[BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100007](https://doi.org/10.1016/j.tbench.2021.100007)



Contents lists available at [ScienceDirect](http://https//www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/)

BenchCouncil Transactions on Benchmarks,

Standards and Evaluations

journal homepage: [https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-](http://https//www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/) [benchmarks-standards-and-evaluations/](http://https//www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/)

[](http://crossmark.crossref.org/dialog/?doi=10.1016/j.tbench.2021.100007&domain=pdf)A parallel sparse approximate inverse preconditioning algorithm based on MPI and CUDA[✩](#_bookmark0)

Yizhou Wang, Wenhao Li, Jiaquan Gao [∗](#_bookmark1)

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

A R T I C L E I N F O A B S T R A C T

*Keywords:*

Sparse approximate inverse Preconditioning

CUDA GPU MPI

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.

# Introduction

It has proved that sparse approximate inverse (SPAI) precondi- tioners can effectively accelerate the convergence rate of Krylov sub- space methods, such as the generalized minimal residual method (GM- RES) [[1](#_bookmark25)] and the biconjugate gradient stabilized method (BiCGSTAB) [[2](#_bookmark26)]. Moreover, compared with the incomplete factorization precon- ditioners [[3](#_bookmark27)–[6](#_bookmark28)] and the factorized sparse approximate inverse (FSAI) preconditioners [[7](#_bookmark29)–[10](#_bookmark30)], SPAI preconditioners neither require exces- sively sparse matrix–vector multiplication operations nor take care of the risk of breakdowns that can be encountered by FSAI precondi- tioners [[11](#_bookmark31)]. Consequently, SPAI preconditioners have attracted much attention [[12](#_bookmark32)–[17](#_bookmark33)].

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](#_bookmark34)–[21](#_bookmark35)]. 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 precon- ditioners on the GPU architecture, and many parallel preconditioning algorithms [[11](#_bookmark31),[22](#_bookmark36)–[26](#_bookmark39)] 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](#_bookmark31),[27](#_bookmark40)], 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](#_bookmark41)] and GSPAI-Adaptive [[29](#_bookmark42)], and give their imple- mentation 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 Hy- bridSPAI. 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](#_bookmark9), 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.

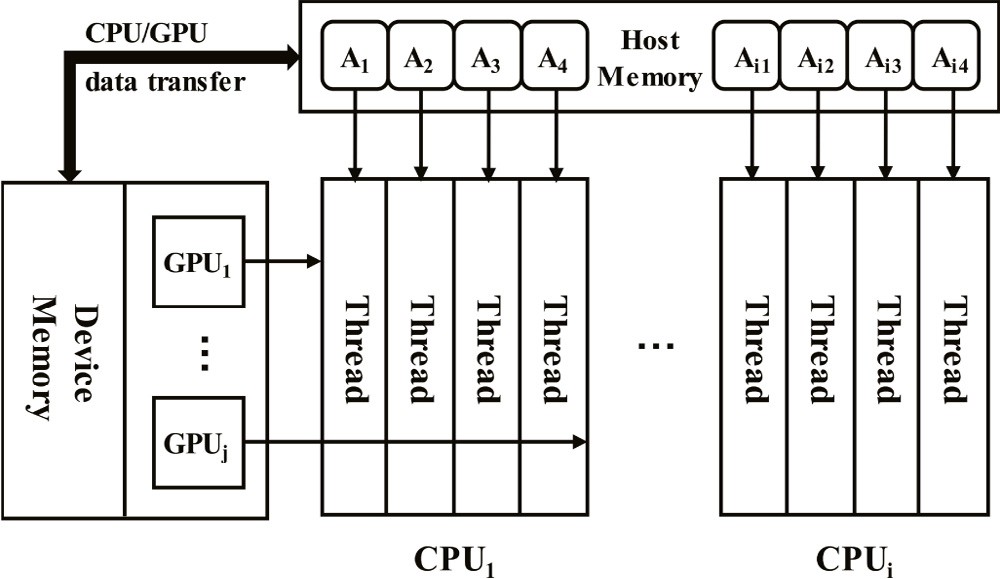
*E-mail addresses:* [1966224230@qq.com](mailto:1966224230@qq.com) (Y. Wang), [917339495@qq.com](mailto:917339495@qq.com) (W. Li), [springf12@163.com](mailto:springf12@163.com) (J. Gao).

