAI by comparing it with several popular preconditioning algorithms, i.e., CSRILU0 in the CUSPARSE 11.1 library (denoted by CSRILU)(NVIDIA 2021), the incomplete SPAI preconditioning algorithm in the MAGMA 2.6.2 library (denoted by ISAI)(Anzt et al. 2018), a static SPAI preconditioning algorithm (denoted by S-VCL) and a dynamic SPAI preconditioning algorithm (denoted by D-VCL) in the ViennaCL

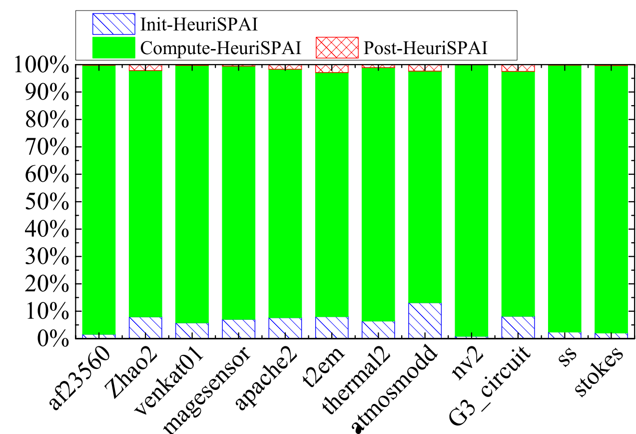

Fig. 5 The fraction of total time spent in the *Init-HeuriSPAI*, *Compute-HeuriSPAI*, *Post-HeuriSPAI* stages

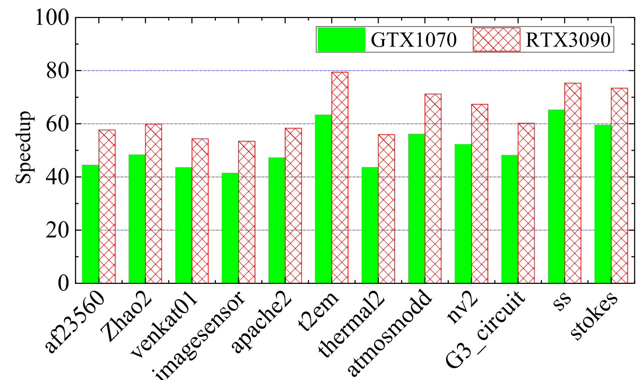

Fig. 6 Ratio of the execution time on CPU to the excution time on GPU

Table 7 Execution time of all preconditioning algorithms and GPUPBICGSTAB on GTX1070

Matrix	CSRILU+ GPUPBICGSTAB	ISAI+ GPUPBICGSTAB	S-VCL+ GPUPBICGSTAB	SSPAI+ GPUPBICGSTAB	HeuriSPAI+ GPUPBICGSTAB
Af23560	/	0.027	/	/	1.562
	/	0.316	/	/	0.397
	>10000	743	>10000	>10000	291
	/	0.343	/	/	1.959
Zhao2	0.337	/	1.380	/	0.091
	2.942	/	0.853	/	0.655
	7009	>10000	1848	>10000	1269
	3.279	/	2.233	/	0.746
Venkat01	1.185	N/A	3.692	1.128	1.358
	0.535	N/A	0.163	0.303	0.298
	11	N/A	48	35	21
	1.720	N/A	3.855	1.431	1.656
Imagesensor	/	0.073	/	0.338	0.780
	/	4.131	/	0.319	0.289
	>10000	4809	>10000	52	31
	/	4.204	/	0.657	1.069
Apache2	2.950	0.398	4.925	0.226	1.043
	8.296	7.044	8.142	3.594	2.812
	517	2138	2503	1090	566
	11.246	7.442	13.067	3.820	3.855
T2em	24.440	1.004	N/A	0.089	0.634
	2.657	2.756	N/A	2.718	2.753
	409	746	N/A	755	573
	27.097	3.760	N/A	2.807	3.387
Thermal2	5.234	0.594	/	0.423	2.901
	63.285	13.675	/	11.721	10.516
	2047	2094	>10000	2086	1399
	68.519	14.269	/	12.144	13.417
Atmosmodd	5.580	0.669	6.123	0.385	1.002
	2.031	0.985	0.549	1.065	0.650
	76	169	91	135	78
	7.611	1.654	6.672	1.450	1.652
Nv2	16.307	N/A	N/A	/	25.343
	64.807	N/A	N/A	/	1.101
	1072	N/A	N/A	>10000	38
	81.114	N/A	N/A	/	26.444
G3_circuit	4.881	0.698	/	0.161	1.018
	14.114	23.335	/	2.887	2.358
	303	3682	>10000	468	324
	18.995	24.033	/	3.048	3.376
Ss	14.791	N/A	N/A	/	29.012
	4.779	N/A	N/A	/	2.651
	79	N/A	N/A	>10000	81
	19.570	N/A	N/A	/	31.663
Stokes	214.791	N/A	N/A	/	223.134
	84.821	N/A	N/A	/	65.367
	1087	N/A	N/A	>10000	775
	299.612	N/A	N/A	/	288.501

