Hindawi
Mathematical Problems in Engineering
Volume 2021, Article ID 6804723, 17 pages
https://doi.org/10.1155/2021/6804723

*Research Article*

# Developing a Multi-GPU-Enabled Preconditioned GMRES with Inexact Triangular Solves for Block Sparse Matrices

**Wenpeng Ma**,[1] **Yiwen Hu**,[1] **Wu Yuan**,[2] **and Xiazhen Liu**[2]

[1]*College of Computer and Information Technology, Xinyang Normal University, Xinyang, Henan 464000, China*
[2]*Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China*

Correspondence should be addressed to Wenpeng Ma; mawp@xynu.edu.cn

Received 22 August 2020; Revised 1 November 2020; Accepted 19 January 2021; Published 28 February 2021

Academic Editor: Hua Fan



Solving triangular systems is the building block for preconditioned GMRES algorithm. Inexact preconditioning becomes attractive because of the feature of high parallelism on accelerators. In this paper, we propose and implement an iterative, inexact block triangular solve on multi-GPUs based on PETSc's framework. In addition, by developing a distributed block sparse matrix-vector multiplication procedure and investigating the optimized vector operations, we form the multi-GPU-enabled preconditioned GMRES with the block Jacobi preconditioner. In the implementation, the GPU-Direct technique is employed to avoid host-device memory copies. The preconditioning step used by PETSc's structure and the cuSPARSE library are also investigated for performance comparisons. The experiments show that the developed GMRES with inexact preconditioning on 8 GPUs can achieve up to 4.4x speedup over the CPU-only implementation with exact preconditioning using 8 MPI processes.

## 1. Introduction

Solving a large sparse linear system of equations is always necessary in scientific applications. For unsymmetric matrices, the Krylov subspace-based Generalized Minimal Residual (GMRES) algorithm [1] is widely used as an effective linear solver. Since the general purpose computing on GPUs was introduced in 2007 by NVIDIA, various GPU-enabled GMRES algorithms were designed using NVIDIA's CUDA (Compute Unified Device Architecture). Li and Saad [2] investigated preconditioned Conjugate Gradient (CG) and GMRES methods on TESLA C1060, and 4.0x speedup can be obtained for GMRES algorithm. Khodja et al. [3] implemented the GMRES algorithm on a GPU cluster by exploiting Message Passing Interface (MPI) and CUDA, and they also focused on minimizing the communication between processes using the compressed storage and hypergraph partitioning techniques. Yamazaki et al. [4, 5] developed various procedures to perform the orthogonalization steps of GMRES efficiently and introduced a serial of preconditioners within domain decomposition methods for communication avoiding GMRES (CA-GMRES) on a hybrid CPU + GPU cluster. Their CPU + GPU implementation could obtain a speedup of 7.4x and 1.7x over CA-GMRES without preconditioning and with preconditioning, respectively. Yang [6, 7] developed the preconditioned GMRES algorithm by parallelizing the ILU(0), ILUT, block ILU($k$), and triangular solves on GPUs. Gao et al. [8] proposed an efficient GPU kernel on the sparse matrix-vector multiplication (SpMV) in GMRES and applied the optimized GMRES to solving the two-dimensional Maxwell's equations. He et al. [9] presented an efficient GPU implementation of the GMRES with ILU preconditioners for solving large linear dynamic systems. They obtained 3.0-12.0x speedup over the CPU implementation. Other studies on the Krylov subspace methods on GPUs can be found in [10–19].

Preconditioning techniques are effective for accelerating the convergence of GMRES algorithm. ILU factorization-based preconditioning scheme [20] is one of the most widely used methods. In this method, the ILU factorizations and triangular solves need to be performed. However, they are inherently sequential procedures and have strong data dependency. Many parallel approaches have been studied, for

example, the level-scheduling algorithms [2, 6, 21] and inexact methods [22–27]. The level-scheduling schemes offer limited parallelism on GPU, and the performance highly depends on the sparsity of the matrix. In contrast, inexact methods can offer sufficient parallelism and become attractive on accelerators. Chow et al. [23, 24] presented an iterative, fine-grained strategy for computing ILU factorizations, and their GPU implementation achieved up to 27x over the level-scheduling approach. Anzt et al. [26] used the iterative and inexact idea to solve triangular systems on GPUs, and they showed an advantage over exact triangular solves in terms of total computing time of GMRES although inexact implementation resulted in more GMRES iterations.

