[BenchCouncil Transactions on Benchmarks, Standards and Evaluations 2 (2022) 100087](https://doi.org/10.1016/j.tbench.2023.100087)



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

BenchCouncil Transactions on Benchmarks,

Standards and Evaluations

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

Research Article

[](http://crossmark.crossref.org/dialog/?doi=10.1016/j.tbench.2023.100087&domain=pdf)Optimizing the sparse approximate inverse preconditioning algorithm on GPU[✩](#_bookmark0)

Xinyue Chu, Yizhou Wang, Qi Chen, 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

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.

# Introduction

Given their many-core structures, graphic processing units (GPUs) have become an important resource for scientific computing in re- cent years. Following the introduction of the programming interfaces such as the compute unified device architecture (CUDA) by NVIDIA in 2007 [[1](#_bookmark21)], GPUs have been increasingly used as tools for high- performance computation in many fields [[2](#_bookmark22)–[8](#_bookmark23)].

Sparse approximate inverse (SPAI) preconditioners based on the Frobenius norm minimization have proven to be effective in improv- ing the convergence of iterative methods based on Krylov subspaces, e.g., the generalized minimal residual method (GMRES) [[9](#_bookmark24)] and the biconjugate gradient stabilized method (BiCGSTAB) [[10](#_bookmark25)]. However, due to the high cost of constructing the SPAI preconditioners, many researchers have attempted to accelerate the SPAI preconditioner con- struction on GPU. Gao et al. follow Chow’s work [[11](#_bookmark26)], and use a

sparse approximate inverse of *𝐴* as the preconditioner in [[12](#_bookmark27)]. Rupp

et al. [[13](#_bookmark28)] show several static and dynamic SPAI implementations on

GPU. In [[14](#_bookmark29)], Dehnavi et al. propose a static SPAI preconditioner on GPU called GSAI. Recently, He and Gao et al. [[15](#_bookmark30)] 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 precondi- tioner has significant difference, the performance of SPAI-Adaptive is greatly decreased. Furthermore, He and Gao et al. [[16](#_bookmark31)] 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

formance than GSPAI-Adaptive. For example, assuming that *𝑛*2*𝑘* is the preconditioner has significant difference, SPAI-Adaptive has worse per- nonzero number of the *𝑘*th column of the preconditioner, *𝑛*2*𝑚𝑎𝑥* = max*𝑘*{*𝑛*2*𝑘*}, and *𝑛*2*𝑎𝑣𝑔* = ∑*𝑛 𝑛*2*𝑘*∕*𝑛*, where *𝑛* is the row number of the preconditioner, we take two integers *𝛼* and *𝛽*, which satisfy 2*𝛼*−1 *<*

*𝑘*=1

*𝑛*2*𝑚𝑎𝑥* ⩽ 2*𝛼* and 2*𝛽*−1 *< 𝑛*2*𝑚𝑎𝑥* ⩽ 2*𝛽* , respectively. If *𝛼* = *𝛽*, we

preconditioner is small; if *𝛼* − *𝛽* ⩾ 3, we say that the nonzero number of say that the difference in the nonzero number of each column of the

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 ⩽ *𝛼* − *𝛽 <* 3. There are no conclusions in [[15](#_bookmark30),[16](#_bookmark31)].

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

dices *𝐼* and *𝐽* , constructing the local submatrix, decomposing the one of main components of the preconditioner such as finding in-

✩ The research has been supported by the Natural Science Foundation of China under grant number 61872422.

∗ Corresponding author.

*E-mail addresses:* [2316607219@qq.com](mailto:2316607219@qq.com) (X. Chu), [1966224230@qq.com](mailto:1966224230@qq.com) (Y. Wang), [1337223917@qq.com](mailto:1337223917@qq.com) (Q. Chen), [springf12@163.com](mailto: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

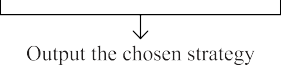
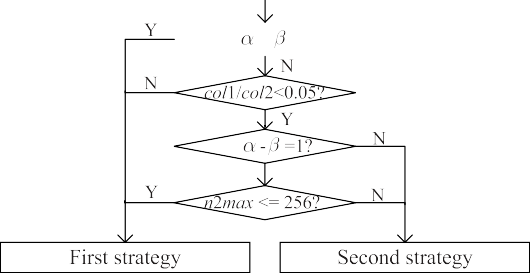
Available online 3 March 2023

2772-4859/© 2023 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-NC-ND license (<http://creativecommons.org/licenses/by-nc-nd/4.0/>).



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

**/ig. 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

posing the local submatrix into *𝑄𝑅* and solving the upper trian- one of some components of the preconditioner such as decom-

gular linear system, a decision tree is constructed to choose the optimization kernel to calculate it.

conditioner, not just the same sparsity pattern as *𝐴*. • Finally, GSPAI-Opt can apply to any sparsity pattern of the pre-

The experimental results show that GSPAI-Opt is effective, and effi- ciently fuses the advantages of SPAI-Adaptive and GSPAI-Adaptive, and outperforms the static SPAI preconditioning algorithm in the ViennaCL library [[13](#_bookmark28)], the recent SPAI-Adaptive [[15](#_bookmark30)] and GSPAI-Adaptive [[16](#_bookmark31)].