Table 8 Execution time of all preconditioning algorithms and GPUPBICGSTAB on RTX3090

Matrix	CSRILU+ GPUPBICGSTAB	ISAI+ GPUPBICGSTAB	S-VCL+ GPUPBICGSTAB	SSPAI+ GPUPBICGSTAB	HeuriSPAI+ GPUPBICGSTAB
Af23560	/	0.014	/	/	0.781
	/	0.131	/	/	0.121
	>10000	743	>10000	>10000	291
	/	0.145	/	/	0.902
Zhao2	0.167	/	0.595	/	0.043
	1.576	/	0.512	/	0.309
	7009	>10000	1848	>10000	1269
	1.743	/	1.107	/	0.352
Venkat01	0.691	N/A	1.748	0.783	0.809
	0.296	N/A	0.206	0.101	0.093
	11	N/A	48	35	21
	0.987	N/A	1.954	0.884	0.902
Imagesensor	/	0.031	/	0.200	0.353
	/	2.087	/	0.110	0.086
	>10000	4809	>10000	52	31
	/	2.118	/	0.310	0.439
Apache2	1.780	0.162	2.061	0.130	0.432
	4.628	2.938	3.353	1.170	0.917
	517	2138	2503	1090	566
	6.408	3.100	5.414	1.300	1.349
T2em	6.983	1.015	N/A	0.044	0.138
	1.759	1.099	N/A	0.929	0.946
	409	746	N/A	755	573
	8.742	2.114	N/A	0.973	1.084
Thermal2	3.096	0.197	/	0.262	0.996
	35.188	6.081	/	3.755	3.432
	2047	2094	>10000	2086	1399
	38.284	6.278	/	4.017	4.424
Atmosmodd	3.329	0.233	3.148	0.232	0.404
	0.912	0.422	0.384	0.339	0.249
	76	169	91	135	78
	4.241	0.655	3.532	0.571	0.653
Nv2	9.059	N/A	N/A	/	14.079
	36.004	N/A	N/A	/	n0.734
	1072	N/A	N/A	>10000	38
	45.063	N/A	N/A	/	14.813
G3_circuit	3.073	0.329	/	0.094	0.403
	8.755	n9.365	/	0.994	0.803
	303	3682	>10000	468	324
	11.828	9.694	/	1.088	1.206
Ss	8.701	N/A	N/A	/	16.117
	2.811	N/A	N/A	/	1.559
	79	N/A	N/A	>10000	81
	11.512	N/A	N/A	/	17.676
Stokes	126.347	N/A	N/A	/	131.703
	47.123	N/A	N/A	/	38.451
	1087	N/A	N/A	>10000	775
	173.470	N/A	N/A	/	170.154

1.7.1 library(Rupp et al. 2016), and a recent sparse approximate inverse preconditioning algorithm (denoted by SSPAI) He et al. (2020). We choose CSRILU because CUSPARSE is an open popular library for NVIDIA GPUs and ILU0 is a classic incomplete factorization method and has been widely applied as the preconditioner. ISAI is chosen because MAGMA is an open popular library for GPU and mutlicore architectures and ISAI preconditioner is a new one. S-VCL, D-VCL and SSPAI are chosen because D-VCL is the only existing dynamic sparse approximate inverse preconditioning algorithm on GPU and S-VCL and SSPAI both are one of the latest static sparse approximate inverse preconditioning algorithms on GPU. GPUPBICGSTAB with CSRILU, GPUPBICGSTAB with S-VCL/D-VCL and GPUPBICGSTAB with ISAI are implemented using the functions in CUBLAS and CUSPARSE, ViennaCL, MAGMA, respectively. GPUPBICGSTAB with SSPAI is implemented based on CUBLAS and CUSPARSE. The last 12 large matrices in Table 3 are used for this test. Tables 7 and 8 show the comparison results of all algorithms on GTX1070 and RTX3090, respectively. In each table, for each matrix and the preconditioner, the first row is the execution time of the preconditioning algorithms, the second and third rows are the execution time of GPUPBICGSTAB and the number of iterations when GPUPBICGSTAB converges to the $1e^{-7}$ residual error in 10,000 iterations, and the fourth row is the total of the execution time of the preconditioning algorithm and GPUPBICGSTAB. If the number of iterations for GPUPBICGSTAB exceeds 10,000 for a matrix, the corresponding rows will be denoted by "/" except that the third row is denoted by ">10000". If GPUPBICGSTAB encounters the error that the size of system is too large for ISAI L or the floating point exception or the out-of-memory error, the four rows will be denoted by "N/A". The time unit is s.