<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

2772-4859/© 2021 The Authors. Publishing services by Elsevier B.V. on behalf of KeAi Communications Co. Ltd. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).



**/ig. 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 fac- torization algorithm in the CUSPARSE library [[30](#_bookmark43)], the static SPAI preconditioning algorithm in the ViennaCL library [[24](#_bookmark37)], and the latest GSPAI-Adaptive [[29](#_bookmark42)].

The main contributions in this paper are summarized as follows.

* A general parallel framework based on MPI and CUDA is pre- sented 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](#_bookmark3) describes the SPAI preconditioning algorithm, Section [3](#_bookmark6) gives the detailed implemen- tation of HybridSPAI, Section [4](#_bookmark22) presents the experimental analysis and evaluation, and Section [5](#_bookmark24) contains our conclusions and points to our future research directions.

# SPAI algorithm

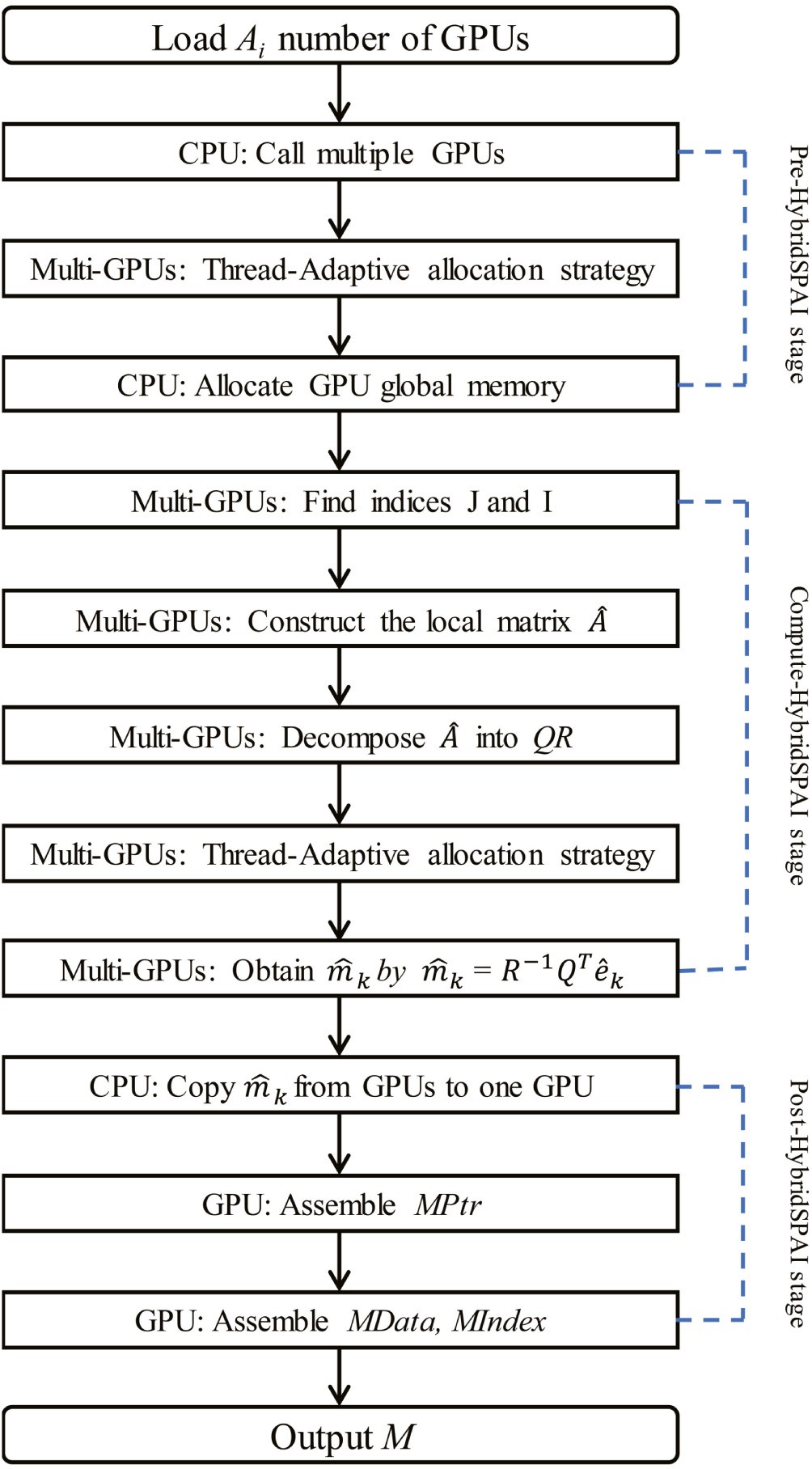
The basic idea of the SPAI procedure [[22](#_bookmark36)] is described as follows: Use a sparse matrix *M*, known as the preconditioner, to approximate the inverse of *A*, and *M* is computed by the following formula:

min ‖*𝐴𝑀* − *𝐸*‖2 *.* (1)

