BenchCouncil Transactions on Benchmarks, Standards and Evaluations 2 (2022) 100087

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Research Article

Optimizing the sparse approximate inverse preconditioning algorithm on GPU☆

Xinyue Chu, Yizhou Wang, Qi Chen, Jiaquan Gao *

Jiangsu Key Laboratory for NSLSCS, School of Computer and Electronic Information, Nanjing Normal University, Nanjing 210023, China

ARTICLE INFO

Keywords:
Sparse approximate inverse
Preconditioning
CUDA
GPU

ABSTRACT

In this study, we present an optimization sparse approximate inverse (SPAI) preconditioning algorithm on GPU, called GSPAI-Opt. In GSPAI-Opt, it fuses the advantages of two popular SPAI preconditioning algorithms, and has the following novelties: (1) an optimization strategy is proposed to choose whether to use the constant or non-constant thread group for any sparse pattern of the preprocessor, and (2) a parallel framework of optimizing the SPAI preconditioner is proposed on GPU, and (3) for each component of the preconditioner, a decision tree is established to choose the optimal kernel of computing it. Experimental results validate the effectiveness of GSPAI-Opt.

1. Introduction

Given their many-core structures, graphic processing units (GPUs) have become an important resource for scientific computing in recent years. Following the introduction of the programming interfaces such as the compute unified device architecture (CUDA) by NVIDIA in 2007 [1], GPUs have been increasingly used as tools for high-performance computation in many fields [2–8].

Sparse approximate inverse (SPAI) preconditioners based on the Frobenius norm minimization have proven to be effective in improving the convergence of iterative methods based on Krylov subspaces, e.g., the generalized minimal residual method (GMRES) [9] and the biconjugate gradient stabilized method (BiCGSTAB) [10]. However, due to the high cost of constructing the SPAI preconditioners, many researchers have attempted to accelerate the SPAI preconditioner construction on GPU. Gao et al. follow Chow's work [11], and use a sparse approximate inverse of A as the preconditioner in [12]. Rupp et al. [13] show several static and dynamic SPAI implementations on GPU. In [14], Dehnavi et al. propose a static SPAI preconditioner on GPU called GSAI. Recently, He and Gao et al. [15] propose a GPU-based static SPAI preconditioning algorithm called SPAI-Adaptive, and verify the effectiveness of SPAI-Adaptive for large-scale matrices. However, when the number of nonzero entries in each column of the preconditioner has significant difference, the performance of SPAI-Adaptive is greatly decreased. Furthermore, He and Gao et al. [16] present a sorted static SPAI preconditioning algorithm, called GSPAI-Adaptive, in order to avoid the drawback of SPAI-Adaptive.

SPAI-Adaptive and GSPAI-Adaptive both can be applied to large-scale matrices, and have their own advantages. When the difference in the nonzero number of each column of the preconditioner is small, the performance of SPAI-Adaptive is generally better than that of GSPAI-Adaptive; when the nonzero number of each column of the preconditioner has significant difference, SPAI-Adaptive has worse performance than GSPAI-Adaptive. For example, assuming that $n2_k$ is the nonzero number of the kth column of the preconditioner, $n2max = \max_k\{n2_k\}$, and $n2avg = \sum_{k=1}^{n} n2_k/n$, where n is the row number of the preconditioner, we take two integers α and β, which satisfy $2^{\alpha-1} < n2max \leqslant 2^{\alpha}$ and $2^{\beta-1} < n2max \leqslant 2^{\beta}$, respectively. If $\alpha = \beta$, we say that the difference in the nonzero number of each column of the preconditioner is small; if $\alpha - \beta \geqslant 3$, we say that the nonzero number of each column of the preconditioner has significant difference. However, when the difference is large but not significant, which one of SPAI-Adaptive and GSPAI-Adaptive has better performance? For example, $1 \leqslant \alpha - \beta < 3$. There are no conclusions in [15,16].

Inspired by these observations, we further investigate how to highly optimize the static SPAI on GPU in this paper. Utilizing the advantages of SPAI-Adaptive and GSPAI-Adaptive, we propose an optimized SPAI preconditioning algorithm on GPU, called GSPAI-Opt. Compared to SPAI-Adaptive and GSPAI-Adaptive, the proposed algorithm has the following distinct characteristics:

- First, an optimization strategy is presented. Using this strategy, for a given sparsity pattern of the preconditioner, we can obtain the optimization scheme of choosing whether to use the constant or nonconstant thread-group size to calculate the preconditioner.
- Second, when the constant thread-group size is applied, for each one of main components of the preconditioner such as finding indices I and J, constructing the local submatrix, decomposing the

☆ The research has been supported by the Natural Science Foundation of China under grant number 61872422.
* Corresponding author.
E-mail addresses: 2316607219@qq.com (X. Chu), 1966224230@qq.com (Y. Wang), 1337223917@qq.com (Q. Chen), springf12@163.com (J. Gao).

https://doi.org/10.1016/j.tbench.2023.100087
Received 11 October 2022; Received in revised form 26 February 2023; Accepted 26 February 2023
Available online 3 March 2023

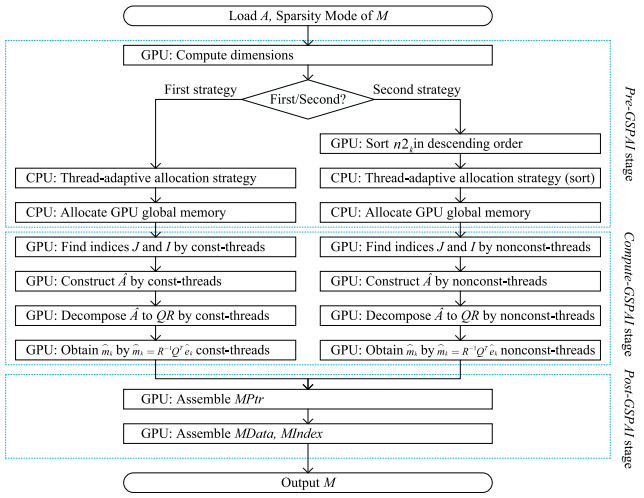

Fig. 1. Parallel framework of GSPAI-Opt.

local submatrix into QR, and solving the upper triangular linear system, a decision tree is established to choose the optimization kernel of calculating it.