From Tables 7 and 8, we observe that on the two GPUs, for the chosen 12 large matrices except for venkat01, ss, and stokes, HeuriSPAI has smaller execution time than CSRILU. Furthermore, the total execution time of HeuriSPAI and GPUPBICGSTAB is less than that of CSRILU and GPUPBICGSTAB for the chosen 12 large matrices except for ss. This verifies that HeuriSPAI is better than CSRILU in general for the test cases. Compared to ISAI, the total time of HeuriSPAI and GPUPBICGSTAB is less than that of ISAI and GPUPBICGSTAB, and GPUPBICGSTAB with HeuriSPAI has smaller number of iterations than GPUPBICGSTAB with ISAI for the 12 large matrices except for af23560. Especially, GPUPBICGSTAB with ISAI encounters the error that the size of system is too large for ISAI L for venkat01, nv2, ss, and storkes, and cannot convergence in 10,000 iterations for Zhao2 while GPUPBICGSTAB with HeuriSPAI can converge to the $1e^{-7}$ residual error in 10,000 iterations. This shows that HeuriSPAI usually has better behavior than ISAI for the test cases. GPUPBICGSTAB with D-VCL is not applicable for the 12 large matrices because of the out-of-memory error while GPUPBICGSTAB with HeuriSPAI can converge in 10,000 iterations. This further validates the fact that HeuriSPAI can alleviate the drawback of D-VCL. Because D-VCL always encounters the out-of-memory error for the 12 large matrices, its results are not shown in Tables 7 and 8. As compared to S-VCL, whether the number of iterations or the total time of the preconditioner and GPUPBICGSTAB, HeuriSPAI outperforms S-VCL. As compared to SSPAI, GPUPBICGSTAB with HeuriSPAI can converge to the $1e^{-7}$ residual error in 10,000 iterations for all test cases. However, for the five matrices such as af23560, Zhao2, nv2, ss, and stokes, GPUPBICGSTAB with SSPAI cannot converge to the $1e^{-7}$ residual error in 10,000 iterations. For venkat01, imagesensor, apache2, t2em, thermal2, atmosmodd, and G3_circuit, GPUPBICGSTAB with HeuriSPAI has much smaller number of iterations than GPUPBICGSTAB with SSPAI, and although the total time of HeuriSPAI and GPUPBICGSTAB is more than that of SSPAI and GPUPBICGSTAB, their difference are slight. Therefore, we can conclude that as compared to SSPAI, HeuriSPAI can in general decrease the iteration count of iterative solvers significantly, and can alleviate the drawback that SSPAI cannot converge for some matrices.

4 Conclusion

In this paper, we present a parallel heuristic dynamic sparse approximate inverse (SPAI) preconditioning algorithm on GPU, called HeuriSPAI. HeuriSPAI fuses the advantages of static and dynamic SPAI preconditioning algorithms, and alleviates the drawbacks of the existing dynamic SPAI preconditioning algorithms on GPU that can encounter the out-of-memory error for large matrices. Experimental results validate the effectiveness and high parallelism of the proposed HeuriSPAI.

Next, we will further do research in this field, and apply the proposed HeuriSPAI to more practical problems to improve it.

Acknowledgements This work was funded by the Natural Science Foundation of China under grant number 61872422.