Block sparse matrices are very common in scientific computing areas, especially in multiphysics simulations. Most of the existing parallel GMRES and preconditioning algorithms mentioned above were designed for pointwise matrices. Motivated by the block matrix applications and the highly parallelized inexact preconditioning technique, in this paper, we focus on developing a multi-GPU-enabled preconditioned GMRES with the inexact block preconditioning algorithm for block sparse matrices. Our contributions are as follows:

(1) Based on PETSc's framework, we develop the distributed block sparse matrix-vector multiplication for GMRES on multi-GPUs using the cuSPARSE library [28] and the GPU-Direct technique. And two optimization strategies are investigated for vector operations in the GMRES method.

(2) We proposed a strategy that implements the iterative, inexact sparse triangular solves for block matrices on GPUs and integrated it into PETSc. In order to make performance comparisons, we also develop other two versions of the preconditioning technique for block sparse matrices.

(3) By employing the block Jacobi method, we investigate the preconditioned GMRES with various block preconditioning methods on a multi-GPU system. Our experiments show that the developed multi-GPU-based GMRES can achieve a speedup of up to 4.4x using 8 GPUs over the CPU-only implementations using 8 MPI processes.

The remainder of this paper is structured as follows. Section 2 introduces the definition of the block sparse matrix with the storage format and describes a block-based right-preconditioned GMRES algorithm. The optimization strategies, iterative inexact block sparse triangular solves, and three implementations for the preconditioning step are presented and discussed in Section 3. In Section 4, we conduct many experiments to make detailed performance comparisons between different strategies and implementations. Section 5 gives a summary of this work.

## 2. Background

*2.1. Block Sparse Matrix.* Given a sparsity pattern $S_p$, we consider a general $n \times n$ block sparse matrix with block size $s$ as

$$A_{n\times n}^{(r,s)} = \left\{A_{i,j}, i, j = 1, 2, 3, \ldots, r\right\}, \tag{1}$$

where $r$ is the number of block rows and columns, $A_{i,j}$ is a $s \times s$ dense block for $(i, j) \in S_p$, and $A_{i,j} = 0$ for $(i, j) \notin S_p$.

To conduct numerical operations on block sparse matrices, an effective and efficient storage algorithm is required. The well-known method for storing a block sparse matrix is the Block Compressed Sparse Row (BCSR) format [28, 29]. Currently, the popular numerical libraries such as Intel MKL [30], NVIDIA's cuSPARSE [28], and PETSc [31] support BCSR format. In this format, the block sparse matrix $A_{n\times n}^{(r,s)}$ with *nnzb* nonzero blocks is represented by block rows using three arrays *rowptr*, *colval*, and *blkval*. We assume the indexing starts from 0 in C programming language:

(1) *rowptr* is of size $r + 1$, and all the column indices for the $i^{\text{th}}$ $(i < r)$ block row is recorded from $rowptr(i)$ to $rowptr(i + 1)$ (not including) in *colval* array.

(2) *colval* is of size *nnzb*, and stores all column indices by block rows.

(3) *blkval* is of size $nnzb \times s^2$, and stores all the values of nonzero blocks by block rows. The values within a $s \times s$ block are stored consecutively.