- Third, when using the nonconstant thread-group size, for each one of some components of the preconditioner such as decomposing the local submatrix into QR and solving the upper triangular linear system, a decision tree is constructed to choose the optimization kernel to calculate it.
- Finally, GSPAI-Opt can apply to any sparsity pattern of the preconditioner, not just the same sparsity pattern as A.

The experimental results show that GSPAI-Opt is effective, and efficiently fuses the advantages of SPAI-Adaptive and GSPAI-Adaptive, and outperforms the static SPAI preconditioning algorithm in the ViennaCL library [13], the recent SPAI-Adaptive [15] and GSPAI-Adaptive [16].

2. Optimizing SPAI on GPU

We present an optimization sparse approximate inverse preconditioning algorithm on GPU, called GSPAI-Opt. Fig. 1 lists the parallel framework of GSPAI-Opt, which is composed of the following stages.

- *Pre-GSPAI* stage: Compute the dimensions, choose whether to allocate the constant thread-group size or nonconstant thread-group size for each column of the preconditioner according to the proposed optimization strategy, and allocate the global memory of GPU;
- *Compute-GSPAI* stage: Find indices J_k and I_k, construct local submatrix $\widehat{A}_k$, decompose $\widehat{A}_k$ into $Q_k R_k$, and solve $R_k \widehat{m}_k = Q_k^T \widehat{e}_k$;
- *Post-GSPAI* stage: Assemble the preconditioner M in the compressed sparse column (CSC) storage format.

Based on the sparsity pattern of the preconditioner, when the thread allocation strategy with the constant thread-group size is more suitable for computing the preconditioner, the thread-adaptive allocation strategy (First strategy) proposed in [15] is adopted; otherwise, the thread-adaptive allocation strategy with the nonconstant thread-group size (Second strategy) proposed in [16] is utilized. Given a matrix, should we use the first strategy or the second strategy? Here we present a selection method, whose main procedure is shown in Fig. 2.

Let us illustrate the selection method in Fig. 2 by apache2. For apache2, we have $n2max = 8$ and $n2avg = 6.74$. Obviously, $n2max$, $n2avg \in (2^2, 2^3]$, and $\alpha = \beta = 3$. Based on the selection method in Fig. 2, the first strategy is chosen.

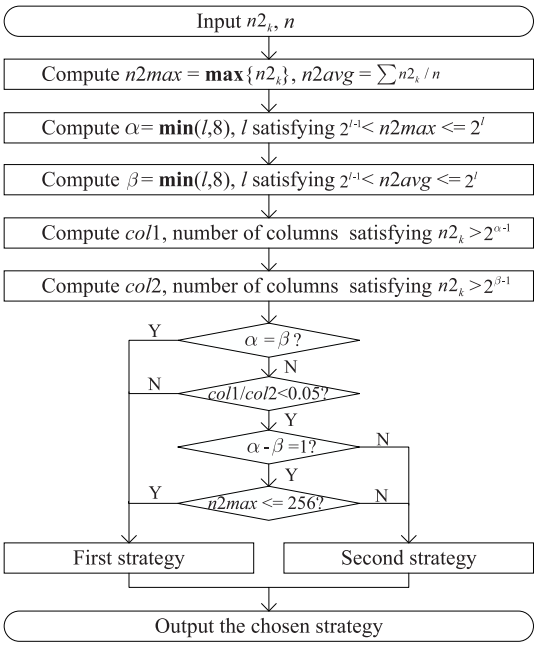

Fig. 2. Main procedure of selecting the First/Second strategy.

2.1. Pre-GSPAI stage

First, we compute the dimensions of all local submatrices. When computing m_k (one column of M), $k = 1, 2, \ldots, n$, the dimensions of the local submatrices ($n1_k$, $n2_k$) constructed for each column of the preconditioner are usually different. To simplify the accesses of data in the memory and enhance the coalescence, the dimensions of all local submatrices are uniformly defined as ($n1max$, $n2max$). Here $n1max = \max_k\{n1_k\}$ and $n2max = \max_k\{n2_k\}$.

Next, we choose whether to use the constant or nonconstant thread-group size for each column of the preconditioner. GSPAI-Opt fuses the advantages of SPAI-Adaptive [15] and GSPAI-Adaptive [16]. For SPAI-Adaptive, a thread-adaptive allocation strategy with the constant thread-group size is presented, and for GSPAI-Adaptive, a thread-adaptive allocation strategy with the nonconstant thread-group size is presented. For the convenience of readers, in the following contents, we introduce them respectively.

Thread-adaptive allocation strategy with the constant thread-group size: The optimized number of threads q is obtained by the following formula:

$$q = \min(2^s, nt), \tag{1}$$

s.t.

$$2^{s-1} < n2max \leqslant 2^s. \tag{2}$$

Here nt is the number of threads per block, and q threads are grouped into a thread group.

Thread-adaptive allocation strategy with the nonconstant thread-group size: First, for each $n2_k$, $k = 1, 2, \ldots, n$, the number of threads q_k assigned to the kth column of the preconditioner is computed by the following formula:

$$q_k = \min(2^s, nt), \tag{3}$$

s.t.

$$2^{s-1} < n2_k \leqslant 2^s. \tag{4}$$

Second, all q_k values are sorted in descending order. Finally, the thread-group size of each block is assigned by the procedure shown in Fig. 3.

Input: q, nt, n
Output: $WSize$, $BCol$, $blocks$
01. $i \leftarrow 0$; $blocks \leftarrow 0$; $BCol[0] \leftarrow 0$;
02. **while** $i < n$
03. $WSize[blocks] \leftarrow q[i]$;
04. i += $nt/WSize[blocks]$;
05. $blocks$++;
06. $BCol[blocks] \leftarrow i$;
07. **end while**

Fig. 3. Main procedure of assigning the thread-group size.

Table 1
Arrays used in GSPAI-Opt.