Declarations

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

Anzt, H., Gates, M., Dongarra, J., et al.: Preconditioned Krylov solvers on GPUs. Parallel Comput. **68**, 32–44 (2017)

Anzt, H., Huckle, T.K., Brackle, J., Dongarra, J.: Incomplete sparse approximate inverses for parallel preconditioning. Parallel Comput. **71**, 1–22 (2018)
Benzi, M., Meyer, C.D., Tuma, M.: A sparse approximate inverse preconditioner for the conjugate gradient method. SIAM J. Sci. Comput. **17**(5), 1135–1149 (1996)
Benzi, M., Cullum, J., Tuma, M.: Robust approximate inverse preconditioning for the conjugate gradient method. SIAM J. Sci. Comput. **22**(4), 1318–1332 (2000)
Bernaschi, M., Bisson, M., Fantozzi, C., Janna, C.: A factored sparse approximate inverse preconditioned conjugate gradient solver on graphics processing units. SIAM. J. Sci. Comput. **38**(1), C53–C72 (2016)
Brandes, T., Arnold, A., Soddemann, T., Reith, D.: CPU vs. GPU - performance comparison for the Gram-Schmidt algorithm. Eur Phys J Spec Top **210**(1), 73–88 (2012)
Chow, E.: A priori sparsity patterns for parallel sparse approximate inverse preconditioners. SIAM J. Sci. Comput. **21**(5), 1804–1822 (2000)
Cosgrove, J.D.F., Diaz, J.C., Griewank, A.: Approximate inverse preconditioning for sparse linear systems. Int. J. Comput. Math. **44**(1–2), 91–110 (1992)
Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM T. Math. Software **38**(1), 1–25 (2011)
Dehnavi, M.M., Fernández, D. M., Gaudiot, J. L., Giannacopoulos, D. D.: Parallel sparse approximate inverse preconditioning on graphic processing units. IEEE T. Parall. Distr **24**(9), 852–1861 (2013)
Duin, A.C.N.V.: Scalable parallel preconditioning with the sparse approximate inverse of triangular systems. SIAM J. Matrix Anal. Appl. **20**(4), 987–1006 (1999)
Ferronato, M., Janna, C., Pini, G.: A generalized block FSAI preconditioner for nonsymmetric linear systems. J. Comput. Appl. Math. **256**, 230–241 (2014)
Gao, J., Liang, R., Wang, J.: Research on the conjugate gradient algorithm with a modified incomplete Cholesky preconditioner on GPU. J. Parallel Distr. Com. **74**(2), 2088–2098 (2014)
Gao, J., Wu, K., Wang, Y., Qi, P., He, G.: GPU-accelerated preconditioned GMRES method for two-dimensional Maxwell's equations. Int. J. Comput. Math. **94**(10), 2122–2144 (2017)
Gao, J., Chen, Q., He, G.: A thread-adaptive sparse approximate inverse preconditioning algorithm on multi-GPUs. Parallel Comput. **101**, 102724 (2021)
Grote, M., Huckle, T.: Parallel preconditioning with sparse approximate inverses. SIAM J. Sci. Comput. **18**(3), 838–853 (1997)
He, G., Yin, R., Gao, J.: An efficient sparse approximate inverse preconditioning algorithm on GPU. Concurr. Comput.-Pract. Exp. **32**(7), e5598 (2020)
Jia, Z., Zhu, B.: A power sparse approximate inverse preconditioning procedure for large sparse linear systems. Numer. Linear Algebr. **16**(4), 259–299 (2009)
Kolotilina, L.Y., Yeremin, A.Y.: Factorized sparse approximate inverse preconditioning I. Theory. SIAM J. Matrix Anal. Appl. **14**(1), 45–58 (1993)
NVIDIA, CUBLAS Library, v11.1 (2021)
NVIDIA, CUDA C Programming Guide, v11.1 (2021)
NVIDIA, CUSPARSE Library, v11.1 (2021)
Rupp, K., Tillet, R., Rudolf, F.: ViennaCL–linear algebra library for multi- and many-core architectures. SIAM J. Sci. Comput. **38**(5), S412–S439 (2016)
Saad, Y.: Iterative Methods for Sparse Linear Systems, second version. SIAM, Philadelphia, PA (2003)
Saad, Y., Schultz, M.H.: GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM J. Sci. Stat. Comput. **7**(3), 856–869 (1986)
van der Vorst, H.A.: A vectorizable variant of some ICCG methods. SIAM J. Sci. Stat. Comput. **3**(3), 350–356 (1982)
van der Vorst, H.A.: Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of non-symmetirc linear systems. SIAM J. Sci. Stat. Comput. **12**(3), 631–644 (1992)

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.