BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100007

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

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

A parallel sparse approximate inverse preconditioning algorithm based on MPI and CUDA☆

Yizhou Wang, Wenhao Li, 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
MPI

ABSTRACT

In this study, we present an efficient parallel sparse approximate inverse (SPAI) preconditioning algorithm based on MPI and CUDA, called HybridSPAI. For HybridSPAI, it optimizes a latest static SPAI preconditioning algorithm, and is extended from one GPU to multiple GPUs in order to process large-scale matrices. We make the following significant contributions: (1) a general parallel framework for optimizing the static SPAI preconditioner based on MPI and CUDA is presented, and (2) for each component of the preconditioner, a decision tree is established to choose the optimal kernel of computing it. Experimental results show that HybridSPAI is effective, and outperforms the popular preconditioning algorithms in two public libraries, and a latest parallel SPAI preconditioning algorithm.

1. Introduction

It has proved that sparse approximate inverse (SPAI) preconditioners can effectively accelerate the convergence rate of Krylov subspace methods, such as the generalized minimal residual method (GMRES) [1] and the biconjugate gradient stabilized method (BiCGSTAB) [2]. Moreover, compared with the incomplete factorization preconditioners [3–6] and the factorized sparse approximate inverse (FSAI) preconditioners [7–10], SPAI preconditioners neither require excessively sparse matrix–vector multiplication operations nor take care of the risk of breakdowns that can be encountered by FSAI preconditioners [11]. Consequently, SPAI preconditioners have attracted much attention [12–17].

In recent years, graphic processing units (GPUs) have become an important resource for scientific computing because of their many core structures and powerful computation efficiency, and have been used as tools for high-performance computation in a lot of fields [18–21]. As we know, the cost of constructing SPAI preconditioners is commonly very expensive for large-scale matrices, because the memory requirements to store them, and the computation requirements to calculate them are approximately the scale with the square to third power of the number of nonzeros in each row.

With the emerging of graphic processing units (GPUs), many studies have been conducted to accelerate the construction of SPAI preconditioners on the GPU architecture, and many parallel preconditioning algorithms [11,22–26] are proposed. Based on the degree of freedom used, SPAI preconditioner generation is classified as static (a priori) or adaptive. In this paper, we focus on optimizing a latest static SPAI preconditioning algorithm and extend it from one GPU to multiple GPUs. There has existed some work about static SPAI preconditioners on GPU [11,27], but the detailed implementations never be given and the source code is not public. Furthermore, He and Gao et al. propose two static SPAI preconditioning algorithms on GPU, called SPAI-Adaptive [28] and GSPAI-Adaptive [29], and give their implementation details. The two algorithms are verified to be effective for large-scale matrices. In this study, inspired by Gao's work, we further investigate how to highly optimize the static SPAI on multi-GPUs instead of only single GPU in this paper. We propose an optimized SPAI preconditioning algorithm based on MPI and CUDA, called HybridSPAI. Compared to a latest static SPAI preconditioning algorithm, the proposed algorithm has the following distinct characteristics. First, a general parallel framework based on MPI and CUDA is presented to optimize the static SPAI preconditioner, and is extended from one GPU to multiple GPUs. For each GPU, it operates same procedures as shown in Section 3.3, such as finding indices I and J, constructing the local submatrix, decomposing the local submatrix into QR, and solving the upper triangular linear systems. For MPI, it provides a simple and easy-to-use parallel controlling capability on multicore CPUs, which dedicates one thread for controlling one GPU. Second, when a sparsity pattern of the preconditioner is given, we use the thread-adaptive allocation strategy to choose the optimized number of threads for each column of the preconditioner, and construct the decision tree to choose the optimization kernel to calculate each one of components

☆ The research has been supported by the Natural Science Foundation of China under grant number 61872422, and the Natural Science Foundation of Jiangsu Province, China under grant number BK20171480.

* Corresponding author.
E-mail addresses: 1966224230@qq.com (Y. Wang), 917339495@qq.com (W. Li), springf12@163.com (J. Gao).

https://doi.org/10.1016/j.tbench.2021.100007
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 6 November 2021

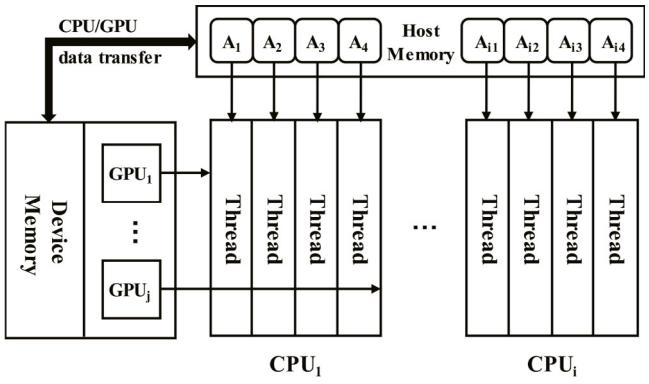

Fig. 1. A CPU–GPU hybrid parallel computing model based on MPI.

of the preconditioner. Experimental results show that HybridSPAI is effective, and is advantageous over the popular incomplete LU factorization algorithm in the CUSPARSE library [30], the static SPAI preconditioning algorithm in the ViennaCL library [24], and the latest GSPAI-Adaptive [29].