# Optimizing SPAI on GPU

We present an optimization sparse approximate inverse precondi- tioning algorithm on GPU, called GSPAI-Opt. [Fig.](#_bookmark2) [1](#_bookmark2) 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 *𝐽𝑘* and *𝐼𝑘*, construct local

**/ig. 2.** Main procedure of selecting the First/Second strategy.

* 1. *Pre-GSPAI stage*

computing *𝑚𝑘* (one column of *𝑀* ), *𝑘* = 1*,* 2*,* … *, 𝑛*, the dimensions of First, we compute the dimensions of all local submatrices. When the local submatrices (*𝑛*1*𝑘*, *𝑛*2*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 (*𝑛*1*𝑚𝑎𝑥*, *𝑛*2*𝑚𝑎𝑥*). Here *𝑛*1*𝑚𝑎𝑥* =

max*𝑘*{*𝑛*1*𝑘*} and *𝑛*2*𝑚𝑎𝑥* = max*𝑘*{*𝑛*2*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](#_bookmark30)] and GSPAI-Adaptive [[16](#_bookmark31)]. For SPAI-Adaptive, a thread-adaptive allocation strategy with the con- stant 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.

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

following formula:

*𝑞* = min(2*𝑠, 𝑛𝑡*)*,* (1)

*𝑠.𝑡.*

submatrix *𝐴̂* , decompose *𝐴̂*

into *𝑄 𝑅* , and solve *𝑅 𝑚̂*

= *𝑄𝑇 𝑒̂* ;

*𝑘 𝑘*

*𝑘 𝑘*

*𝑘 𝑘*

*𝑘 𝑘*

2*𝑠*−1 *< 𝑛*2*𝑚𝑎𝑥* ⩽ 2*𝑠.* (2)

* *Post-GSPAI* stage: Assemble the preconditioner *𝑀* in the com-

pressed 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 suit- able for computing the preconditioner, the thread-adaptive allocation strategy (First strategy) proposed in [[15](#_bookmark30)] is adopted; otherwise, the thread-adaptive allocation strategy with the nonconstant thread-group size (Second strategy) proposed in [[16](#_bookmark31)] is utilized. Given a matrix, should we use the first strategy or the second strategy? Here we present

Here *𝑛𝑡* is the number of threads per block, and *𝑞* threads are grouped

into a thread group.

***group size***: First, for each *𝑛*2*𝑘*, *𝑘* = 1*,* 2*,* … *, 𝑛*, the number of threads *𝑞𝑘* ***Thread-adaptive allocation strategy with the nonconstant thread-*** assigned to the *𝑘*th column of the preconditioner is computed by the

following formula:

*𝑞𝑘* = min(2*𝑠, 𝑛𝑡*)*,* (3)

*𝑠.𝑡.*

a selection method, whose main procedure is shown in [Fig.](#_bookmark3) [2](#_bookmark3).

Let us illustrate the selection method in [Fig.](#_bookmark3) [2](#_bookmark3) by apache2. For

2*𝑠*−1 *< 𝑛*2*𝑘*

⩽ 2*𝑠.* (4)

apache2, we have *n2max* = 8 and *n2avg* = 6.74. Obviously, *n2max*, *n2avg* ∈ (22*,* 23], and *𝛼* = *𝛽* = 3. Based on the selection method in

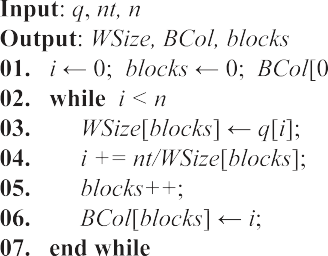


[Fig.](#_bookmark3) [2](#_bookmark3), the first strategy is chosen.

Second, all *𝑞𝑘* values are sorted in descending order. Finally, the

thread-group size of each block is assigned by the procedure shown

in [Fig.](#_bookmark4) [3](#_bookmark4).



**/ig. 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 | *𝑚̂* | ns × *𝑛*2*𝑚𝑎𝑥* | double |
| *AIndex* | nonzeros | integer | *𝐴̂* | ns × *𝑛*1*𝑚𝑎𝑥* × *𝑛*2*𝑚𝑎𝑥* | double |
| *APtr* | *𝑛* | integer | *𝑅* | ns × *𝑛*2*𝑚𝑎𝑥* × *𝑛*2*𝑚𝑎𝑥* | double |
| *RCol* | *𝑛* | integer | *𝐼* | ns × *𝑛*1*𝑚𝑎𝑥* | integer |
| *atomic* | *𝑛* | integer | *iPTR* | ns | integer |
| *WSize* | *𝑏𝑙𝑜𝑐𝑘𝑠* | integer | *𝐽* | ns × *𝑛*2*𝑚𝑎𝑥* | integer |
| *BCol* | *𝑏𝑙𝑜𝑐𝑘𝑠* | integer | *jPTR* | ns | integer |

Finally, we allocate global memory for arrays in [Table](#_bookmark5) [1](#_bookmark5), and *RCol*, *BCol*, and *WSize* values are transferred to the GPU global memory if the second strategy is applied.