*𝐹*

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

**/ig. 2.** Parallel framework of HybridSPAI.

* 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 pro- posed model, as a device in CUDA, GPU can be controlled by each

min *𝐴𝑚𝑘*

‖

*𝑚𝑘*

– *𝑒𝑘*‖2 *, 𝑘* = 1*,* 2*,* … *, 𝑛* (2)

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

where *𝑒𝑘* is the *k*th column of the identity matrix and *𝑚𝑘* represents

2

column *k* in matrix *M*. For a description of the implementation details

of SPAI, we refer to the literature [[27](#_bookmark40)].

# Optimizing SPAI on GPUs

We present an optimization SPAI preconditioning algorithm based on CPU–GPU platforms, called HybridSPAI. The hybrid parallel com- puting model is illustrated in [Fig.](#_bookmark2) [1](#_bookmark2). and the parallel framework of Hy- bridSPAI is shown in [Fig.](#_bookmark5) [2](#_bookmark5), which includes the following three stages: *Pre-HybridSPAI* stage, *Compute-HybridSPAI* stage, and *Post-HybridSPAI* stage.

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](#_bookmark44)]. The hybrid parallel computing

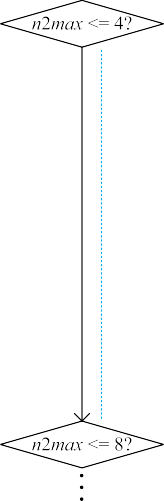
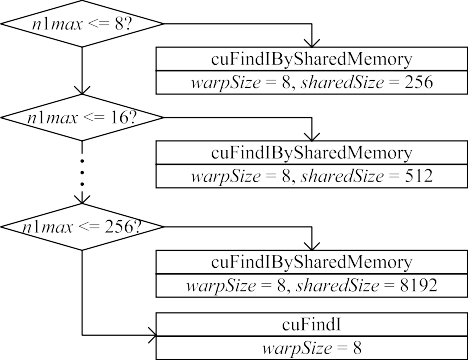
model is illustrated in [Fig.](#_bookmark2) [1](#_bookmark2), where *𝐴*1, *𝐴*2, . . . , *𝐴𝑖*3, *𝐴𝑖*4,are submatrices

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 di- vided 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* × *n1max* Integer

*atmoic n* Integer

*J ns* × *n2max* Integer

*jPTR ns* Integer

*𝑚̂*

*𝐴̂*

*ns* × *n2max* Double

*ns* × *n1max* × *n2max* Double

*R ns* × *n1max* × *n2max* Double

*iPTR ns* Integer

* 1. *Pre-HybridSPAI Stage*

In this paper, we summarize the sparsity of *M* in advance with the main method in [[25](#_bookmark38)]. *M(i,j)* is considered a nonzero if

|*𝐴*(*𝑖, 𝑗*)| *>* (1 − *𝜏*) max |*𝐴*(*𝑖, 𝑗*)|*,* 0 ⩽ *𝜏* ⩽ 1 (3)

*𝑗*

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 (*𝑛*1*𝑘*, *𝑛*2*𝑘*) are usually distinct for

different *k*, (*k* = 1, 2, . . . ,n). To simplify the accesses of data in memory

are uniformly defined as (*𝑛*1*𝑚𝑎𝑥*, *𝑛*2*𝑚𝑎𝑥*), where *𝑛*1*𝑚𝑎𝑥* = max*𝑘*{*𝑛*1*𝑘*} and increasing the coalescence, the dimensions of all local submatrices and *𝑛*2*𝑚𝑎𝑥* = max*𝑘*{*𝑛*2*𝑘*}.

matrix, the number of threads *𝑧* for each column of the preconditioner Finally, the thread-adaptive allocation strategy is proposed. For any

is calculated by the following formulas:

*𝑧* = min (2*𝑙, 𝑛𝑡*) (4)

s.t. 2*𝑙*−1 *< 𝑛*2 *𝑚𝑎𝑥* ⩽ 2*𝑙 .* (5)

In Eqs. ([4](#_bookmark8)), *𝑛𝑡* is a fixed thread block size. *𝑧* threads are grouped into a

lowercase ‘‘L’’ in the Eqs. ([4](#_bookmark8)) was required to compute the suitable *𝑧* thread group, which is assigned to compute the *k*th column of *M*. The

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.