The main contributions in this paper are summarized as follows.

- A general parallel framework based on MPI and CUDA is presented for optimizing the static SPAI preconditioner, and is extended from one GPU to multiple GPUs, also the CPU and GPU tasks are designated.
- A strategy is presented to choose the optimal number of threads for each column of the preconditioner.
- On the basis of the parallel framework and proposed strategy, an optimization SPAI preconditioning algorithm based on MPI and CUDA, called HybridSPAI, is presented. In HybridSPAI, finding indices, constructing local submatrix, decomposing the local submatrix into *QR*, and solving the upper triangular linear system are computed in parallel, and the kernels of calculating them are selected by the decision tree optimization.

The rest of this paper is organized as follows. Section 2 describes the SPAI preconditioning algorithm, Section 3 gives the detailed implementation of HybridSPAI, Section 4 presents the experimental analysis and evaluation, and Section 5 contains our conclusions and points to our future research directions.

2. SPAI algorithm

The basic idea of the SPAI procedure [22] is described as follows: Use a sparse matrix M, known as the preconditioner, to approximate the inverse of A, and M is computed by the following formula:

$$\min \|AM - E\|_F^2. \tag{1}$$

Owing to the independence of the columns of M, the equation mentioned above can be separated into the following n independent least squares problems

$$\min_{m_k} \left\|Am_k - e_k\right\|_2^2, \quad k = 1, 2, \ldots, n \tag{2}$$

where e_k is the kth column of the identity matrix and m_k represents column k in matrix M. For a description of the implementation details of SPAI, we refer to the literature [27].

3. Optimizing SPAI on GPUs

We present an optimization SPAI preconditioning algorithm based on CPU–GPU platforms, called HybridSPAI. The hybrid parallel computing model is illustrated in Fig. 1. and the parallel framework of HybridSPAI is shown in Fig. 2, which includes the following three stages: *Pre-HybridSPAI* stage, *Compute-HybridSPAI* stage, and *Post-HybridSPAI* stage.

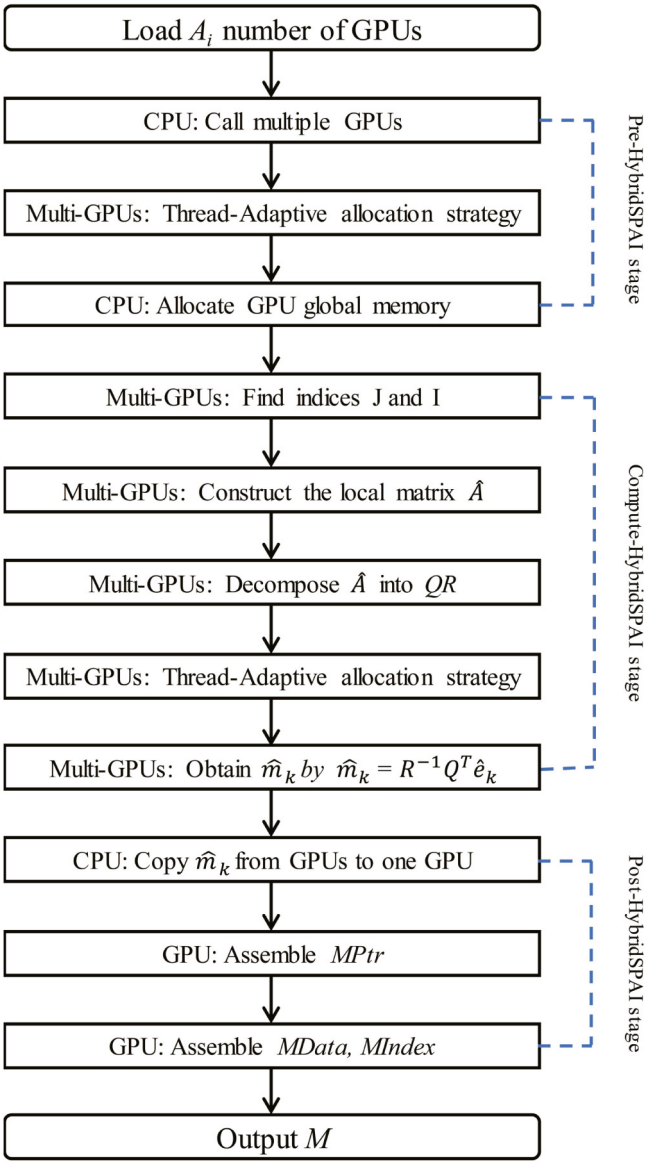

Fig. 2. Parallel framework of HybridSPAI.