* 1. *Compute-GSPAI stage*

**/inding indices:** This part is to find indices *𝐽* and *𝐼* by the con- stant/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 *𝑛*2*𝑚𝑎𝑥* and *𝑛*1*𝑚𝑎𝑥*. Assume that the threads

a segment of the decision tree for finding *𝐼* . Here *𝑠ℎ𝑎𝑟𝑒𝑑𝑆𝑖𝑧𝑒* = per block are 256 and NIVIDA GTX1070 GPU is used, [Fig.](#_bookmark6) [4](#_bookmark6) shows

number of columns of the preconditioner computed in a thread block

× upper boundary closest to *𝑛*1*𝑚𝑎𝑥*. For example, when *𝑛*1*𝑚𝑎𝑥* ⩽ 8,

*𝑠ℎ𝑎𝑟𝑒𝑑𝑆𝑖𝑧𝑒* = 32 × 8 and cuFindIBySharedMemory kernel with shared

each thread group finds one subset of *𝐼* , e.g., *𝐼𝑘*, which mainly includes memory of 256 size is used. In the cuFindIBySharedMemory kernel,

indices of the first column referenced in one subset of *𝐽* , e.g., *𝐽𝑘*, to the following steps. First, the threads in the thread group load the row shared memory *𝑠𝐼* . Second, 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. Third, inside the thread group, the indices of *𝑠𝐼* are sorted in ascending order in parallel. Finally, the indices of *𝑠𝐼* are copied to *𝐼𝑘*. cuFindI kernel is

similar to cuFindIBySharedMemory kernel except that the operations

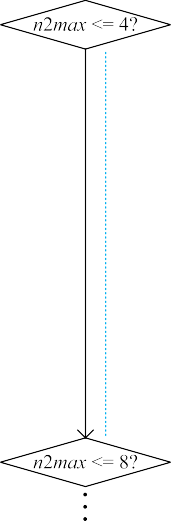
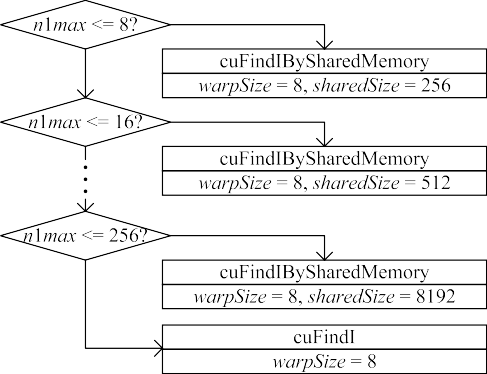
are executed on global memory instead of shared memory.

* + 1. ***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

pattern *𝑀* in parallel and store them to one subset of *𝐽* . The main threads inside each thread group read one column of the sparsity procedure of the kernel that finds *𝐼* is as same as that in [[16](#_bookmark31)]. Each thread group is assigned to find one subset of *𝐼* , e.g., *𝐼𝑘*, which includes

the thread-group size *𝑤𝑎𝑟𝑝𝑆𝑖𝑧𝑒*. In the second stage, the row indices the following three stages. In the first stage, the thread group obtains

**/ig. 4.** A segment of the decision tree of using constant threads to find *𝐼* .



of the first column referenced in *𝐽𝑘* are first loaded into *𝐼𝑘*, and the row index vectors of successive columns that are referenced by *𝐽𝑘* are calculated in parallel with values in *𝐼𝑘*, and the new indices are appended to *𝐼𝑘* by utilizing the atomic operations. In the third stage, the indices in *𝐼𝑘* are sorted in ascending order in parallel.

**Constructing the local submatrix:** Using *𝐽* and *𝐼* obtained above,

the local submatrix set, *𝐴̂*, is computed by the constant/nonconstant

thread-group size.

***size***: Each thread group is assigned to compute one subset of *𝐴̂*, e.g., *𝐴̂* , **(1) *Constructing the local submatrix by the constant thread-group***

*𝑘*

parameters, we establish a decision tree for constructing *𝐴̂*. For any and all thread groups are the same size. Based on the GPU feature given *𝑛*2*𝑚𝑎𝑥* and *𝑛*1*𝑚𝑎𝑥*, an optimized kernel for constructing *𝐴̂* is

achieved by using the decision tree. For example, on NIVIDA GTX1070

segment of the decision tree for constructing *𝐴̂*. When 4 *< 𝑛*2*𝑚𝑎𝑥* ⩽ 8, GPU, assume that the threads per block are 256, [Fig.](#_bookmark7) [5](#_bookmark7) shows a corresponding to different *𝑛*1*𝑚𝑎𝑥*, 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 *𝐴̂* , all threads in the thread group

first read values in *𝐼𝑘* into shared memory *𝑠𝐼* in parallel, and *𝐴̂* is

*𝑘*

*𝑘*

by *𝐽𝑘* and matching them to *𝑠𝐼* in parallel. cuComputeTildeA kernel established on the global memory by loading columns that are indexed 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-*

subset of *𝐴̂*, e.g., *𝐴̂* , and the thread-group size is same in a block but it ***group size***: In this case, each thread group is assigned to calculate one

*𝑘*

that constructs *𝐴̂* is as same as that in [[16](#_bookmark31)]. can be different for different block. The main procedure of the kernel

**Decomposing the local submatrix into** *𝑄𝑅***:** This part is used to

decompose the local submatrix into *𝑄𝑅* by the constant/nonconstant

thread-group size.