However, the BCSR format cloud be specified in row-major or column-major order depending on which index is firstly ordered within a block. The following matrix:

$$\begin{bmatrix} 14 & 15 & 1 & 3 & 0 & 0 \\ 17 & 11 & 5 & 7 & 0 & 0 \\ 0 & 0 & 5 & 20 & 27 & 20 \\ 0 & 0 & 33 & 15 & 10 & 15 \\ 36 & 0 & 32 & 7 & 12 & 16 \\ 4 & 31 & 6 & 13 & 19 & 11 \end{bmatrix}_{6\times 6}^{(3,2)}, \tag{2}$$

*rowptr*: 0 2 4 7, *colval*: 0 1 1 2 0 1 2, *blkval*: {14, 17, 15, 11}, {1, 5, 3, 7}, {5, 33, 20, 15}, {27, 10, 20, 15}, {36, 4, 0, 31}, {32, 6, 7, 13}, {12, 19, 16, 11} illustrates an example of the column-major BCSR format where four values within blocks are stored by columns. In this paper, we employ column-major BCSR format in accordance with PETSc.

*2.2. Preconditioned GMRES Algorithm.* The GMRES method, proposed by Saad and Schultz [1] for solving linear systems, is an iterative process where the residual vector of a linear system is minimized over a Krylov subspace at every

iteration. In this paper, we consider the computation of a block sparse linear system as

$$A_B \vec{x} = \vec{b}, \tag{3}$$

where $B = (n, r, s)$ represents the number of rows and columns, the number of block rows and columns, and block size of the block sparse matrix, respectively.

To accelerate the convergence of GMRES, we apply right-preconditioning technique to the linear system as

$$\begin{aligned} A_B M_B^{-1} \vec{x}_d &= \vec{b}, \\ \vec{x} &= M_B^{-1} \vec{x}_d, \end{aligned} \tag{4}$$

where $M_B$ is the block preconditioning matrix.

Similar to the GMRES for pointwise matrices, it is straightforward to write the preconditioned GMRES algorithm in the block format. The process is shown in Algorithm 1. Compared to the pointwise case, the algorithm performs block sparse vector multiplication (line 7 in Algorithm 1) and block preconditioning steps (line 6 and line 21 in Algorithm 1) instead.

When equation (4) is solved in a whole domain, the preconditioning step can be realized by conducting the incomplete LU (ILU) factorization [6, 20, 23] of $A_B$ and performing two triangular solves [20, 21, 26] as

$$A_B \approx L_B U_B = M_B, \tag{5}$$

$$\begin{aligned} M_B^{-1} \vec{v} = \vec{z} \longrightarrow L_B \vec{f} &= \vec{v}, \\ U_B \vec{z} &= \vec{f}, \end{aligned} \tag{6}$$

where $L_B$ is the lower triangular factor and $U_B$ is the upper triangular factor. However, the ILU factorizations are used as subdomain preconditioners within the domain decomposition methods such as the Additive Schwarz method (ASM) [32] which becomes the block Jacobi method without overlaps. And, the block Jacobi preconditioner is used for the GMRES solver on a distributed system in this work.

## 3. Multi-GPU Implementations

*3.1. Chart of Development.* PETSc [31] offers a wide range of Application Programming Interfaces (APIs) for users to manipulate low-level data structures. Furthermore, it uses function pointers in standard C language instead of an Object-Oriented language such as C++ to achieve data encapsulation and polymorphism [33]. We take advantage of function pointers and structures in C language to develop a user friendly multi-GPU version of the preconditioned GMRES algorithm.

The chart of our code development is shown in Figure 1. Data structure *GMRESInfo* is designed to store all necessary data fields on the GPU. The memory spaces of the fields are allocated when an instance of *GMRESInfo* is created. Before the GMRES solver is executed, the fields in *GMRESInfo* are initialized on GPUs by launching the memory copy operations provided by the basic CUDA library from the host to device. The encapsulated data on the host that needs to be transferred to GPUs can be accessed using the low-level APIs in PETSc. To make full use of the existing numerical toolkits, we employ the efficient functions in cuBLAS and cuSPARSE libraries to perform the local sparse matrix and vector operations in Algorithm 1. On a multi-GPU platform, a portion of the computation inevitably requires data exchange among GPUs. This is implemented and optimized using the CUDA-aware MPI library that supports the GPUDirect technology and transfers data directly among GPU memories through network adapters.