3.1. Hybrid parallel programming based on MPI and CUDA

A hybrid parallel programming model must be designed for the architectures of GPU and CPU to improve the computing performance, and has the characteristics of extending to more devices. In our proposed model, as a device in CUDA, GPU can be controlled by each thread of multicore CPU, also can be controlled by each individual CPU. In addition, the data is transferred from the host memory to the GPU device memory, then the CPU launches the calculation process on the GPU by calling the kernel function.

MPI provides a simple and convenient parallel computing capability of multi-threads on multicore CPUs [31]. The hybrid parallel computing model is illustrated in Fig. 1, where A_1, A_2, …, A_{i3}, A_{i4},are submatrices which are stored in the host memory, and *Thread* are multi-threads which are assigned to cores of CPUs.

Note that when using this model, a computing matrix will be divided into multiple submatrices which corresponding with the number of calling threads of CPUs, so that these submatrices are assigned to each GPU to perform respectively.

Table 1
Arrays in HybridSPAI.

Array	Size	Type
AData	nonzeros	Double
AIndex	nonzeros	Integer
APtr	n	Integer
I	$ns \times n1max$	Integer
atmoic	n	Integer
J	$ns \times n2max$	Integer
jPTR	ns	Integer
$\hat{m}$	$ns \times n2max$	Double
$\hat{A}$	$ns \times n1max \times n2max$	Double
R	$ns \times n1max \times n2max$	Double
iPTR	ns	Integer

3.2. Pre-HybridSPAI Stage

In this paper, we summarize the sparsity of M in advance with the main method in [25]. *M(i,j)* is considered a nonzero if

$$|A(i,j)| > (1-\tau)\max_j |A(i,j)|, 0 \leqslant \tau \leqslant 1 \tag{3}$$

is satisfied, where τ is a user defined tolerance parameter (the main diagonal is always included).

Next, A is stored in host memory using the compressed sparse column(CSC) storage format, and M is also stored in columns. The dimensions of local submatrices ($n1_k$, $n2_k$) are usually distinct for different k, ($k = 1, 2, \ldots$,n). To simplify the accesses of data in memory and increasing the coalescence, the dimensions of all local submatrices are uniformly defined as ($n1max$, $n2max$), where $n1max = \max_k\{n1_k\}$ and $n2max = \max_k\{n2_k\}$.

Finally, the thread-adaptive allocation strategy is proposed. For any matrix, the number of threads z for each column of the preconditioner is calculated by the following formulas:

$$z = \min\left(2^l, nt\right) \tag{4}$$

$$\text{s.t. } 2^{l-1} < n2\ max \leqslant 2^l. \tag{5}$$

In Eqs. (4), nt is a fixed thread block size. z threads are grouped into a thread group, which is assigned to compute the kth column of M. The lowercase "L" in the Eqs. (4) was required to compute the suitable z threads. Note that we used a 1D array of the thread blocks to organize the compute grid in this paper, and used a 1D array of threads to organize the thread block as well.

3.3. Compute-HybridSPAI Stage

In the Compute-HybridSPAI stage, the allocations of every GPU global memory are shown in Table 1. Based on the characteristics of message interface, MPI is very convenient to scatter and gather data between the multiple threads of CPU. The following steps are implemented to compute M.

Finding J and I: In all blocks, each thread-group block size that is used to find J and I is same, and each thread group (warpSize threads) is assigned to find one subset of J and I, which making many subsets of J and I can be simultaneously obtained. Furthermore, parallelism is also exploited inside each thread group. For the kernel that finds J, the threads inside each warp (thread group) read one column of M in parallel, and store them to shared memory using atomic operation. For the kernel that finds I, a decision tree is established and for any given $n2max$ and $n1max$, this optimized kernel can be effective. Fig. 3 shows a segment of the decision tree for finding I. When 4 < n2max ≤ 8, cuFindIBySharedMemory kernel with shared memory of *sharedSize* size or cuFindI kernel with global memory will be selected according to different the $n1max$. Here *sharedSize* = number of computing columns of the preconditioner ×upper boundary closest to $n1max$. Fig. 4 shows the main procedure of cuFindIBySharedMemory kernel. Each thread group finds one subset of I, e.g., I_k. First, the row indices of the first column referenced in one subset of J, e.g., J_k are loaded to shared memory sI with the threads in the thread group. Then 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. Second, 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. When $n1max > 256$, cuFindI kernel is executed on global memory instead of shared memory, which is similar to cuFindIBySharedMemory kernel.