Array	Size	Type	Array	Size	Type
AData	nonzeros	double	$\widehat{m}$	ns × *n2max*	double
AIndex	nonzeros	integer	$\widehat{A}$	ns × *n1max* × *n2max*	double
APtr	*n*	integer	*R*	ns × *n2max* × *n2max*	double
RCol	*n*	integer	*I*	ns × *n1max*	integer
atomic	*n*	integer	*iPTR*	ns	integer
WSize	*blocks*	integer	*J*	ns × *n2max*	integer
BCol	*blocks*	integer	*jPTR*	ns	integer

Finally, we allocate global memory for arrays in Table 1, and *RCol*, *BCol*, and *WSize* values are transferred to the GPU global memory if the second strategy is applied.

2.2. Compute-GSPAI stage

Finding indices: This part is to find indices J and I by the constant/nonconstant thread-group size.

(1) ***Finding J and I by the constant thread-group size***: In this case, the thread-group size that is used to find J and I is same in all blocks. For the kernel that finds J, the threads inside each thread group read one column of the sparsity pattern M in parallel and store them to one subset of J. And then on this basis of J, we implement the construction of I. We establish a decision tree to find I based on the GPU feature parameters. Utilizing the decision tree, an optimized kernel for finding I is obtained for any given $n2max$ and $n1max$. Assume that the threads per block are 256 and NIVIDA GTX1070 GPU is used, Fig. 4 shows a segment of the decision tree for finding I. Here $sharedSize$ = number of columns of the preconditioner computed in a thread block × upper boundary closest to $n1max$. For example, when $n1max \leqslant 8$, $sharedSize = 32 \times 8$ and cuFindIBySharedMemory kernel with shared memory of 256 size is used. In the cuFindIBySharedMemory kernel, each thread group finds one subset of I, e.g., I_k, which mainly includes the following steps. First, the threads in the thread group load the row indices of the first column referenced in one subset of J, e.g., J_k, to shared memory sI. Second, the index vectors of successive columns referenced by J_k are compared in parallel with values in sI and new indices are appended to sI by utilizing the atomic operations. Third, inside the thread group, the indices of sI are sorted in ascending order in parallel. Finally, the indices of sI are copied to I_k. cuFindI kernel is similar to cuFindIBySharedMemory kernel except that the operations are executed on global memory instead of shared memory.

(2) ***Finding J and I by the nonconstant thread-group size***: The thread-group size of finding J and I is same in a block while it is usually different for different blocks. For the kernel that finds J, the threads inside each thread group read one column of the sparsity pattern M in parallel and store them to one subset of J. The main procedure of the kernel that finds I is as same as that in [16]. Each thread group is assigned to find one subset of I, e.g., I_k, which includes the following three stages. In the first stage, the thread group obtains the thread-group size $warpSize$. In the second stage, the row indices of the first column referenced in J_k are first loaded into I_k, and the row index vectors of successive columns that are referenced by J_k are calculated in parallel with values in I_k, and the new indices are appended to I_k by utilizing the atomic operations. In the third stage, the indices in I_k are sorted in ascending order in parallel.

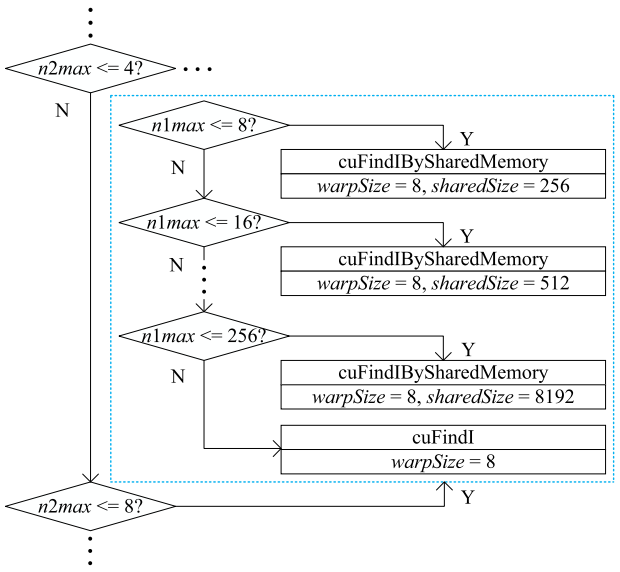

Fig. 4. A segment of the decision tree of using constant threads to find I.

Constructing the local submatrix: Using J and I obtained above, the local submatrix set, $\widehat{A}$, is computed by the constant/nonconstant thread-group size.

(1) ***Constructing the local submatrix by the constant thread-group size***: Each thread group is assigned to compute one subset of $\widehat{A}$, e.g., $\widehat{A}_k$, and all thread groups are the same size. Based on the GPU feature parameters, we establish a decision tree for constructing $\widehat{A}$. For any given $n2max$ and $n1max$, an optimized kernel for constructing $\widehat{A}$ is achieved by using the decision tree. For example, on NIVIDA GTX1070 GPU, assume that the threads per block are 256, Fig. 5 shows a segment of the decision tree for constructing $\widehat{A}$. When $4 < n2max \leqslant 8$, corresponding to different $n1max$, cuComputeTildeABySharedMemory kernel with shared memory of $sharedSize$ size and cuComputeTildeA kernel with non shared memory are selected. The main procedure of cuComputeTildeABySharedMemory kernel is listed as follows. For the thread group that calculates $\widehat{A}_k$, all threads in the thread group first read values in I_k into shared memory sI in parallel, and $\widehat{A}_k$ is established on the global memory by loading columns that are indexed by J_k and matching them to sI in parallel. cuComputeTildeA kernel is similar to cuComputeTildeABySharedMemory kernel except that I is executed on global memory instead of shared memory.

(2) ***Constructing the local submatrix by the nonconstant thread-group size***: In this case, each thread group is assigned to calculate one subset of $\widehat{A}$, e.g., $\widehat{A}_k$, and the thread-group size is same in a block but it can be different for different block. The main procedure of the kernel that constructs $\widehat{A}$ is as same as that in [16].

Decomposing the local submatrix into QR: This part is used to decompose the local submatrix into QR by the constant/nonconstant thread-group size.