To adopt the multi-GPU version of GMRES solver in an application, a user needs to call three additional functions. Two of them creates and destroys a *GMRESInfo* instance, respectively, while the other (*SetUserDefinedGMRES*) sets the function pointer of GMRES to our developed procedure.

To implement Algorithm 1 on a multi-GPU system, the workload for the host and device should be proper assigned. The matrix-vector operations (line 3 and line 7) and the basic vector operations such as inner product (line 9 and line 12) and the linear combinations (line 10 and line 17) are of high parallelism, so they are fully performed on the GPUs. To conduct convergence monitoring, the conditional statements (line 14–16) are inevitable. And, this part is executed on the host. Because cuSPARSE library allows users to pass a reference on the host when an API is executed on the GPU, the scalar parameter $\text{norm}_w$ is declared on the host to receive the result of $\|\vec{w}\|_2$ and can be directly used in line 14. The size of the Hessenberg matrix is $(m+1) \times m$, where $m$ ($m$ is set to 30 in PETSc by default) is usually much smaller than the matrix size, so the updating of Hessenberg matrix (line 13) and solving of the least square problem (line 20) are performed on the host.

To distributedly store the data on a multi-GPU system, block matrices and vectors are partitioned by rows. On a GPU cluster with $N$ GPUs, the $i^{\text{th}}$ GPU stores $r_i$ block rows of $A_B$ which is represented in BCSR format. In addition, there are several vectors that need to be allocated on the $i^{\text{th}}$ GPU. $\vec{r}_0^{\,i}$, $\vec{z}^{\,i}$, $\vec{x}^{\,i}$, and $\vec{w}^{\,i}$ are allocated with the size of $r_i \times s$, and $V_t$ stores up to $m$ vectors each of which is of size $r_i \times s$. In the preconditioning step (line 6), the preconditioning matrix on the $i^{\text{th}}$ GPU, denoted as $M_B^{(i)}$, is stored in separate factors, i.e., $L_B^{(i)}$ and $U_B^{(i)}$, and the size of memory requirements for ILU(0) level equals to the size of $A_B^{(i)}$, where $A_B^{(i)}$ is the main diagonal part of the partitioned matrix on the $i^{\text{th}}$ GPU.

*3.2. Implementation and Optimization.* The major computation in Algorithm 1 consists of block sparse matrix-vector multiplication (BSpMV), the preconditioning step, and the vector operations. For a block sparse matrix on a distributed system, PETSc partitions the matrix by consecutive block rows and stores each partition in two separate structures. One structure is for the main diagonal of the partitioned matrix, the other is for the off diagonals. Figure 2 shows the strategy of partitioning and storing a $18\,\text{s} \times 18\,\text{s}$ ($s$ is the block size) block sparse matrix obtained by the 5-point finite

**Input**: (1) a block sparse matrix, $A_B$; (2) the preconditioning matrix, $M_B$; (3) the right-hand side, $\vec{b}$; (4) the initial guess, $\vec{x}_0$; (5) the relative tolerance, $\epsilon$; (6) the maximum number of iterations, *maxit*; (7) the restart number, $m$.
**Output**: the solution vector $\vec{x}$.