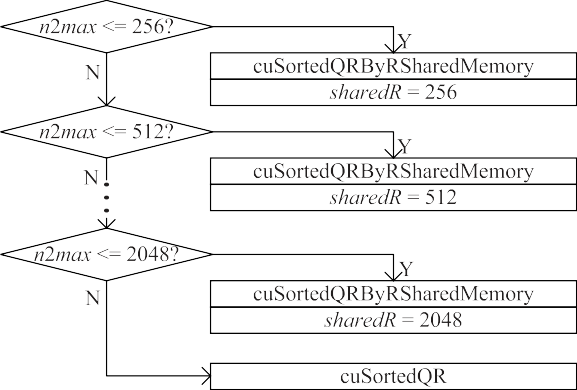
## *Decomposing the local submatrix into* 𝑄𝑅 *by the constant*

submatrix into *𝑄𝑅* is same in all blocks. Based on the GPU feature ***thread-group size***: The thread-group size of decomposing the local

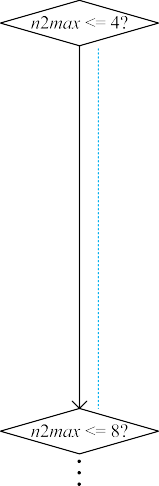
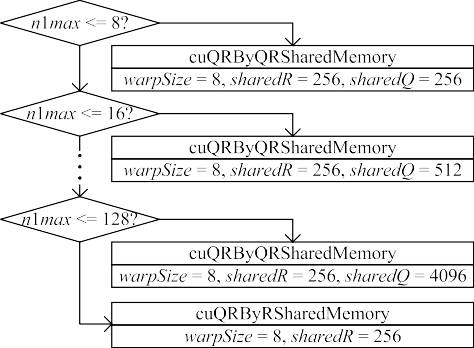
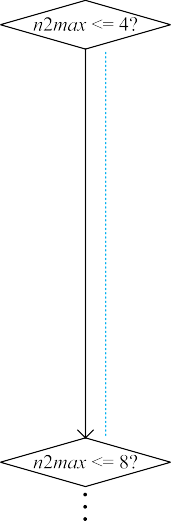
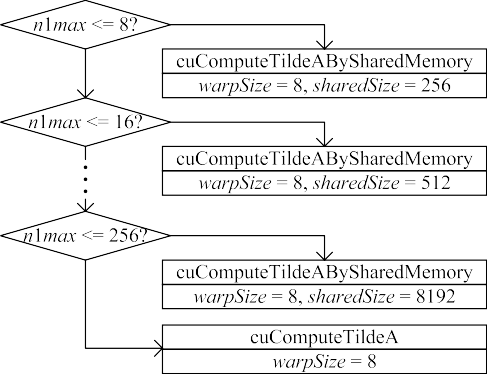
submatrix into *𝑄𝑅*. For example, on NIVIDA GTX1070 GPU, assume parameters, we establish a decision tree for decomposing the local

decision tree for decomposing the local submatrix into *𝑄𝑅*. When that the threads per block are 256, [Fig.](#_bookmark8) [6](#_bookmark8) shows a segment of the

4 *< 𝑛*2*𝑚𝑎𝑥* ⩽ 8, two shared memories *𝑠ℎ𝑎𝑟𝑒𝑑𝑅* and *𝑠ℎ𝑎𝑟𝑒𝑑𝑄* are

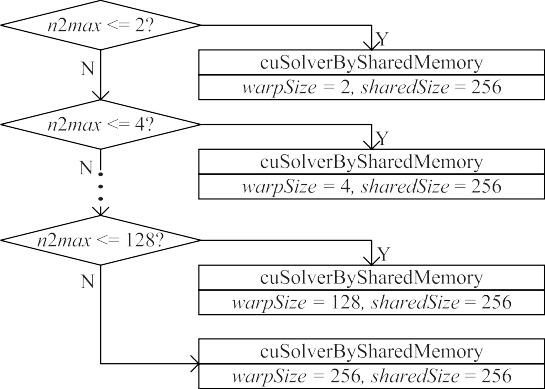


**/ig. 5.** A segment of the decision tree of using constant threads to construct *𝐴̂*.



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

into *𝑄𝑅*. **/ig. 7.** Decision tree of using nonconstant threads to decompose the local submatrix



**/ig. 8.** Decision tree of using constant threads to solve the upper triangular linear system.

in [Fig.](#_bookmark9) [7](#_bookmark9) when the threads per block are 256. Obviously, utilizing

corresponding to shared memory of *𝑠ℎ𝑎𝑟𝑒𝑑𝑅* size or cuSortedQR kernel the decision tree, an optimized kernel cuSortedQRByRSharedMemory is chosen for a given *𝑛*2*𝑚𝑎𝑥* value. The main procedure of cuSort-

edQRByRSharedMemory kernel is as same as that in [[16](#_bookmark31)]. 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 *𝑚̂𝑘* =

*𝑅*−1*𝑄𝑇 𝑒̂𝑘* are computed by the constant/nonconstant thread-group size.

*𝑘 𝑘*

## *Solving the upper triangular linear system by the constant*

used in the optimized kernel. Here the size of *𝑠ℎ𝑎𝑟𝑒𝑑𝑄* is related to

*𝑛*1*𝑚𝑎𝑥*. In the cuQRByQRSharedMemory kernel, each thread group is responsible for one *𝑄𝑅* decomposition. In a thread group, the local submatrix, e.g., *𝐴̂* , is decomposed into *𝑄𝑅* by the following four steps at each iteration *𝑖*. In the first step, the threads read the *𝑖*th column of *𝑄𝑘* into shared memory *𝑠𝑄* in parallel. In the second step, the *𝑖*th row of the upper triangle matrix *𝑅𝑘* are computed in parallel and are 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.