* 1. *Compute-HybridSPAI Stage*

In the Compute-HybridSPAI stage, the allocations of every GPU global memory are shown in [Table](#_bookmark7) [1](#_bookmark7). 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*.

**/inding** *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 *𝑛*2*𝑚𝑎𝑥* and *𝑛*1*𝑚𝑎𝑥*, this optimized kernel can be effective. [Fig.](#_bookmark10) [3](#_bookmark10)

shows a segment of the decision tree for finding *I*. When 4 *<* n2max ≤

8, cuFindIBySharedMemory kernel with shared memory of *sharedSize*

to different the *𝑛*1*𝑚𝑎𝑥*. Here *sharedSize* = number of computing columns size or cuFindI kernel with global memory will be selected according of the preconditioner ×upper boundary closest to *𝑛*1*𝑚𝑎𝑥*. [Fig.](#_bookmark11) [4](#_bookmark11) shows

the main procedure of cuFindIBySharedMemory kernel. Each thread

**/ig. 3.** A segment of the decision tree of find *𝐼* .



**/ig. 4.** Main procedure of cuFindIBySharedMemory kernel.

group finds one subset of *I*, e.g., *𝐼𝑘*. First, the row indices of the first column referenced in one subset of *J*, e.g., *𝐽𝑘* are loaded to shared memory *𝑠𝐼* with the threads in the thread group. Then the index vectors of successive columns referenced by *𝐽𝑘* 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

*𝐼𝑘*. When *n1max >* 256, cuFindI kernel is executed on global memory

instead of shared memory, which is similar to cuFindIBySharedMemory

kernel.

the local matrix set *𝐴̂*, is computed by kernel with shared memory or **Constructing the local submatrix:**Using *J* and *I* obtained above,

[Fig.](#_bookmark12) [5](#_bookmark12) shows a segment of the decision tree for constructing *𝐴̂*. When 4 kernel with global memory according to the established decision tree.

*<* n2max ≤ 8, cuComputeTildeABySharedMemory kernel with shared

memory will be selected according to different *𝑛*1*𝑚𝑎𝑥*. [Fig.](#_bookmark13) [6](#_bookmark13) shows the memory of *sharedSize* size or cuComputeTildeA kernel with global

thread group on each GPU that calculates *𝐴̂*, all threads in the thread main procedure of cuComputeTildeABySharedMemory kernel. For the group first read values in *𝐼𝑘* into shared memory *𝑠𝐼* in parallel, and *𝐴̂* is constructed on global memory by loading columns indexed in *𝐽𝑘* and matching them to *𝐼𝑘* in parallel. When *n1max >* 256, cuComputeTildeA

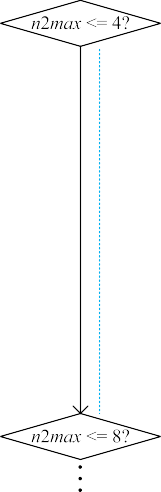
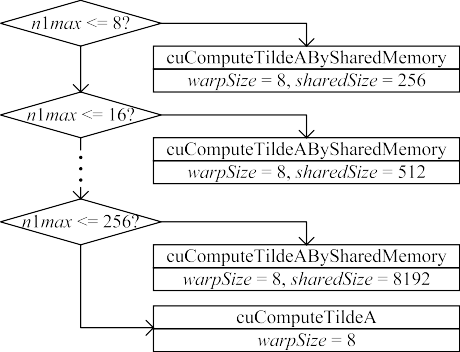
*𝑘*

*𝑘*

kernel is executed on global memory instead of shared memory, which

is similar to cuComputeTildeABySharedMemory kernel.