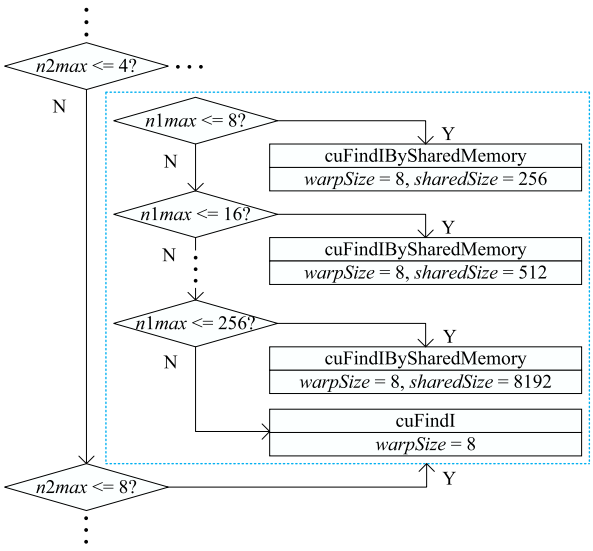

Fig. 3. A segment of the decision tree of find I.

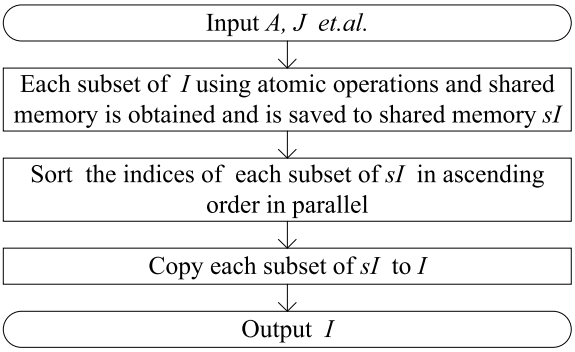

Fig. 4. Main procedure of cuFindIBySharedMemory kernel.

Constructing the local submatrix: Using J and I obtained above, the local matrix set $\hat{A}$, is computed by kernel with shared memory or kernel with global memory according to the established decision tree. Fig. 5 shows a segment of the decision tree for constructing $\hat{A}$. When 4 < n2max ≤ 8, cuComputeTildeABySharedMemory kernel with shared memory of *sharedSize* size or cuComputeTildeA kernel with global memory will be selected according to different $n1max$. Fig. 6 shows the main procedure of cuComputeTildeABySharedMemory kernel. For the thread group on each GPU 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 constructed on global memory by loading columns indexed in J_k and matching them to I_k in parallel. When $n1max > 256$, cuComputeTildeA kernel is executed on global memory instead of shared memory, which is similar to cuComputeTildeABySharedMemory kernel.

Decomposing the Local Submatrix into *QR*:The thread-group size of decomposing the local submatrix into *QR* is same in all blocks. Being similar with above two steps, the constructed decision tree is used again

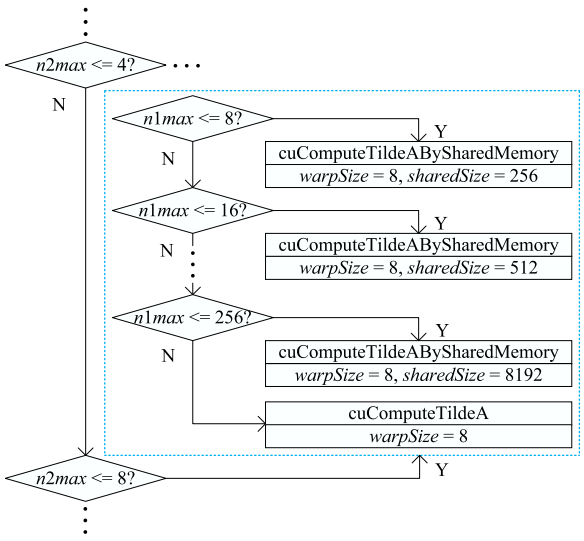

Fig. 5. A segment of the decision tree to construct $\hat{A}$.

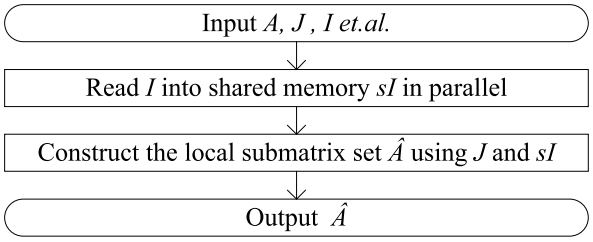

Fig. 6. Main procedure of cuComputeTildeABySharedMemory kernel.