(1) ***Decomposing the local submatrix into QR by the constant thread-group size***: The thread-group size of decomposing the local submatrix into QR is same in all blocks. Based on the GPU feature parameters, we establish a decision tree for decomposing the local submatrix into QR. For example, on NIVIDA GTX1070 GPU, assume that the threads per block are 256, Fig. 6 shows a segment of the decision tree for decomposing the local submatrix into QR. When $4 < n2max \leqslant 8$, two shared memories $sharedR$ and $sharedQ$ are

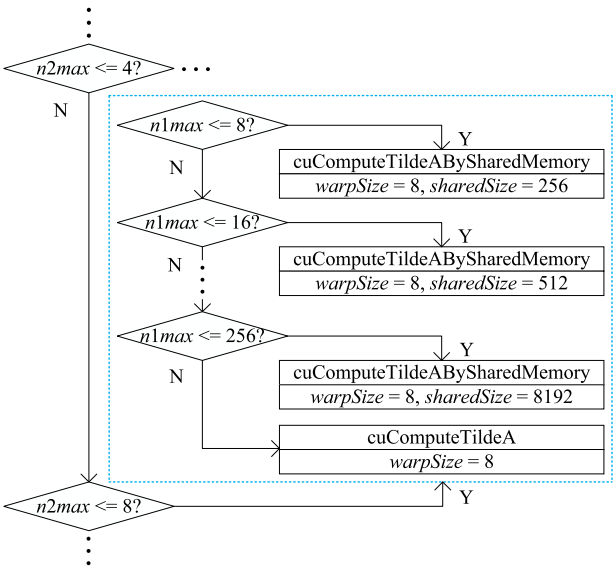

Fig. 5. A segment of the decision tree of using constant threads to construct $\hat{A}$.

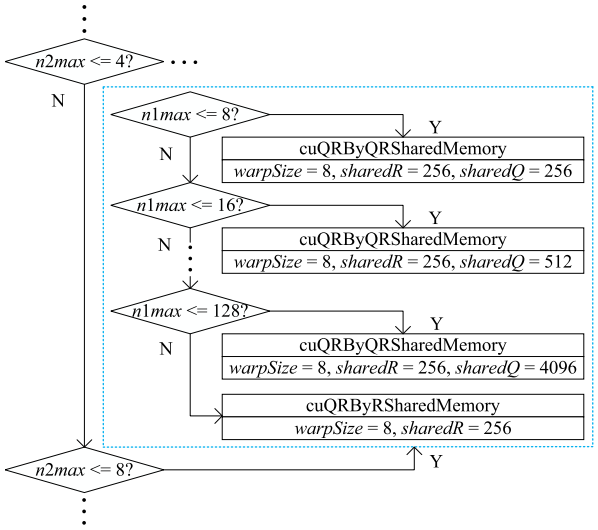

Fig. 6. A segment of the decision tree of using constant threads to decompose the local submatrix into QR.

used in the optimized kernel. Here the size of $sharedQ$ is related to $n1max$. In the cuQRByQRSharedMemory kernel, each thread group is responsible for one QR decomposition. In a thread group, the local submatrix, e.g., $\hat{A}_k$, is decomposed into QR by the following four steps at each iteration i. In the first step, the threads read the ith column of Q_k into shared memory sQ in parallel. In the second step, the ith row of the upper triangle matrix R_k are computed in parallel and are put into shared memory sR. In the third step, the column i of Q_k and sQ are concurrently normalized, and the projection factors R_k and sR are calculated. In the fourth step, the values of all columns of Q_k are updated by using shared memory sQ and sR in parallel. cuQRByRSharedMemory kernel is similar to cuQRByQRSharedMemory kernel except the shared memory sQ is not utilized.

(2) *Decomposing the local submatrix into QR by the nonconstant thread-group size*: The thread-group size of decomposing the local submatrix into QR is same in a block while it is usually different for different blocks. We establish a decision tree for decomposing the local submatrix into QR. For example, on NIVIDA GTX1070 GPU, the decision tree for decomposing the local submatrix into QR is shown

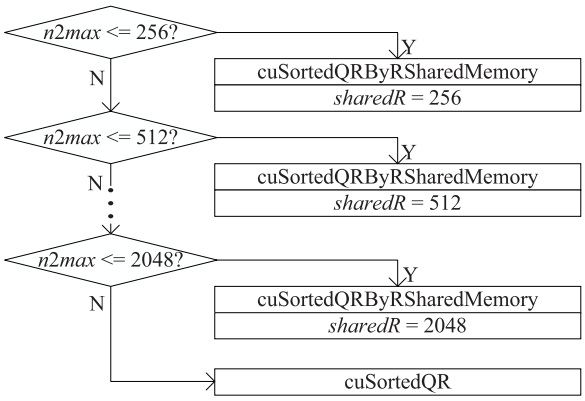

Fig. 7. Decision tree of using nonconstant threads to decompose the local submatrix into QR.

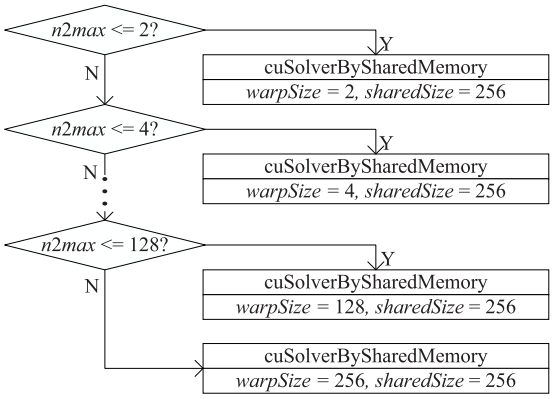

Fig. 8. Decision tree of using constant threads to solve the upper triangular linear system.

in Fig. 7 when the threads per block are 256. Obviously, utilizing the decision tree, an optimized kernel cuSortedQRByRSharedMemory corresponding to shared memory of $sharedR$ size or cuSortedQR kernel is chosen for a given $n2max$ value. The main procedure of cuSortedQRByRSharedMemory kernel is as same as that in [16]. cuSortedQR kernel is similar to cuSortedQRByRSharedMemory kernel except that the shared memory sR is not utilized.

Solving the upper triangular linear system: The values of $\hat{m}_k = R_k^{-1} Q_k^T \hat{e}_k$ are computed by the constant/nonconstant thread-group size.