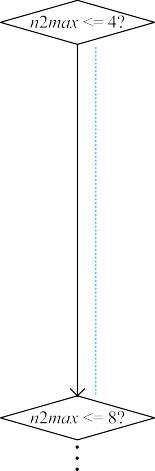
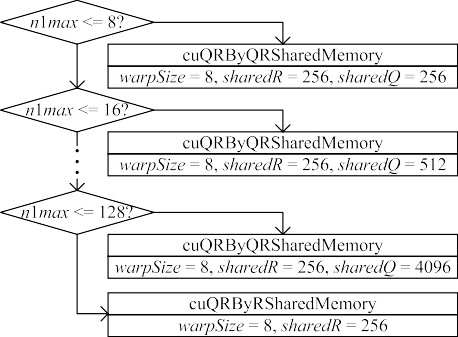
**Decomposing the Local Submatrix into *QR*:**The thread-group size of decomposing the local submatrix into *QR* is same in all blocks. Being similar with above two steps, the constructed decision tree is used again



**/ig. 5.** A segment of the decision tree to construct *𝐴̂*.



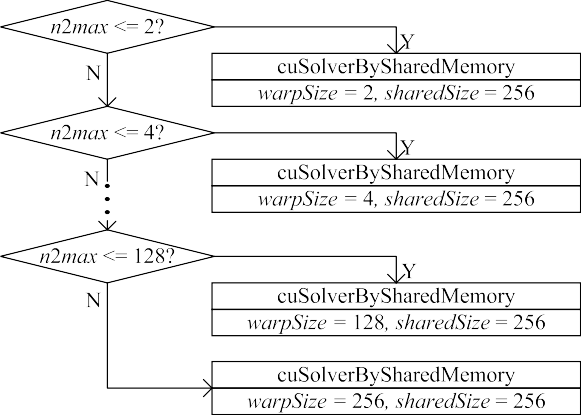
**/ig. 6.** Main procedure of cuComputeTildeABySharedMemory kernel.



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

to decompose local submatrix. [Fig.](#_bookmark14) [7](#_bookmark14) shows a segment of the decision

**/ig. 8.** Main procedure of cuQRByQRSharedMemory kernel.



**/ig. 9.** A segment of the decision tree to solve the upper triangular linear system.

the first step, the *i*th column of *𝑄𝑘* are read into shared memory *sQ* in parallel. In the second step, the threads computed the *i*th row of the

upper triangle matrix *𝑅𝑘* in parallel and put into shared memory *sR*. In the third step, the column *i* of *𝑄𝑘* and *sQ* are concurrently normalized, and the projection factors *𝑅𝑘* and *sR* are calculated. In the fourth step, the values of all columns of *𝑄𝑘* 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 *𝑚̂𝑘* = *𝑅*−1*𝑄𝑇 𝑒̂𝑘* are computed by solving an upper trian-

*𝑘 𝑘*

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.](#_bookmark15) [8](#_bookmark15) shows the main procedure of cuQRByQRSharedMemory kernel. In addition, Eeach thread group is responsible for one *QR* decomposi- tion. For a description of its detailed implementation, please refer to

the literature [[25](#_bookmark38)]. In a thread group, the local submatrix, e.g., *𝐴̂*, is

*𝑘*

decomposed into *QR* by the following four steps at each iteration *i*. In

gular linear system. [Fig.](#_bookmark16) [9](#_bookmark16) 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.](#_bookmark18) [10](#_bookmark18) shows the main procedure of cuSolverBySharedMemory kernel. For each thread group,T the steps

to compute *𝑚̂* , e.g., *𝑚̂𝑘*, include: ([1](#_bookmark4)) *𝑄𝑇 𝑒̂𝑘* is calculated in parallel and

saved to the shared memory *𝑥𝐸*, and (2) the values of *𝑚̂𝑘* are obtained

*𝑘*

**Table 3**

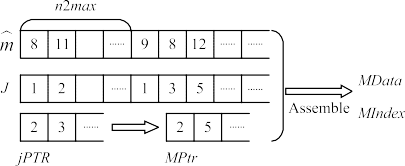


Iterations and execution time of GPUBICGSTAB on GTX 1080 Ti. Matrix GPUBICGSTAB GPUPBICGSTAB

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 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 |