(1) $continue = 1$; $it = 0$; allocate $H_m = \{H_{p,q}, 0 \le p \le m, 0 \le q < m\}$;
(2) **while** *continue* **do**
(3) $\vec{r}_0 = \vec{b} - A_B\vec{x}_0$; $\beta = \|\vec{r}_0\|_2$; $\vec{v}_0 = (\vec{r}_0/\beta)$;
(4) **if** $(it == 0)$ **then** $\text{norm}_0 = \beta$;
(5) **for** $i = 0$ **to** $(m - 1)$ **do**
(6) $\vec{z} = M_B^{-1}\vec{v}_i$;
(7) $\vec{w} = A_B\vec{z}$;
(8) **for** $k = 0$ **to** $i$ **do**
(9) $H_{k,i} = (\vec{w}, \vec{v}_k)$;
(10) $\vec{w} = \vec{w} - H_{k,i}\vec{v}_k$;
(11) **end**
(12) $\text{norm}_w = \|\vec{w}\|_2$; $H_{i+1,i} = \text{norm}_w$; $t = i$;
(13) updateHessenberg$(H, \text{norm})$;
(14) **if** $\text{norm}_w == 0$ **or** $\log_{10}(\text{norm}/\text{norm}_0) < \log_{10}\varepsilon$ or $++\text{it} == \text{maxit}$ **then**
(15) continue = 0; **break**;
(16) **end**
(17) $\vec{v}_{i+1} = (\vec{w}/\text{norm}_w)$;
(18) **end**
(19) $t = t + 1$;
(20) solve $\vec{y}$ from $\underset{\vec{y}}{\text{argmin}}\|\beta\vec{e}_1 - H_t\vec{y}\|_2$, where $H_t = \{H_{p,q}, 0 \le p \le t, 0 \le q < t\}$
(21) $\vec{x}_0 = \vec{x}_0 + M_B^{-1}V_t\vec{y}$, where $V_t = [\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{t-1}]$;
(22) **end**
(23) $x = x_0$

Algorithm 1: GMRES algorithm with right preconditioning for block sparse matrices.

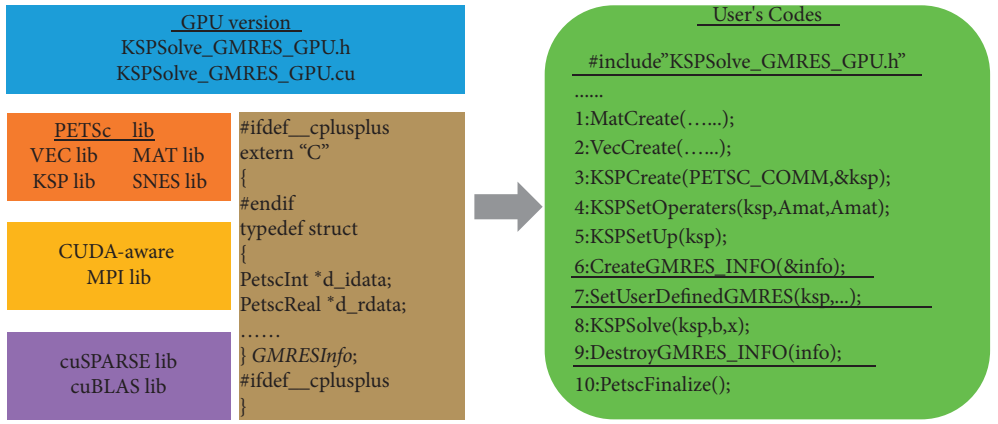


Figure 1: The development chart of the multi-GPU version of GMRES algorithm based on PETSc.

differencing scheme in PETSc. Each partition is responsible for storing 6 block rows and assigned to a MPI process. Take the second partition (dashed box in red) as an example, PETSc stores the 6 s × 6 s square block sparse matrix in the main diagonal as the BCSR format by renumbering the block rows and columns starting from zero. The two off-diagonal parts are merged and renumbered as a local 6 s × 12 s block sparse matrix in the BCSR format. The main advantage of this strategy is that it separates the main diagonal part (MDP, communication-free) from the off-diagonal part (ODP, communication-required), which can be used for overlapping the computation and communication. Therefore, each partition needs three steps to perform the multiplication between a block sparse matrix and a vector. The MDP firstly multiplies the local part of the vector and stores the result temporarily. All partitions then launch MPI sending and receiving calls to exchange data of the vector globally. Lastly, the ODP in each partition multiplies the received vector and accumulates the results to the former temporary vector.