(1) *Solving the upper triangular linear system by the constant thread-group size*: Each thread group computes one subset of $\hat{m}$ by solving an upper triangular linear system, and the thread-group size is same in all blocks. In this case, assume that the threads per block are 256, the decision tree for solving the upper triangular linear system is shown in Fig. 8. For any given $n2max$ value, an optimized kernel, cuSolverBySharedMemory with shared memory of 256 size and thread-group size of $warpSize$, is chosen. In the cuSolverBySharedMemory kernel, each thread group calculates a subset of $\hat{m}$, e.g., $\hat{m}_k$, and its procedure includes two steps. First, Calculate $Q_k^T \hat{e}_k$ in parallel and save the result to the shared memory xE. Second, the values of $\hat{m}_k$ are obtained by solving the upper triangular linear system $R_k \hat{m}_k = xE$, in parallel.

(2) *Solving the upper triangular linear system by the nonconstant thread-group size*: Each thread group is responsible for obtaining a subset of $\hat{m}$ by solving an upper triangular linear system, and the thread-group size is same inside a block but it can be different for different blocks. A decision tree is established to solve the upper triangular linear system. For example, Fig. 9 lists the decision tree for solving the upper triangular linear system on NIVIDA GTX1070 GPU. For any

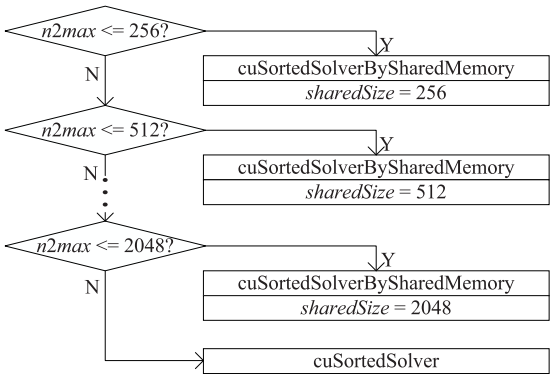

Fig. 9. Decision tree of using nonconstant threads to solve the upper triangular linear system.

Table 2
Overview of GPUs.

Hardware	GTX1070	A40
Cores	1920	10752
Clock speed (GHz)	1.506	1.305
Memory type	GDDR5	GDDR6
Memory size (GB)	8	48
Max-bandwidth (GB/s)	256	696
Compute capability	6.1	8.6

given $n2max$ value, we always choose an optimized kernel, which may be a cuSortedSolverBySharedMemory kernel that uses shared memory of $sharedSize$ size, or a cuSortedSolver kernel. The main procedure of cuSortedSolverBySharedMemory kernel is as same as that in [16]. cuSortedSolver kernel is similar to cuSortedSolverBySharedMemory kernel except that the shared memory xE is not used.

2.3. Post-GSPAI stage

In the *Post-GSPAI* stage, the preconditioner M is assembled in the CSC storage format which contains three arrays of $MPtr$, $MIndex$ and $MData$. Fig. 10 illustrates the procedure of assembling these arrays. First, *MPtr* is assembled utilizing *jPTR*. Second, $MData$ and $MIndex$ are assembled using $\hat{m}$ and J. In order to reduce the cost of array transfer, we assemble all arrays mentioned above on the GPU memory, and each thread group is responsible for generating one $\hat{m}_k$ to $MData$ and one J_k to $MIndex$.

3. Experimental results

In this section, we take two NVIDIA GPUs (GTX1070 and A40) shown in Table 2 to evaluate the performance of GSPAI-Opt. The test matrices are listed in Table 3, which are chosen from the SuiteSparse Matrix Collection [17], and have been widely used in some publications [14–16]. Table 3 summarizes the information of the sparse matrices, including the name, kind, number of rows, total number of nonzeros, average number of nonzeros, maximum number of nonzero entries

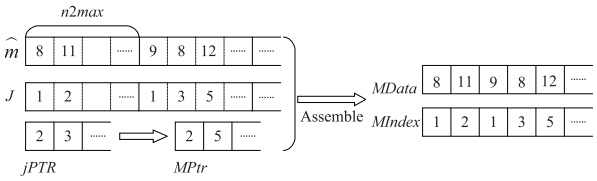

Fig. 10. Assemble M.

of columns, and minimum number of nonzero entries of columns. The matrices in Table 3 are chosen due to the following reasons. The matrices such as cbuckle, ASIC_320ks, power9, and Fault_639 are chosen to test whether the second strategy is chosen when the nonzero number of each column of the preconditioner has significant difference ($\alpha - \beta \geqslant 3$). The matrices such as 2cubes_sphere, offshore, apache2, G3_circuit are chosen to test whether the first strategy is chosen when the difference in the nonzero number of each column of the preconditioner is small ($\alpha = \beta$). The matrices such as msdoor and thermal2 are chosen to test whether the predicted strategy is well matched with the measured strategy when the difference in the nonzero number of each column of the preconditioner is large but not significant ($1 \leqslant \alpha - \beta < 3$). The source codes are compiled and executed using the CUDA toolkit 11.1 [18].

3.1. Accuracy of selection method

We take GTX1070 to test the accuracy of the proposed selection method of using the first strategy (denoted by S1) or the second strategy (denoted by S2). The sparse pattern of the preconditioner is a priori, so we test the accuracy in two popular patterns [14–16], $(E + |A|)^k, k = 1, 2$. The matrices in Table 3 are used as the test matrices. For all test matrices, both the optimal strategy predicted by the proposed selection method and the strategy obtained from actual tests are shown in Table 4. Note that if $|t1 - t2| / \max(t1, t2) \leqslant 0.05$, both S1 and S2 can be considered as the measured optimization strategy; otherwise, the strategy corresponding to $\min(t1, t2)$ is chosen as the measured optimization one. Here $t1$ and $t2$ are the time of constructing the preconditioner using S1 and S2, respectively. We can observe that for the two sparsity patterns, the estimated and measured optimal strategies are matched very well for the test cases. This verifies good accuracy of our proposed selection method.