**/ig. 10.** Main procedure of cuSolverBySharedMemory kernel.

**Table 4**

Execution time of HybridSPAI and GPUPBICGSTAB.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | Matrix | GPU | 1 thread | 2 thread | 4 thread | 8 thread |
|  | 0.506 | 0.501 | 0.262 | 0.151 | 0.102 |
| venkat01 | 0.806 | 0.771 | 0.761 | 0.736 | 0.715 |
|  | 1.312 | 1.272 | 1.023 | 0.887 | 0.817 |
|  |  |  |  |  |  |  |  | 0.228 | 0.227 | 0.179 | 0.103 | 0.104 |
|  |  |  | |  | |  | imagesensor | 0.808 | 0.785 | 0.767 | 0.745 | 0.713 |
|  |  |  | |  | |  |  | 1.036 | 1.012 | 0.946 | 0.848 | 0.817 |
|  |  |  | |  | |  |  | 1.187 | 1.191 | 0.631 | 0.356 | 0.224 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **/ig. 11.** | Assemble | *M*. |  |  |  |  | cfd2 | 2.331 | 2.231 | 2.294 | 2.178 | 2.101 |
|  |  |  |  |  |  |  |  | 3.518 | 3.422 | 2.925 | 2.534 | 2.325 |
| **Table 2** |  |  |  |  |  |  |  | 0.226 | 0.219 | 0.126 | 0.101 | 0.133 |
| Descriptions of test matrices. |  |  |  |  |  |  | apache2 | 2.806 | 2.761 | 2.746 | 2.734 | 2.838 |
| Name Kind | Rows | Nonzeros | *avg* | *max* | *min* |  |  | 3.032 | 2.980 | 2.872 | 2.835 | 2.971 |
| venkat01 CFD sequence | 62,424 | 1,717,792 | 27.52 | 44 | 16 |  |  | 0.075 | 0.070 | 0.060 | 0.064 | 0.103 |
| imagesensor Semiconductor device | 118,758 | 1,446,396 | 12.18 | 21 | 2 |  | t2em | 2.263 | 2.253 | 2.241 | 2.231 | 2.268 |
| cfd2 CFDproblem | 123,440 | 3,085,406 | 25.00 | 30 | 8 |  |  | 2.338 | 2.323 | 2.301 | 2.295 | 2.371 |

G3\_circuit Circuitsimulation 1,585,478 7,660,826 4.83 6 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 |

by solving the upper triangular linear system, *𝑅𝑘𝑚̂𝑘* = *xE*, in parallel using shared memory.

thermal2

G3\_circuit

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

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

* 1. *Post-HybridSPAI Stage*

The Post-HybridSPAI Stage is to assemble *M* in the CSC storage format from multiple GPUs. [Fig.](#_bookmark20) [11](#_bookmark20) illustrates the procedure of as- sembling *MPtr*, *MIndex* and *MData* arrays on each GPU. First, *MPtr*

is assembled utilizing *jPTR*. Second, *𝑚̂* 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.

# Evaluation and analysis

We evaluate the performance of HybridSPAI in this section. The test matrices in [Table](#_bookmark21) [2](#_bookmark21) 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.

* 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 1*𝑒*−7,

or the number of iterations exceeds 10,000. [Table](#_bookmark17) [3](#_bookmark17) 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 Hy- bridSPAI and GPUPBICGSTAB with HybridSPAI. [Table](#_bookmark19) [4](#_bookmark19) demonstrates

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 1*𝑒*−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.

* 1. *Performance comparison*

We test the HybridSPAI performance by comparing it with a pop- ular preconditioning algorithms: CSRILU0 in CUSPARSE (denoted by CSRILU) [[32](#_bookmark45)], a static sparse approximate inverse preconditioning algorithm in ViennaCL (denoted by SSPAI-VCL) [[33](#_bookmark46)], and a latest paral- lel SPAI preconditioning algorithm(denoted by GSPAI-Adaptive) [[29](#_bookmark42)]. [Table](#_bookmark23) [5](#_bookmark23) 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

1*𝑒*−7. Note that ‘‘/’’ represents the number of iterations for HybridSPAI

except that the third row is denoted by ‘‘*>* 10000’’. The minimum value exceeds 10,000, and all the other rows for each matrix will be denoted

of the fourth row for each matrix is marked in the red font.