*𝑘*

kernel except the shared memory *𝑠𝑄* is not utilized. cuQRByRSharedMemory kernel is similar to cuQRByQRSharedMemory

## *Decomposing the local submatrix into* 𝑄𝑅 *by the nonconstant*

submatrix into *𝑄𝑅* is same in a block while it is usually different ***thread-group size***: The thread-group size of decomposing the local

local submatrix into *𝑄𝑅*. For example, on NIVIDA GTX1070 GPU, the for different blocks. We establish a decision tree for decomposing the decision tree for decomposing the local submatrix into *𝑄𝑅* is shown

***thread-group size***: Each thread group computes one subset of *𝑚̂* 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.](#_bookmark10) [8](#_bookmark10). For any given *𝑛*2*𝑚𝑎𝑥* value, an optimized kernel,

group size of *𝑤𝑎𝑟𝑝𝑆𝑖𝑧𝑒*, is chosen. In the cuSolverBySharedMemory cuSolverBySharedMemory with shared memory of 256 size and thread- kernel, each thread group calculates a subset of *𝑚̂*, e.g., *𝑚̂𝑘*, and its procedure includes two steps. First, Calculate *𝑄𝑇 𝑒̂𝑘* in parallel and save the result to the shared memory *𝑥𝐸*. Second, the values of *𝑚̂𝑘* are obtained by solving the upper triangular linear system *𝑅𝑘𝑚̂𝑘* = *𝑥𝐸*, in

*𝑘*

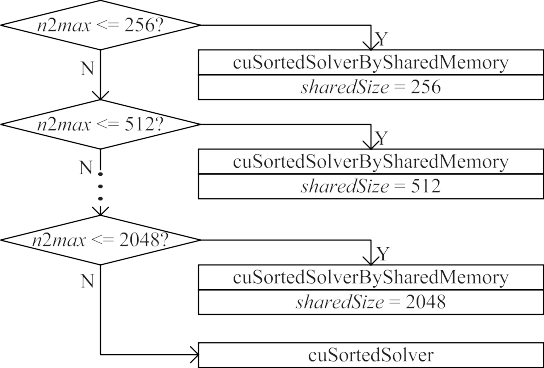
parallel.

**(2) *Solving the upper triangular linear system by the nonconstant thread-group size***: Each thread group is responsible for obtaining a

subset of *𝑚̂* 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.](#_bookmark11) [9](#_bookmark11) lists the decision tree for solving the upper triangular linear system on NIVIDA GTX1070 GPU. For any

**/ig. 9.** Decision tree of using nonconstant threads to solve the upper triangular linear system.

**Table 2**

Overview of GPUs.

|  |  |  |
| --- | --- | --- |
| Hardware | GTX1070 | A40 |
| Cores | 1920 | 10 752 |
| 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 *𝑛*2*𝑚𝑎𝑥* 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](#_bookmark31)].

kernel except that the shared memory *𝑥𝐸* is not used. cuSortedSolver kernel is similar to cuSortedSolverBySharedMemory

* 1. *Post-GSPAI stage*

In the *Post-GSPAI* stage, the preconditioner *𝑀* is assembled in the CSC storage format which contains three arrays of *𝑀𝑃 𝑡𝑟*, *𝑀𝐼𝑛𝑑𝑒𝑥* and

*𝑀𝐷𝑎𝑡𝑎*. [Fig.](#_bookmark13) [10](#_bookmark13) illustrates the procedure of assembling these arrays. First, *MPtr* is assembled utilizing *jPTR*. Second, *𝑀𝐷𝑎𝑡𝑎* and *𝑀𝐼𝑛𝑑𝑒𝑥* are assembled using *𝑚̂* and *𝐽* . In order to reduce the cost of array transfer,

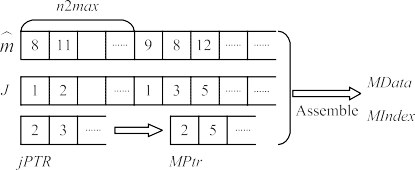
thread group is responsible for generating one *𝑚̂𝑘* to *𝑀𝐷𝑎𝑡𝑎* and one we assemble all arrays mentioned above on the GPU memory, and each

*𝐽𝑘* to *𝑀𝐼𝑛𝑑𝑒𝑥*.

# Experimental results

In this section, we take two NVIDIA GPUs (GTX1070 and A40) shown in [Table](#_bookmark12) [2](#_bookmark12) to evaluate the performance of GSPAI-Opt. The test matrices are listed in [Table](#_bookmark14) [3](#_bookmark14), which are chosen from the SuiteSparse Matrix Collection [[17](#_bookmark32)], and have been widely used in some publica- tions [[14](#_bookmark29)–[16](#_bookmark31)]. [Table](#_bookmark14) [3](#_bookmark14) summarizes the information of the sparse matri- ces, including the name, kind, number of rows, total number of nonze- ros, average number of nonzeros, maximum number of nonzero entries

**/ig. 10.** Assemble *𝑀* .

of columns, and minimum number of nonzero entries of columns. The matrices in [Table](#_bookmark14) [3](#_bookmark14) are chosen due to the following reasons. The matri- ces 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 (*𝛼* −*𝛽* ⩾ 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

(*𝛼* = *𝛽*). The matrices such as msdoor and thermal2 are chosen to

test whether the predicted strategy is well matched with the measured

the preconditioner is large but not significant (1 ⩽ *𝛼*−*𝛽 <* 3). The source strategy when the difference in the nonzero number of each column of

codes are compiled and executed using the CUDA toolkit 11.1 [[18](#_bookmark33)].