3.2. Performance comparison

In order to test the effectiveness of our proposed GSPAI-Opt, we take the sparsity pattern $(E + |A|)$ to compare it with a static SPAI preconditioning algorithm in ViennaCL (denoted by SSPAI-VCL) [13], and two recent SPAI preconditioning algorithms SPAI-Adaptive [15] and GSPAI-Adaptive [16] on GTX1070 and A40, and their comparison results are listed in Tables 5 and 6, respectively. Moreover, since SSPAI-VCL cannot be suitable for the sparsity pattern $(E + |A|)^2$, only the comparison results of SPAI-Adaptive, GSPAI-Adaptive and GSPAI-Opt on two GPUs are shown in Table 7. In each table, for any matrix and

Table 3
Descriptions of test matrices.

Name	Kind	Rows	Nonzeros	Avg	Max	Min
cbuckle	Structural	13,681	676,515	49.45	600	26
2cubes_sphere	Electromagnetics	101,492	1,647,264	16.23	31	5
offshore	Electromagnetics	259,789	4,242,673	16.33	31	5
ASIC_320ks	Circuit simulation	321,671	1,316,085	4.09	210	1
apache2	Structural	715,176	4,817,870	6.74	8	4
G3_circuit	Circuit simulation	1,585,478	7,660,826	4.83	6	2
power9	Semiconductor device	155,376	1,887,730	12.15	627	1
msdoor	Structural	415,863	19,173,163	46.10	77	1
thermal2	Thermal	1,228,045	8,580,313	6.99	11	1
Fault_639	Structural	638,802	27,245,944	42.65	267	1

Table 4
Predicted and measured sparsity pattern.

Matrix	$(E+\|A\|)$		$(E+\|A\|)^2$	
	Predicted	Measured	Predicted	Measured
cbuckle	S2	S2	S2	S2
2cubes_sphere	S1	S1	S1	S1/S2
offshore	S1	S1	S1	S1/S2
ASIC_320ks	S2	S2	S2	S2
apache2	S1	S1	S1	S1
G3_circuit	S1	S1	S1	S1
power9	S2	S2	S2	S2
msdoor	S1	S1/S2	S1	S1/S2
thermal2	S1	S1	S1	S1
Fault_639	S2	S2	S2	S2

Table 5
Comparison of four algorithms with $(E+|A|)$ on GTX1070.

Matrix	SSPAI-V	SPAI-A	GSPAI-A	GSPAI-Opt
cbuckle	N/A	7.976	2.046	1.815
	N/A	0.362	0.356	0.348
	N/A	96	96	96
	N/A	8.338	2.402	**2.163**
2cubes_sphere	7.278	0.833	0.697	0.539
	0.025	0.300	0.296	0.294
	5	4	4	4
	7.303	1.133	0.993	**0.833**
offshore	20.468	2.177	2.052	1.380
	0.053	0.323	0.324	0.327
	12	5	5	5
	20.521	2.500	2.376	**1.707**
ASIC_320ks	N/A	5.000	1.398	0.846
	N/A	0.347	0.342	0.339
	N/A	10	10	10
	N/A	5.347	1.740	**1.185**
apache2	5.722	0.238	0.328	0.222
	7.963	3.583	3.574	3.585
	2503	1090	1090	1090
	13.685	3.821	3.902	**3.807**
G3_circuit	/	0.148	0.170	0.148
	/	2.887	2.881	2.885
	>10 000	468	468	468
	/	3.035	3.051	**3.033**
power9	N/A	4.504	10.620	2.848
	N/A	0.436	0.435	0.418
	N/A	37	37	37
	N/A	4.940	11.055	**3.266**
msdoor	N/A	59.794	21.374	20.206
	N/A	5.378	5.373	5.442
	N/A	892	892	892
	N/A	65.172	26.747	**25.648**
thermal2	/	0.401	0.527	0.340
	/	11.700	11.696	11.701
	>10 000	2086	2086	2086
	/	12.101	12.223	**12.041**
Fault_639	N/A	185.893	42.994	37.524
	N/A	10.032	10.022	10.013
	N/A	1226	1226	1226
	N/A	195.925	53.016	**47.537**

any given preconditioner, the first two rows are the execution time of the preconditioning algorithm and GPUPBICGSTAB, respectively, and the third row is the iteration number of GPUPBICGSTAB, and the fourth row is the total of the first two rows; if the iteration number of GPUPBICGSTAB is more than 10,000, we record the number of iterations ">10 000" in the third row, and the other rows that record the time are represented by "/"; if the out-of-memory error for GPUPBICGSTAB is encountered, all rows will be denoted by "N/A". The time unit is second (s), and the minimum value of the fourth row for each matrix is marked in the red font. For the convenience, SSPAI-VCL + GPUPBICGSTAB, SPAI-Adaptive + GPUPBICGSTAB, GSPAI-Adaptive + GPUPBICGSTAB and GSPAI-Opt + GPUPBICGSTAB are denoted by SSPAI-V, SPAI-A, GSPAI-A and GSPAI-Opt, respectively.

Table 6
Comparison of four algorithms with $(E+|A|)$ on A40.

Matrix	SSPAI-V	SPAI-A	GSPAI-A	GSPAI-Opt
cbuckle	N/A	3.717	1.167	0.878
	N/A	0.272	0.318	0.236
	N/A	96	96	96
	N/A	3.989	1.485	**1.114**
2cubes_sphere	4.952	0.336	0.327	0.221
	0.019	0.236	0.311	0.317
	5	4	4	4
	4.971	0.572	0.638	**0.538**
offshore	13.726	0.858	1.075	0.627
	0.047	0.255	0.284	0.269
	12	5	5	5
	13.773	1.113	1.359	**0.896**
ASIC_320ks	N/A	2.764	0.772	0.307
	N/A	0.253	0.316	0.286
	N/A	10	10	10
	N/A	3.017	1.088	**0.593**
apache2	4.471	0.122	0.201	0.088
	1.646	1.353	1.709	1.331
	2503	1256	1256	1256
	6.117	1.475	1.910	**1.419**
G3_circuit	/	0.069	0.106	0.061
	/	1.072	1.189	1.068
	>10 000	472	472	472
	/	1.141	1.295	**1.129**
power9	N/A	2.073	7.346	1.519
	N/A	0.336	0.359	0.345
	N/A	37	37	37
	N/A	2.409	7.705	**1.864**
msdoor	N/A	20.142	9.964	8.225
	N/A	1.635	2.033	1.790
	N/A	656	656	656
	N/A	21.777	11.997	**10.015**
thermal2	/	0.176	0.273	0.144
	/	3.757	3.806	3.699
	>10 000	2186	2186	2186
	/	3.933	4.079	**3.843**
Fault_639	N/A	65.339	20.396	15.558
	N/A	3.348	3.821	3.419
	N/A	1149	1149	1149
	N/A	68.687	24.217	**18.977**