From [Table](#_bookmark23) [5](#_bookmark23), 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 Hybrid- SPAI 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 |
| 1.835 | 38.856 | 0.506 | 0.102 |
| 1.574 | 0.036 | 0.806 | 0.715 |
| venkat01 11 | 48 | 35 | 35 |
| 3.427 | 38.892 | 1.312 | 0.817 |
| / | / | 0.228 | 0.104 |
| imagesensor / | / | 0.808 | 0.713 |
| 10000 | 10000 | 52 | 52 |
| / | / | 1.036 | 0.817 |
| / | / | 1.187 | 0.224 |
| / | / | 2.331 | 2.101 |
| cfd2 10000 | 10000 | 1613 | 1613 |
| / | / | 3.518 | 2.325 |
| 3.386 | 43.532 | 0.226 | 0.101 |
| apache2 6.776 | 2.995 | 2.806 | 2.734 |
| 475 | 2503 | 1106 | 1106 |
| 10.162 | 46.527 | 3.032 | 2.835 |
| 19.884 | / | 0.075 | 0.064 |
| t2em 2998.63 | / | 2.263 | 2.231 |
| 427 | 10000 | 768 | 768 |
| 3018.514 | / | 2.338 | 2.295 |
| 5.502 | / | 0.332 | 0.164 |
| 45.008 | / | 9.416 | 9.187 |
| thermal2 1619 | 10000 | 2584 | 2584 |
| 50.510 | / | 9.748 | 9.351 |
| 5.245 | / | 0.167 | 0.115 |
| G3\_circuit 12.475 | / | 2.363 | 2.290 |
| 257 | 10000 | 475 | 475 |
| 17.720 | / | 2.530 | 2.405 |

# Conclusion

We present an efficient parallel sparse approximate inverse precon- ditioning 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](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb1) [solving nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 7 (1986)](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb1) [856–869.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb1)
2. [H.A: Bi-CGSTAB: a fast and smoothly converging variant of BiCG for the solution](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb2) [of nonsymmetric linear systems, SIAM J. Stat. Comput. 13 (2) (1992) 631.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb2)
3. [Y. Saad, Iterative Methods for Sparse Linear Systems, second version, SIAM,](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb3) [Philadelphia, PA, 2003.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb3)
4. [J. Gao, R. Liang, J. Wang, Research on the conjugate gradient algorithm with a](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb4) [modified incomplete cholesky preconditioner on GPU, J. Parallel Distr. Com. 74](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb4)

[(2) (2014) 2088–2098.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb4)

1. [S.C. Rennich, D. Stosic, T.A. Davis, Accelerating sparse Cholesky factorization on](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb5) [GPUs, Parallel Comput. 59 (2016) 140–150.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb5)
2. [H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, M. Kohler,](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb6) [Preconditioned Krylov solvers on GPUs, Parallel Comput. 68 (2017) 32–44.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb6)
3. [L.Y. Kolotilina, A.Y. Yeremin, Factorized sparse approximate inverse precondi-](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb7) [tioning I. theory, SIAM J. Matrix Anal. Appl. 14 (1) (1993) 45–58.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb7)
4. [M. Benzi, C.D. Meyer, M. Tuma, A sparse approximate inverse preconditioner for](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb8) [the conjugate gradient method, SIAM J. Sci. Comput. 17 (5) (1996) 1135–1149.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb8)
5. [M. Ferronato, C. Janna, G. Pini, A generalized block FSAI preconditioner for](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb9) [nonsymmetric linear systems, J. Comput. Appl. Math. 256 (2014) 230–241.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb9)
6. V.A.P. Magri, A. Franceschini, M. Ferronato, C. Janna, Multilevel approaches for FSAI preconditioning, Numer. Linear Algebr. (2018) e2183, [http://dx.doi.org/](http://dx.doi.org/10.1002/nla.2183) [10.1002/nla.2183](http://dx.doi.org/10.1002/nla.2183).
7. [M.M. Dehnavi, D.M. Fern´andez, J.L. Gaudiot, D.D. Giannacopoulos, Parallel](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb11) [sparse approximate inverse preconditioning on graphic processing units, IEEE](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb11)

[T. Parall. Distr. 24 (9) (2013) 1852–1861.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb11)