* 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](#_bookmark29)–[16](#_bookmark31)],

(*𝐸* + |*𝐴*|)*𝑘, 𝑘* = 1*,* 2. The matrices in [Table](#_bookmark14) [3](#_bookmark14) are used as the test

matrices. For all test matrices, both the optimal strategy predicted by

tests are shown in [Table](#_bookmark15) [4](#_bookmark15). Note that if |*𝑡*1 − *𝑡*2|∕ max(*𝑡*1*, 𝑡*2) ⩽ 0*.*05, both the proposed selection method and the strategy obtained from actual

otherwise, the strategy corresponding to min(*𝑡*1*, 𝑡*2) is chosen as the S1 and S2 can be considered as the measured optimization strategy; measured optimization one. Here *𝑡*1 and *𝑡*2 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.

* 1. *Performance comparison*

take the sparsity pattern (*𝐸* + |*𝐴*|) to compare it with a static SPAI In order to test the effectiveness of our proposed GSPAI-Opt, we

preconditioning algorithm in ViennaCL (denoted by SSPAI-VCL) [[13](#_bookmark28)], and two recent SPAI preconditioning algorithms SPAI-Adaptive [[15](#_bookmark30)] and GSPAI-Adaptive [[16](#_bookmark31)] on GTX1070 and A40, and their comparison results are listed in [Tables](#_bookmark17) [5](#_bookmark17) and [6](#_bookmark16), 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](#_bookmark18) [7](#_bookmark18). 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.

**Table 6**

Comparison of four algorithms with (*𝐸* + |*𝐴*|) on A40.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Matrix | (*𝐸* + |*𝐴*|) | | (*𝐸* + |*𝐴*|)2 | |  | Matrix | SSPAI-V | SPAI-A | GSPAI-A | GSPAI-Opt |
|  | Predicted Measured | | Predicted Measured | |  |  | N/A | 3.717 | 1.167 | 0.878 |
| cbuckle | S2 | S2 | S2 | S2 | cbuckle | | N/A | 0.272 | 0.318 | 0.236 |
| 2cubes\_sphere | S1 | S1 | S1 | S1/S2 |  | | N/A | 96 | 96 | 96 |
|  |  |  |  |  |  | |  |  |  |  |
| offshore | S1 | S1 | S1 | S1/S2 | |  |  |  |  |  |
| ASIC\_320ks | S2 | S2 | S2 | S2 | |  | 4.952 | 0.336 | 0.327 | 0.221 |
| apache2 | S1 | S1 | S1 | S1 | | 2cubes\_sphere | 0.019 | 0.236 | 0.311 | 0.317 |
| G3\_circuit | S1 | S1 | S1 | S1 | 5 | | | 4 | 4 | 4 |
| power9 | S2 | S2 | S2 | S2 | 4.971 | | | 0.572 | 0.638 | **0.538** |

N/A 3.989 1.485 **1.114**

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 N/A 7.976 2.046 1.815

offshore

ASIC\_320ks

|  |  |  |  |
| --- | --- | --- | --- |
| 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** |
| 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** |
| 4.471 | 0.122 | 0.201 | 0.088 |
| 1.646 | 1.353 | 1.709 | 1.331 |

cbuckle

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| N/A | 0.362 | 0.356 | 0.348 apach | e2 2503 | 1256 | 1256 | 1256 |
| N/A | 96 | 96 | 96 | 6.117 | 1.475 | 1.910 | **1.419** |
|  |  |  |  |  |  |  |  |

2cubes\_sphere

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | |
| 5 | 4 | 4 | 4 | / | 1.141 | 1.295 | **1.129** |
|  |  |  |  |  |  |  |  |

offshore

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0.053 | 0.323 | 0.324 | 0.327 power9 N/A | | 37 | 37 | 37 |
| 12 | 5 | 5 | 5 | N/A | 2.409 | 7.705 | **1.864** |
|  |  |  |  |  |  |  |  |

ASIC\_320ks

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | |
| N/A | 10 | 10 | 10 | N/A | 21.777 | 11.997 | **10.015** |
|  |  |  |  |  |  |  |  |

apache2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7.963 | 3.583 | 3.574 | 3.585 thermal2 *>*10 000 | | 2186 | 2186 | 2186 |
| 2503 | 1090 | 1090 | 1090 | / | 3.933 | 4.079 | **3.843** |
|  |  |  |  |  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| N/A | 65.339 | 20.396 | 15.558 |
| N/A | 3.348 | 3.821 | 3.419 |