From Tables 5 and 6, we can see that as compared to SSPAI-VCL on two GPUs, for some matrices such as thermal2 and G3_circuit, GPUPBICGSTAB with SSPAI-VCL cannot converge in 10,000 iterations while GSPAI-Opt can. Especially, for the matrices with large $n2max$ value, e.g., cbuckle, power9, ASIC_320ks, msdoor and Fault_639, GPUPBICGSTAB with SSPAI-VCL will encounter the out-of-memory error. Furthermore, for 2cubes_sphere, offshore, and apache2, the total time of SSPAI-VCL and GPUPBICGSTAB on two GPUs is much more than that of GSPAI-Opt and GPUPBICGSTAB. This verifies that GSPAI-Opt is much better than SSPAI-VCL for all test matrices. Compared with SPAI-Adaptive and GSPAI-Adaptive, GSPAI-Opt does not only have smaller execution time, but also the total time of GSPAI-Opt and GPUPBICGSTAB is much less than that of SPAI-Adaptive and GPUPBICGSTAB and that of GSPAI-Adaptive and GPUPBICGSTAB for all test cases. Fig. 11 shows the execution time ratios of SPAI-Adaptive to GSPAI-Opt and GSPAI-Adaptive to GSPAI-Opt on two GPUs. On the GTX1070 GPU, the minimum and maximum execution time ratios of SPAI-Adaptive to GSPAI-Opt are roughly 1.0 and 4.95, respectively, and the average ratio is roughly 2.62; the minimum and maximum execution time ratios of GSPAI-Adaptive to GSPAI-Opt are roughly 1.13 and 3.73, respectively, and the average ratio is roughly 1.57. On the A40 GPU, the minimum and maximum execution time ratios of SPAI-Adaptive to GSPAI-Opt

Table 7
Comparison of three algorithms with $(E+|A|)^2$.

Matrix	GTX1070			TITANXp		
	SPAI-A	GSPAI-A	GSPAI-Opt	SPAI-A	GSPAI-A	GSPAI-Opt
cbuckle	29.119	9.213	7.159	12.773	4.361	3.071
	0.492	0.470	0.477	0.272	0.319	0.270
	66	66	66	55	55	55
	29.479	9.564	**7.500**	13.045	4.680	**3.341**
2cubes_sphere	50.005	25.548	25.114	16.383	10.040	8.461
	0.171	0.171	0.161	0.132	0.141	0.151
	2	2	2	4	4	4
	50.176	25.719	**25.275**	16.515	10.181	**8.612**
offshore	133.587	73.907	69.016	44.813	30.238	22.370
	0.213	0.213	0.196	0.173	0.186	0.207
	3	3	3	3	3	3
	133.800	74.120	**69.212**	44.986	30.424	**22.577**
ASIC_320ks	10.223	2.460	1.699	5.667	1.185	1.040
	0.346	0.338	0.341	0.287	0.282	0.291
	6	6	6	6	6	6
	10.569	2.798	**2.040**	5.954	1.467	**1.331**
apache2	3.934	3.627	3.249	1.500	1.391	1.314
	2.913	2.907	2.883	1.026	0.989	0.984
	629	629	629	600	600	600
	6.847	6.534	**6.132**	2.526	2.380	**2.298**
G3_circuit	1.864	2.094	1.467	0.709	0.946	0.652
	2.291	2.283	2.292	0.985	0.946	0.982
	299	299	299	345	345	345
	4.155	4.377	**3.759**	1.714	1.892	**1.634**
power9	20.575	26.960	13.497	10.866	17.445	6.981
	0.411	0.409	0.393	0.329	0.302	0.309
	21	21	21	21	21	21
	20.986	27.369	**13.890**	11.195	17.747	**7.290**
msdoor	335.084	115.082	108.323	108.962	44.192	33.335
	2.471	2.470	2.471	1.072	0.959	0.955
	892	892	892	298	298	298
	337.555	117.552	**110.794**	110.034	45.151	**33.290**
thermal2	4.082	3.753	2.991	1.526	1.431	1.260
	11.462	11.466	11.469	3.469	3.477	3.529
	1502	1502	1502	1464	1464	1464
	15.544	15.219	**14.460**	4.995	4.908	**4.789**
Fault_639	627.18	201.616	180.840	235.209	83.214	63.198
	16.245	6.242	6.242	2.373	1.748	1.751
	588	588	588	473	473	473
	633.425	207.858	**187.082**	237.582	84.962	**64.949**

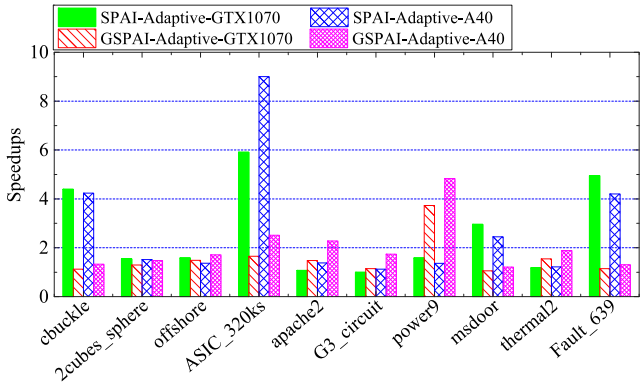

Fig. 11. Execution time ratios of SPAI-Adaptive vs GSPAI-Opt and GSPAI-Adaptive vs GSPAI-Opt for $E+|A|$ on two GPUs.

are roughly 1.12 and 9, respectively, and the average ratio is roughly 2.79; the minimum and maximum execution time ratios of GSPAI-Adaptive to GSPAI-Opt are roughly 1.21 and 4.83, respectively, and the average ratio is roughly 2.03. These observations verify that GSPAI-Opt outperforms SPAI-Adaptive and GSPAI-Adaptive.