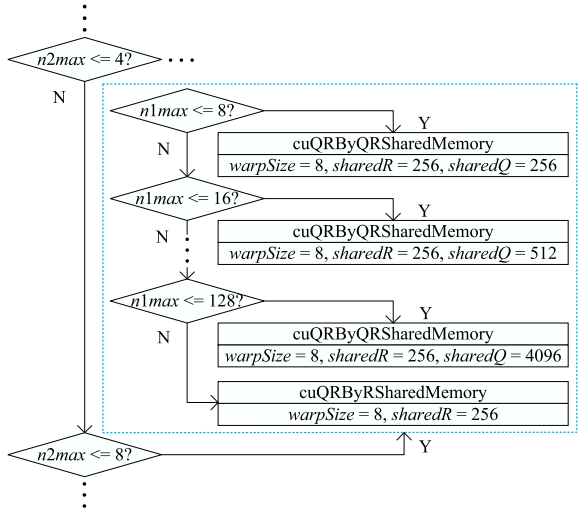

Fig. 7. A segment of the decision tree to decompose the local submatrix into *QR*.

to decompose local submatrix. Fig. 7 shows a segment of the decision tree for decomposing the local submatrix into *QR*. When 4 < n2max ≤ 8, cuQRByQRSharedMemory kernel with shared memory of *sharedSize* size and *sharedQ* size or cuQRByRSharedMemory kernel with shared memory of *sharedR* size will be selected according to different *n1max*. Fig. 8 shows the main procedure of cuQRByQRSharedMemory kernel. In addition, Eeach thread group is responsible for one *QR* decomposition. For a description of its detailed implementation, please refer to the literature [25]. In a thread group, the local submatrix, e.g., $\widehat{A_k}$, is decomposed into *QR* by the following four steps at each iteration *i*. In

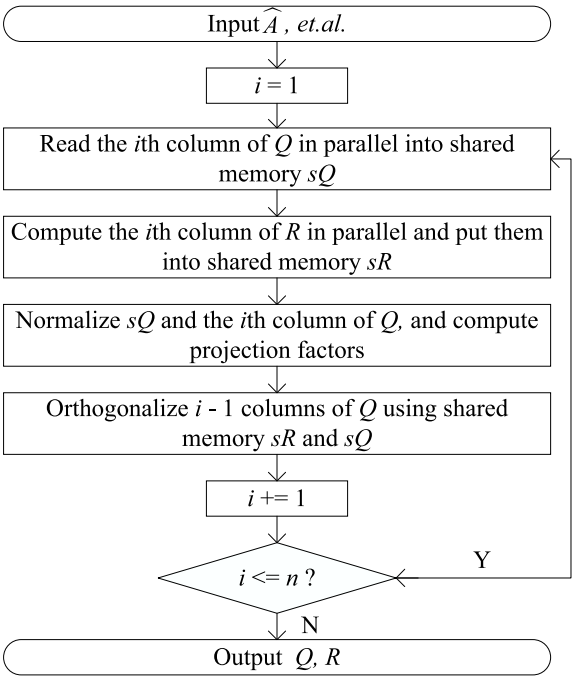

Fig. 8. Main procedure of cuQRByQRSharedMemory kernel.

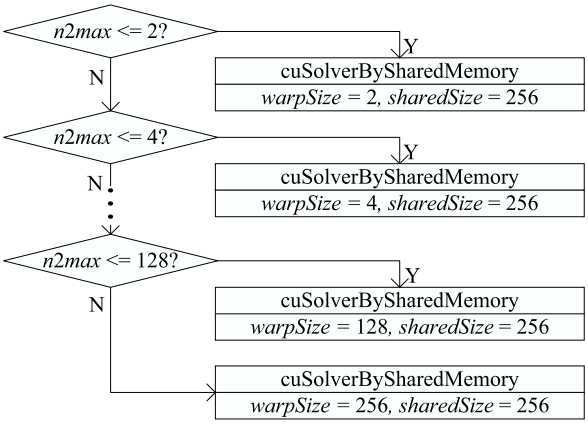

Fig. 9. A segment of the decision tree to solve the upper triangular linear system.

the first step, the *i*th column of Q_k are read into shared memory *sQ* in parallel. In the second step, the threads computed the *i*th row of the upper triangle matrix R_k in parallel and 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. When *n1max* > 128. cuQRByRSharedMemory kernel is executed by utilizing shared memory *sR* instead of shared memory *sQ*, which is similar to cuQRByQRSharedMemory kernel.