1. [Z. Jia, B. Zhu, A power sparse approximate inverse preconditioning procedure](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb12) [for large sparse linear systems, Numer. Linear Algebr. 16 (4) (2009) 259–299.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb12)
2. [J.D.F. Cosgrove, J.C. Diaz, A. Griewank, Approximate inverse preconditioning](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb13) [for sparse linear systems, Int. J. Comput. Math. 44 (1–2) (1992) 91–110.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb13)
3. [M. Grote, T. Huckle, Parallel preconditioning with sparse approximate inverses,](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb14) [SIAM J. Sci. Comput. 18 (3) (1997) 838–853.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb14)
4. [E. Chow, Y. Saad, Approximate inverse preconditioners via sparse-sparse](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb15) [iterations, SIAM J. Sci. Comput. 19 (3) (1998) 995–1023.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb15)
5. [E. Chow, A priori sparsity patterns for parallel sparse approximate inverse](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb16) [preconditioners, SIAM J. Sci. Comput. 21 (5) (2000) 1804–1822.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb16)
6. [E. Chow, A. Patel, Fine-grained parallel incomplete LU factorization, SIAM J.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb17) [Sci. Comput. 37 (2) (2015) C169–C193.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb17)
7. [J. Gao, Z. Li, R. Liang, G. He, Adaptive optimization l1-minimization solvers on](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb18) [GPU, Int. J. Parallel Program. 45 (3) (2017) 508–529.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb18)
8. [K. Li, W. Yang, K. Li, A hybrid parallel solving algorithm on GPU for quasitridi-](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb19) [agonal system of linear equations, IEEE Trans. Parallel Distrib. Syst. 27 (10)](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb19) [(2016) 2795–2808.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb19)
9. [J. Gao, Y. Zhou, G. He, Y. Xia, A multi-GPU parallel optimization model for the](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb20) [preconditioned conjugate gradient algorithm, Parallel Comput. 63 (2017) 1–16.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb20)
10. 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/](http://dx.doi.org/10.1002/cpe.4705) [cpe.4705](http://dx.doi.org/10.1002/cpe.4705).
11. [J. Gao, K. Wu, Y. Wang, P. Qi, G. He, GPU-accelerated preconditioned GMRES](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb22) [method for two-dimensional Maxwell’s equations, Int. J. Comput. Math. 94 (10)](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb22) [(2017) 2122–2144.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb22)
12. [M. Lukash, K. Rupp, S. Selberherr, Sparse approximate inverse preconditioners](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb23) [for iterative solvers on GPUs, in: Proceedings of the 2012 Symposium on High](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb23) [Performance Computing, Society for Computer Simulation, San Diego, CA, USA,](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb23) [2012, pp. 1–8.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb23)
13. [K. Rupp, R. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jungel, S.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb24) [Selberher, ViennaCL-linear algebra library for multi-and many-core architectures,](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb24) [SIAM J. Sci. Comput. 38 (5) (2016) S412–S439.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb24)
14. [G. He, R. Yin, J. Gao, An efficient sparse approximate inverse preconditioning](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb25) [algorithm on GPU, Concurr. Comput.-Pract. Exp. 32 (7) (2020) e5598.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb25)
15. J. Gao, Q. Chen, G. He, A thread-adaptive sparse approximate inverse pre- conditioning algorithm on multi-GPUs, Parallel Comput. 101 (2021) 102724, <http://dx.doi.org/10.1016/j.parco.2020.102724>.
16. [J. Gao, K. Wu, Y. Wang, P. Qi, G. He, GPU-accelerated preconditioned GMRES](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb27) [method for two-dimensional Maxwell’s equations, Int. J. Comput. Math. 94 (10)](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb27) [(2017) 2122–2144.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb27)
17. G. He, R. Yin, J. Gao, An efficient sparse approximate inverse preconditioning algorithm on GPU, Concurr. Comput.-Pract. Exp. 32 (7) (2020) e5598, [http:](http://dx.doi.org/10.1002/cpe.5598)

[//dx.doi.org/10.1002/cpe.5598](http://dx.doi.org/10.1002/cpe.5598).

1. J. Gao, Q. Chen, G. He, A thread-adaptive sparse approximate inverse pre- conditioning algorithm on multi-GPUs, Parallel Comput. 101 (2021) 102724, <http://dx.doi.org/10.1016/j.parco.2020.102724>.
2. NVIDIA, Cusparse library, 2019, v10.1, [https://docs.nvidia.com/cuda/cusparse/](https://docs.nvidia.com/cuda/cusparse/index.html) [index.html](https://docs.nvidia.com/cuda/cusparse/index.html).
3. 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>.
4. Cusparse library, v10.1. <https://docs.nvidia.com/cuda/cusparse/index.html>.
5. [K. Rupp, et al., ViennaCl–linear algebra library for multi-and many-core](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb33) [architectures, SIAM J. Sci. Comput. 38 (5) (2016) S412–S439.](http://refhub.elsevier.com/S2772-4859(21)00007-7/sb33)