On a multi-GPU system, we consider using a MPI process to control a GPU card. In each MPI process, the arrays expressing the MDP and ODP in the BCSR format for the partitioned matrix are copied to the corresponding GPU memories whose data pointers are encapsulated in a *GMRESInfo* instance. This makes the MDP and ODP capable

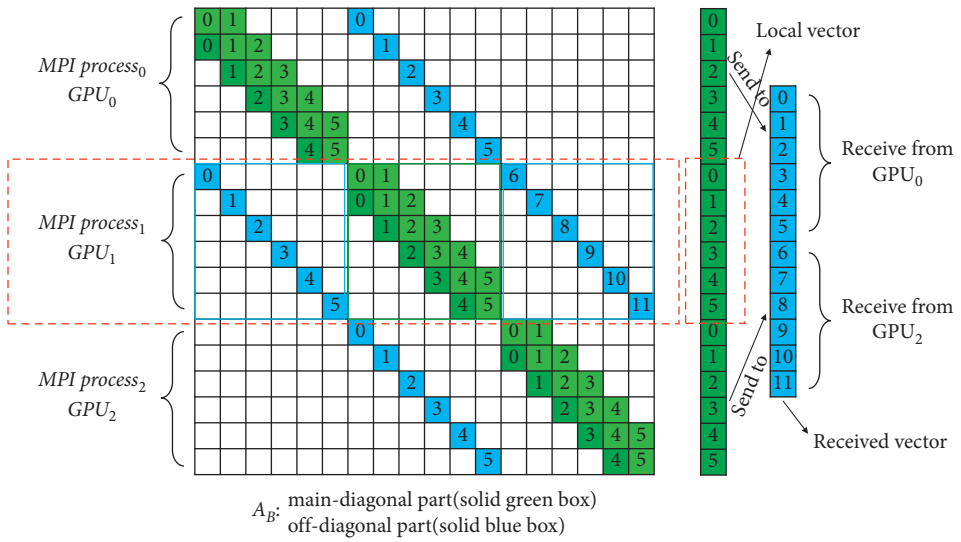


Figure 2: Strategy of partitioning, storing a block sparse matrix, and performing BSpMV on a multi-GPU system based on PETSc.

of conducting the local BSpMV on GPUs by using cusparseDbsrmv from the cuSPARSE library [28]. However, the multiplication between the ODP and the received vector cannot be proceeded until the MPI communication has completed. In our implementation, we perform the MPI communication from GPUs to GPUs directly using the GPU-Direct technique. Note that although in the latest version of PETSc, the GPU-Direct communication is supported in the VecScatter object, the VecScatterBegin, and VecScatterEnd functions that manage the data exchange between vectors (with either ViennaCL or CUDA type in PETSc) cannot be applied directly because the GPU vectors in our implementation are designed in the *GMRESInfo* structure. Efforts are made to extract the low-level communication map from the VectorScatter object that is used for distributed BSpMV on CPUs, and additional kernels are developed. The BSpMV with GPU-Direct communication is summarized in Algorithm 2.

According to the indices in the VecScatter object that record the partial data required to be sent to remote GPUs, the GPU kernel SetGPUSendbuffer is developed to move the data from the local vector to the GPU buffer. cusparseDbsrmv is called asynchronously to perform the MDP BSpMV in a separate CUDA stream, which makes the computation overlap with the following MPI communication. Then, the CUDA-aware MPI can automatically identify all GPU buffers that are requested to send and receive and transfer data between GPUs directly. Once the data is received successfully in each GPU's memory, the BSpMV can be conducted between the ODP and the received buffer on GPUs.

The vector operations in Algorithm 1 including vector inner products (lines 3 and 12), scaling (lines 3 and 17), and linear operations (line 21) are implemented using efficient subroutines from the cuBLAS [34] library. When these functions are called, scalar values are passed by reference on the host instead of the device to receive the computing results from GPUs because some of the results are used by the calculations on CPUs. For example, lines 13–16 and 20 are assigned to CPUs due to very light workload, and they need norm$_w$ and $H$ for computing and logic judgements on CPUs. Therefore, passing host reference in the call of cuBLAS functions avoids manually copying data back to the host memories.