Solving the Upper Triangular Linear System: In this section, one subset of $\hat{m_k} = R_k^{-1} Q_k^T \hat{e_k}$ are computed by solving an upper triangular linear system. Fig. 9 shows a segment of the decision tree for solving an upper triangular linear system. For any given *n2max* value, cuSolverBySharedMemory with shared memory of 256 size and thread-group size of *warpSize*, is chosen. For example, when 4 < n2max ≤ 8, cuSolverBySharedMemory kernel with shared memory of 256 size and thread-group size of 8 is selected. Fig. 10 shows the main procedure of cuSolverBySharedMemory kernel. For each thread group,T the steps to compute $\hat{m}$, e.g., $\widehat{m_k}$, include: (1) $Q_k^T \hat{e_k}$ is calculated in parallel and saved to the shared memory xE, and (2) the values of $\hat{m_k}$ are obtained

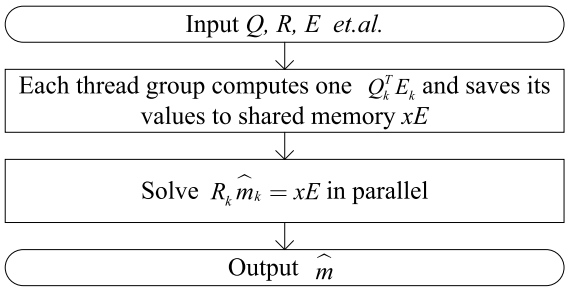

Fig. 10. Main procedure of cuSolverBySharedMemory kernel.

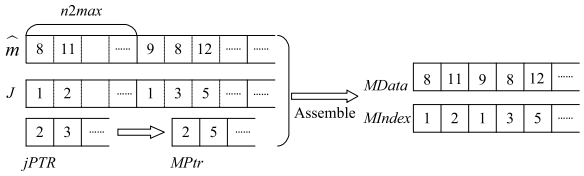

Fig. 11. Assemble *M*.

Table 2
Descriptions of test matrices.

Name	Kind	Rows	Nonzeros	*avg*	*max*	*min*
venkat01	CFD sequence	62,424	1,717,792	27.52	44	16
imagesensor	Semiconductor device	118,758	1,446,396	12.18	21	2
cfd2	CFDproblem	123,440	3,085,406	25.00	30	8
apache2	Structural	715,176	4,817,870	6.74	8	4
t2em	Electronmagnetics	921,632	4,590,832	4.98	5	1
thermal2	Thermal	1,228,045	8,580,313	6.99	11	1
G3_circuit	Circuitsimulation	1,585,478	7,660,826	4.83	6	2

by solving the upper triangular linear system, $R_k \hat{m}_k = xE$, in parallel using shared memory.

3.4. Post-HybridSPAI Stage

The Post-HybridSPAI Stage is to assemble *M* in the CSC storage format from multiple GPUs. Fig. 11 illustrates the procedure of assembling *MPtr*, *MIndex* and *MData* arrays on each GPU. First, *MPtr* is assembled utilizing *jPTR*. Second, $\hat{m}$ and *J* are utilized to assemble *MIndex* and *MData*. Finally, *MData* arrays on each GPU are transferred to the respective threads of CPU according to the device ID of GPUs. On the CPU, each thread utilize the function MPI_Gatherv() of MPI to gather the *MData* into a complete array in parallel.

4. Evaluation and analysis

We evaluate the performance of HybridSPAI in this section. The test matrices in Table 2 are used to evaluate the performance of NVIDIA GTX 1080 Ti GPUs, which are selected from University of Florida Sparse Matrix Collection. The source codes are compiled and executed using the CUDA toolkit 10.1.

4.1. Effectiveness analysis

For each test matrix, GPUPBICGSTAB are called to solve *Ax=b* on GTX 1080 Ti, where all values of *b* are 1 and the produced *M* is used as the preconditioner. They stop when the residual error is less than $1e^{-7}$, or the number of iterations exceeds 10,000. Table 3 shows the results, and the time unit is second (*s*).

In addition, we take GTX 1080 Ti to investigate the effort of single GPU and increasing the number of *threads* on the execution time of HybridSPAI and GPUPBICGSTAB with HybridSPAI. Table 4 demonstrates

Table 3
Iterations and execution time of GPUBICGSTAB on GTX 1080 Ti.

Matrix	GPUBICGSTAB		GPUPBICGSTAB	
	Iterations	Execution time	Iterations	Execution time
venkat01	10000	/	35	1.312
imagesensor	10000	/	52	1.036
cfd2	7768	5.167	1613	3.518
apache2	5813	8.061	1106	3.032
t2em	1661	3.122	768	2.338
thermal2	4095	9.771	2584	9.748
G3_circuit	10000	/	475	2.53

Table 4
Execution time of HybridSPAI and GPUPBICGSTAB.