N/A 8.338 2.402 **2.163**

7.278 0.833 0.697 0.539

0.025 0.300 0.296 0.294

7.303 1.133 0.993 **0.833**

20.468 2.177 2.052 1.380

20.521 2.500 2.376 **1.707**

N/A 5.000 1.398 0.846

N/A 0.347 0.342 0.339

N/A 5.347 1.740 **1.185**

5.722 0.238 0.328 0.222

13.685 3.821 3.902 **3.807**

/ 0.148 0.170 0.148

G3\_circuit

msdoor

|  |  |  |  |
| --- | --- | --- | --- |
| / | 0.069 | 0.106 | 0.061 |
| / | 1.072 | 1.189 | 1.068 |

|  |  |  |  |
| --- | --- | --- | --- |
| N/A | 2.073 | 7.346 | 1.519 |
| N/A | 0.336 | 0.359 | 0.345 |

|  |  |  |  |
| --- | --- | --- | --- |
| N/A | 20.142 | 9.964 | 8.225 |
| N/A | 1.635 | 2.033 | 1.790 |

*>*10 000 472 472 472

N/A 656 656 656

|  |  |  |  |
| --- | --- | --- | --- |
| / | 0.176 | 0.273 | 0.144 |
| / | 3.757 | 3.806 | 3.699 |

SPAI-Adaptive + GPUPBICGSTAB, GSPAI-Adaptive + GPUPBICGSTAB

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| G3\_circuit / | 2.887 | 2.881 | 2.885 | Fault\_639 N/A | 1149 | 1149 | 1149 |
| *>*10 000 | 468 | 468 | 468 | N/A | 68.687 | 24.217 | **18.977** |
| / | 3.035 | 3.051 | **3.033** |  | | | |
| N/A | 4.504 | 10.620 | 2.848 |
| power9 N/A | 0.436 | 0.435 | 0.418 |

|  |  |  |  |
| --- | --- | --- | --- |
| N/A | 37 | 37 | 37 |
| N/A | 4.940 | 11.055 | **3.266** |
| N/A | 59.794 | 21.374 | 20.206 |
| msdoor N/A | 5.378 | 5.373 | 5.442 |
| N/A | 892 | 892 | 892 |
| N/A | 65.172 | 26.747 | **25.648** |
| / | 0.401 | 0.527 | 0.340 |

and GSPAI-Opt + GPUPBICGSTAB are denoted by SSPAI-V, SPAI-A,

GSPAI-A and GSPAI-Opt, respectively.

From [Tables](#_bookmark17) [5](#_bookmark17) and [6](#_bookmark16), 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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| thermal2 /  *>*10 000 | 11.700  2086 | 11.696  2086 | 11.701  2086 | while GSPAI-Opt can. Especially, for the matrices with large *𝑛*2*𝑚𝑎𝑥*  value, e.g., cbuckle, power9, ASIC\_320ks, msdoor and Fault\_639, GPUP- |
| / | 12.101 | 12.223 | **12.041** | BICGSTAB with SSPAI-VCL will encounter the out-of-memory error. |
| N/A | 185.893 | 42.994 | 37.524 | Furthermore, for 2cubes\_sphere, offshore, and apache2, the total time |
| Fault\_639 N/A | 10.032 | 10.022 | 10.013 | of SSPAI-VCL and GPUPBICGSTAB on two GPUs is much more than |

|  |  |  |  |
| --- | --- | --- | --- |
| 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 GPUP- BICGSTAB 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

(*𝑠*), and the minimum value of the fourth row for each matrix is marked encountered, all rows will be denoted by ‘‘N/A’’. The time unit is second in the red font. For the convenience, SSPAI-VCL + GPUPBICGSTAB,

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 GPUP- BICGSTAB is much less than that of SPAI-Adaptive and GPUPBICGSTAB and that of GSPAI-Adaptive and GPUPBICGSTAB for all test cases. [Fig.](#_bookmark19) [11](#_bookmark19) 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

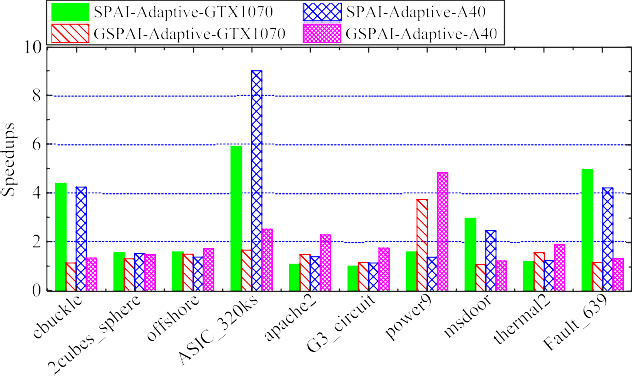
**Table 7**

Comparison of three algorithms with (*𝐸* + |*𝐴*|)2 .

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Matrix | GTX1070 |  |  |  | TITANXp |  | |
|  | SPAI-A | GSPAI-A | GSPAI-Opt |  | SPAI-A | GSPAI-A | GSPAI-Opt |
|  | 29.119 | 9.213 | 7.159 |  | 12.773 | 4.361 | 3.071 |
| cbuckle | 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** |
|  | 50.005 | 25.548 | 25.114 |  | 16.383 | 10.040 | 8.461 |
| 2cubes\_sphere | 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** |
|  | 133.587 | 73.907 | 69.016 |  | 44.813 | 30.238 | 22.370 |
| offshore | 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** |
|  | 10.223 | 2.460 | 1.699 |  | 5.667 | 1.185 | 1.040 |
|  | 0.346 | 0.338 | 0.341 |  | 0.287 | 0.282 | 0.291 |
| ASIC\_320ks | 6 | 6 | 6 |  | 6 | 6 | 6 |
|  | 10.569 | 2.798 | **2.040** |  | 5.954 | 1.467 | **1.331** |
|  | 3.934 | 3.627 | 3.249 |  | 1.500 | 1.391 | 1.314 |
| apache2 | 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** |
|  | 1.864 | 2.094 | 1.467 |  | 0.709 | 0.946 | 0.652 |
|  | 2.291 | 2.283 | 2.292 |  | 0.985 | 0.946 | 0.982 |
| G3\_circuit | 299 | 299 | 299 |  | 345 | 345 | 345 |
|  | 4.155 | 4.377 | **3.759** |  | 1.714 | 1.892 | **1.634** |
|  | 20.575 | 26.960 | 13.497 |  | 10.866 | 17.445 | 6.981 |
| power9 | 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** |
|  | 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** |
|  | 4.082 | 3.753 | 2.991 |  | 1.526 | 1.431 | 1.260 |
| thermal2 | 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** |
|  | 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** |