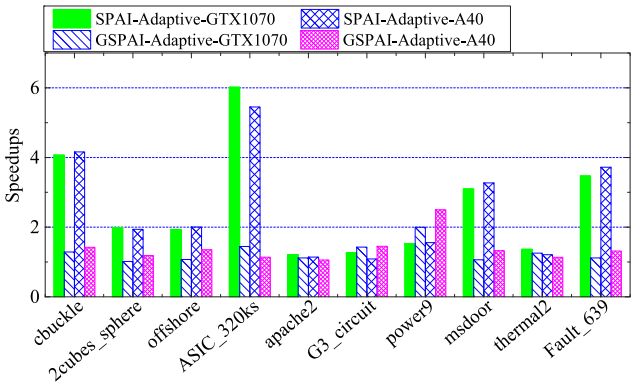

Fig. 12. Execution time ratios of SPAI-Adaptive vs GSPAI-Opt and GSPAI-Adaptive vs GSPAI-Opt for $(E+|A|)^2$ on two GPUs.

For the sparsity pattern of $(E+|A|)^2$, from Table 7, we can observe that comparing with SPAI-Adaptive and GSPAI-Adaptive, we can draw the same conclusion as the sparsity pattern of $(E+|A|)$ for GSPAI-Opt. GSPAI-Opt is much better than SPAI-Adaptive and GSPAI-Adaptive. This can also be confirmed from Fig. 12. On the GTX1070 GPU, the minimum and maximum execution time ratios of SPAI-Adaptive to GSPAI-Opt are roughly 1.99 and 6.02, respectively, and the average ratio is roughly 2.59; the minimum and maximum execution time ratios of GSPAI-Adaptive to GSPAI-Opt are roughly 1.01 and 2, respectively, and the average ratio is roughly 1.28. On the A40 GPU, the minimum and maximum execution time ratios of SPAI-Adaptive to GSPAI-Opt are roughly 1.14 and 5.45, respectively, and the average ratio is roughly 2.55; the minimum and maximum execution time ratios of GSPAI-Adaptive to GSPAI-Opt are roughly 1.05 and 2.5, respectively, and the average ratio is roughly 1.39.

4. Conclusion

In this study, we propose an optimized sparse approximate inverse preconditioners on GPU called GSPAI-Opt. In the proposed GSPAI-Opt, for any given sparsity pattern of the preconditioner, a selection strategy is presented to determine the size of the thread group for each column of the preconditioner. Furthermore, no matter which strategy we choose, each column of the preconditioner is performed in parallel within a thread group. The experimental results verify that GSPAI-Opt can well fuse the advantages of SPAI-Adaptive and GSPAI-Adaptive and is highly effective.

Next, we will further do research in this field, and apply the proposed GSPAI-Opt to more practical problems.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] CUDA C Programming guide 1.0, 2007, https://developer.nvidia.com/content/cuda-10.
[2] X. Chu, J. Gao, B. Sheng, Efficient concurrent L1-minimization solvers on GPUs, Comput. Syst. Sci. Eng. 38 (3) (2021) 305–320.
[3] J. Gao, Y. Xia, R. Yin, G. He, Adaptive diagonal sparse matrixvector multiplication on GPU, J. Parallel Distrib. Comput. 157 (2021) 287–302.
[4] K. Li, W. Yang, K. Li, A hybrid parallel solving algorithm on GPU for quasi-tridiagonal system of linear equations, IEEE Trans. Parallel Distrib. 27 (10) (2016) 2795–2808.
[5] S.C. Rennich, D. Stosic, T.A. Davis, Accelerating sparse cholesky factorization on GPUs, Parallel Comput. 59 (2016) 140–150.

[6] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, M. Kohler, Preconditioned Krylov solvers on GPUs, Parallel Comput. 68 (2017) 32–44.
[7] E. Chow, A. Patel, Fine-grained parallel incomplete LU factorization, SIAM J. Sci. Comput. 37 (2) (2015) C169–C193.
[8] J. Gao, X. Chu, X. Wu, J. Wang, G. He, Parallel dynamic sparse approximate inverse preconditioning algorithm on GPU, IEEE Trans. Parallel Distrib. 33 (12) (2022) 4723–4737.
[9] Y. Saad, M.H. Schultz, GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 7 (3) (1986) 856–869.
[10] H.A. van der Vorst, 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) (1992) 631–644.
[11] E. Chow, A priori sparsity patterns for parallel sparse approximate inverse preconditioners, SIAM J. Sci. Comput. 21 (5) (2000) 1804–1822.
[12] J. Gao, K. Wu, Y. Wang, P. Qi, G. He, GPU-accelerated preconditioned GMRES method for two-dimensional Maxwell's equations, Int. J. Comput. Math. 94 (10) (2017) 2122–2144.
[13] K. Rupp, R. Tillet, F. Rudolf, et al., ViennaCL-linear algebra library for multi- and many-core architectures, SIAM J. Sci. Comput. 38 (5) (2016) S412–S439.
[14] M.M. Dehnavi, D.M. Fernandez, J.L. Gaudiot, Parallel sparse approximate inverse preconditioning on graphic processing units, IEEE Trans. Parallel Distrib. 24 (9) (2013) 1852–1861.
[15] G. He, R. Yin, J. Gao, An efficient sparse approximate inverse preconditioning algorithm on GPU, Concurr. Comput.-Pract. Exp. 32 (7) (2020) e5598, http://dx.doi.org/10.1002/cpe.5598.
[16] J. Gao, Q. Chen, G. He, A thread-adaptive sparse approximate inverse preconditioning algorithm on multi-GPUs, Parallel Comput. 101 (2021) 102724, http://dx.doi.org/10.1016/j.parco.2020.102724.
[17] T.A. Davis, Y. Hu, The university of florida sparse matrix collection, ACM Trans. Math. Software 38 (1) (2011) 1–25.
[18] CUDA C Programming guide 11.1, 2021, http://docs.nvidia.com/cuda/cuda-c-programming-guide.