Matrix	GPU	1 thread	2 thread	4 thread	8 thread
venkat01	0.506	0.501	0.262	0.151	0.102
	0.806	0.771	0.761	0.736	0.715
	1.312	1.272	1.023	0.887	0.817
imagesensor	0.228	0.227	0.179	0.103	0.104
	0.808	0.785	0.767	0.745	0.713
	1.036	1.012	0.946	0.848	0.817
cfd2	1.187	1.191	0.631	0.356	0.224
	2.331	2.231	2.294	2.178	2.101
	3.518	3.422	2.925	2.534	2.325
apache2	0.226	0.219	0.126	0.101	0.133
	2.806	2.761	2.746	2.734	2.838
	3.032	2.980	2.872	2.835	2.971
t2em	0.075	0.070	0.060	0.064	0.103
	2.263	2.253	2.241	2.231	2.268
	2.338	2.323	2.301	2.295	2.371
thermal2	0.332	0.329	0.201	0.165	0.164
	9.416	9.443	9.367	9.369	9.187
	9.748	9.772	9.568	9.534	9.351
G3_circuit	0.167	0.156	0.113	0.094	0.115
	2.363	2.302	2.321	2.329	2.290
	2.530	2.458	2.434	2.423	2.405

the execution time of this. For each matrix and given number of *threads*, the first row and second row are respectively the computing time of HybridSPAI and GPUPBICGSTAB, and the third row is the sum of time of the first two row. GPUPBICGSTAB stops while the residual error is less than $1e^{-7}$. The minimum values of the second and third rows for each matrix both are marked in the red font. In addition, we observe that when the time of computing the preconditioner keeps less than 228 ms on single GPU, increasing the number of GPU cannot provide significant acceleration.

4.2. Performance comparison

We test the HybridSPAI performance by comparing it with a popular preconditioning algorithms: CSRILU0 in CUSPARSE (denoted by CSRILU) [32], a static sparse approximate inverse preconditioning algorithm in ViennaCL (denoted by SSPAI-VCL) [33], and a latest parallel SPAI preconditioning algorithm(denoted by GSPAI-Adaptive) [29]. Table 5 demonstrate the comparison results on GTX 1080 Ti GPUs. For each matrix and the preconditioner, the first row is the computing time of these four preconditioning algorithms, and the second row and the third row are respectively the execution time and the number of iterations of GPUPBICGSTAB while the residual error is less than $1e^{-7}$. Note that "/" represents the number of iterations for HybridSPAI exceeds 10,000, and all the other rows for each matrix will be denoted except that the third row is denoted by "> 10000". The minimum value of the fourth row for each matrix is marked in the red font.

From Table 5, we observe that on GTX 1080 Ti, the total time of HybridSPAI and GPUPBICGSTAB with HybridSPAI is the smallest among all algorithms for any matrices. This displays that HybridSPAI outperforms CSRILU and SSPAI-VCL, and is advantageous over GSPAI-Adaptive.

Table 5
Execution time of all preconditioning algorithms and GPUPBICGSTAB on GTX 1080 Ti.

Matrix	CSRILU	SSPAI-VCL	GSPAI-Adaptive	HybridSPAI
venkat01	1.835	38.856	0.506	0.102
	1.574	0.036	0.806	0.715
	11	48	35	35
	3.427	38.892	1.312	0.817
imagesensor	/	/	0.228	0.104
	/	/	0.808	0.713
	10000	10000	52	52
	/	/	1.036	0.817
cfd2	/	/	1.187	0.224
	/	/	2.331	2.101
	10000	10000	1613	1613
	/	/	3.518	2.325
apache2	3.386	43.532	0.226	0.101
	6.776	2.995	2.806	2.734
	475	2503	1106	1106
	10.162	46.527	3.032	2.835
t2em	19.884	/	0.075	0.064
	2998.63	/	2.263	2.231
	427	10000	768	768
	3018.514	/	2.338	2.295
thermal2	5.502	/	0.332	0.164
	45.008	/	9.416	9.187
	1619	10000	2584	2584
	50.510	/	9.748	9.351
G3_circuit	5.245	/	0.167	0.115
	12.475	/	2.363	2.290
	257	10000	475	475
	17.720	/	2.530	2.405

5. Conclusion

We present an efficient parallel sparse approximate inverse preconditioning algorithm on multi-GPUs in this paper, which is based MPI and CUDA, called HybridSPAI. In our proposed HybridSPAI, a general parallel framework is embraced for optimizing the static SPAI on multi-GPUs, and a decision tree is established to choose the optimal kernel for computing it. The experimental results demonstrate a noticeable performance and high effectiveness of our proposed HybridSPAI.