msdoor

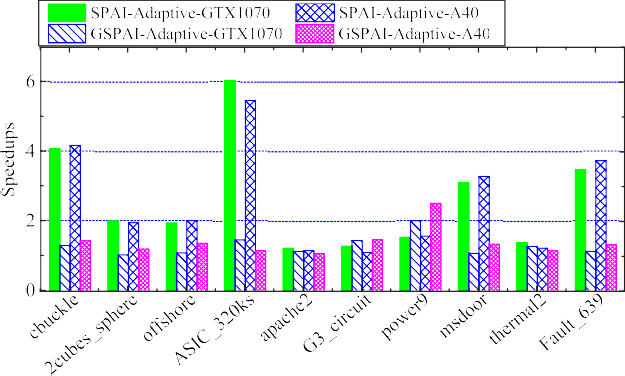
Fault\_639



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

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.



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

For the sparsity pattern of (*𝐸* + |*𝐴*|)2, from [Table](#_bookmark18) [7](#_bookmark18), 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.](#_bookmark20) [12](#_bookmark20). 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.

# 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 finan- cial 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/](https://developer.nvidia.com/content/cuda-10)

[cuda-10](https://developer.nvidia.com/content/cuda-10).

1. [X. Chu, J. Gao, B. Sheng, Efficient concurrent L1-minimization solvers on GPUs,](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb2) [Comput. Syst. Sci. Eng. 38 (3) (2021) 305–320.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb2)
2. [J. Gao, Y. Xia, R. Yin, G. He, Adaptive diagonal sparse matrixvector](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb3) [multiplication on GPU, J. Parallel Distrib. Comput. 157 (2021) 287–302.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb3)
3. [K. Li, W. Yang, K. Li, A hybrid parallel solving algorithm on GPU for quasi-](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb4) [tridiagonal system of linear equations, IEEE Trans. Parallel Distrib. 27 (10)](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb4) [(2016) 2795–2808.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb4)
4. [S.C. Rennich, D. Stosic, T.A. Davis, Accelerating sparse cholesky factorization on](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb5) [GPUs, Parallel Comput. 59 (2016) 140–150.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb5)
5. [H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, M. Kohler,](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb6) [Preconditioned Krylov solvers on GPUs, Parallel Comput. 68 (2017)](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb6) [32–44.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb6)
6. [E. Chow, A. Patel, Fine-grained parallel incomplete LU factorization, SIAM J.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb7) [Sci. Comput. 37 (2) (2015) C169–C193.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb7)
7. [J. Gao, X. Chu, X. Wu, J. Wang, G. He, Parallel dynamic sparse approximate](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb8) [inverse preconditioning algorithm on GPU, IEEE Trans. Parallel Distrib. 33 (12)](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb8) [(2022) 4723–4737.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb8)
8. [Y. Saad, M.H. Schultz, GMRES: a generalized minimal residual algorithm for](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb9) [solving nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 7 (3) (1986)](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb9) [856–869.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb9)
9. [H.A. van der Vorst, Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb10) [for the solution of non-symmetirc linear systems, SIAM J. Sci. Stat. Comput. 12](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb10)

[(3) (1992) 631–644.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb10)

1. [E. Chow, A priori sparsity patterns for parallel sparse approximate inverse](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb11) [preconditioners, SIAM J. Sci. Comput. 21 (5) (2000) 1804–1822.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb11)
2. [J. Gao, K. Wu, Y. Wang, P. Qi, G. He, GPU-accelerated preconditioned GMRES](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb12) [method for two-dimensional Maxwell’s equations, Int. J. Comput. Math. 94 (10)](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb12) [(2017) 2122–2144.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb12)
3. [K. Rupp, R. Tillet, F. Rudolf, et al., ViennaCL-linear algebra library for multi-](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb13) [and many-core architectures, SIAM J. Sci. Comput. 38 (5) (2016) S412–S439.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb13)
4. [M.M. Dehnavi, D.M. Fernandez, J.L. Gaudiot, Parallel sparse approximate inverse](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb14) [preconditioning on graphic processing units, IEEE Trans. Parallel Distrib. 24 (9)](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb14) [(2013) 1852–1861.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb14)
5. 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. [T.A. Davis, Y. Hu, The university of florida sparse matrix collection, ACM Trans.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb17) [Math. Software 38 (1) (2011) 1–25.](http://refhub.elsevier.com/S2772-4859(23)00004-2/sb17)
3. CUDA C Programming guide 11.1, 2021, [http://docs.nvidia.com/cuda/cuda-c-](http://docs.nvidia.com/cuda/cuda-c-programming-guide) [programming-guide](http://docs.nvidia.com/cuda/cuda-c-programming-guide).