References

[1] Y. Saad, M.H. Schultz, GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 7 (1986) 856–869.
[2] H.A: Bi-CGSTAB: a fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems, SIAM J. Stat. Comput. 13 (2) (1992) 631.
[3] Y. Saad, Iterative Methods for Sparse Linear Systems, second version, SIAM, Philadelphia, PA, 2003.
[4] J. Gao, R. Liang, J. Wang, Research on the conjugate gradient algorithm with a modified incomplete cholesky preconditioner on GPU, J. Parallel Distr. Com. 74 (2) (2014) 2088–2098.
[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] L.Y. Kolotilina, A.Y. Yeremin, Factorized sparse approximate inverse preconditioning I. theory, SIAM J. Matrix Anal. Appl. 14 (1) (1993) 45–58.
[8] M. Benzi, C.D. Meyer, M. Tuma, A sparse approximate inverse preconditioner for the conjugate gradient method, SIAM J. Sci. Comput. 17 (5) (1996) 1135–1149.
[9] M. Ferronato, C. Janna, G. Pini, A generalized block FSAI preconditioner for nonsymmetric linear systems, J. Comput. Appl. Math. 256 (2014) 230–241.
[10] V.A.P. Magri, A. Franceschini, M. Ferronato, C. Janna, Multilevel approaches for FSAI preconditioning, Numer. Linear Algebr. (2018) e2183, http://dx.doi.org/10.1002/nla.2183.
[11] M.M. Dehnavi, D.M. Fern´andez, J.L. Gaudiot, D.D. Giannacopoulos, Parallel sparse approximate inverse preconditioning on graphic processing units, IEEE T. Parall. Distr. 24 (9) (2013) 1852–1861.
[12] Z. Jia, B. Zhu, A power sparse approximate inverse preconditioning procedure for large sparse linear systems, Numer. Linear Algebr. 16 (4) (2009) 259–299.
[13] J.D.F. Cosgrove, J.C. Diaz, A. Griewank, Approximate inverse preconditioning for sparse linear systems, Int. J. Comput. Math. 44 (1–2) (1992) 91–110.
[14] M. Grote, T. Huckle, Parallel preconditioning with sparse approximate inverses, SIAM J. Sci. Comput. 18 (3) (1997) 838–853.
[15] E. Chow, Y. Saad, Approximate inverse preconditioners via sparse-sparse iterations, SIAM J. Sci. Comput. 19 (3) (1998) 995–1023.
[16] E. Chow, A priori sparsity patterns for parallel sparse approximate inverse preconditioners, SIAM J. Sci. Comput. 21 (5) (2000) 1804–1822.
[17] E. Chow, A. Patel, Fine-grained parallel incomplete LU factorization, SIAM J. Sci. Comput. 37 (2) (2015) C169–C193.
[18] J. Gao, Z. Li, R. Liang, G. He, Adaptive optimization l1-minimization solvers on GPU, Int. J. Parallel Program. 45 (3) (2017) 508–529.
[19] K. Li, W. Yang, K. Li, A hybrid parallel solving algorithm on GPU for quasitridiagonal system of linear equations, IEEE Trans. Parallel Distrib. Syst. 27 (10) (2016) 2795–2808.
[20] J. Gao, Y. Zhou, G. He, Y. Xia, A multi-GPU parallel optimization model for the preconditioned conjugate gradient algorithm, Parallel Comput. 63 (2017) 1–16.
[21] G. He, J. Gao, J. Wang, Efficient dense matrix–vector multiplication on GPU, Concurr. Comput. Pract. Exp. 30 (19) (2018) e4705, http://dx.doi.org/10.1002/cpe.4705.
[22] 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.
[23] M. Lukash, K. Rupp, S. Selberherr, Sparse approximate inverse preconditioners for iterative solvers on GPUs, in: Proceedings of the 2012 Symposium on High Performance Computing, Society for Computer Simulation, San Diego, CA, USA, 2012, pp. 1–8.
[24] K. Rupp, R. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jungel, S. Selberher, ViennaCL-linear algebra library for multi-and many-core architectures, SIAM J. Sci. Comput. 38 (5) (2016) S412–S439.
[25] G. He, R. Yin, J. Gao, An efficient sparse approximate inverse preconditioning algorithm on GPU, Concurr. Comput.-Pract. Exp. 32 (7) (2020) e5598.
[26] 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.
[27] 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.
[28] 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.
[29] 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.
[30] NVIDIA, Cusparse library, 2019, v10.1, https://docs.nvidia.com/cuda/cusparse/index.html.
[31] Yang. C.T., Huang. C.L., Lin. C.F., Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters, Comp. Phys. Commun. 182 (1) (2011) 266–269, http://dx.doi.org/10.1016/j.cpc.2010.06.035.
[32] Cusparse library, v10.1. https://docs.nvidia.com/cuda/cusparse/index.html.
[33] K. Rupp, et al., ViennaCl–linear algebra library for multi-and many-core architectures, SIAM J. Sci. Comput. 38 (5) (2016) S412–S439.