The vector operations in lines 9-10 are time consuming because they are in a nested loop. For each $i\,(0 \le i < m)$, they require $i + 1$ times of MPIAllreduce communication in total because each computation of $H_{k,i}$ requires one MPIAllreduce call and $H_{k,i}$ has to be used to update $\vec{w}$ immediately. In fact, since $\vec{v}_k\,(0 \le k \le i)$ are orthogonal with each other, PETSc splits the single loop into two separate ones each of which is applied to one line of lines 9-10. Therefore, the communication in obtaining all $H_{k,i}$ can be reduced by launching only one call of MPIAllreduce. Furthermore, PETSc optimized the loop of computing $(\vec{w}, \vec{v}_k)\,(0 \le k \le i)$ by using batch computing. For example, the method conducts 8 inner products from $(\vec{w}, \vec{v}_0)$ to $(\vec{w}, \vec{v}_7)$ by launching two CUDA kernels each of which performs 4 inner products in a batch. In each batch, $\vec{w}$ is fetched from global memory only once and reused for all four inner products. This technique reduces the visiting of global memory for $\vec{w}$ repeatedly. Another method where the inner product is transformed to the multiplication between a dense matrix (constructed by all $\vec{v}_k$) and a vector $(\vec{w})$ was studied in [4]. However, this method is not covered in this paper since all of our implementations and optimizations are based on PETSc's framework.

The classical inner product on GPUs [35] employed by PETSc requires the host to copy the reductions of thread blocks from the GPU and conduct a final reduction over all thread blocks on the CPU. In the case of batch computing, the memory copy from the GPU to the host and the final reduction on the host for one batch can be overlapped with the GPU computing for another batch. Figure 3(b) illustrates the idea of asynchronization on different batches, whereas Figure 3(a) shows the synchronized case. To implement this

(1) SetGPUSendbuffer(...); ⟵ from the local vector to the $i^{\text{th}}$ GPU sending buffer.
(2) cusparseDbsrmv(...); ⟵ MDP BSpMV on the $i^{\text{th}}$ GPU in a CUDA stream.
(3) SpMVGPUDirectComm(...); ⟵ exchange data between GPUs and wait until completed.
(4) cusparseDbsrmv(...); ⟵ ODP BSpMV on the $i^{\text{th}}$ GPU.

Algorithm 2: BSpMV on multi-GPUs with GPU-Direct communication.

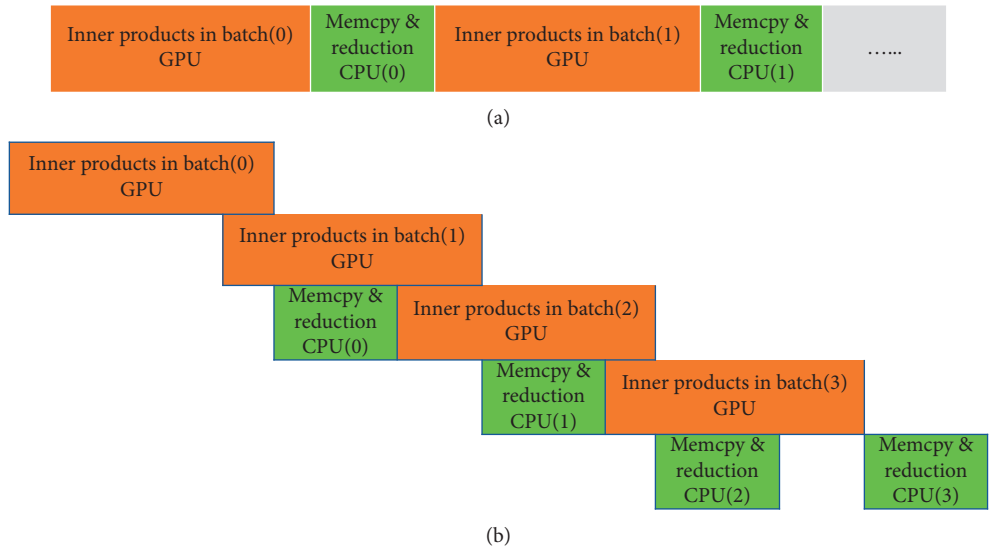


Figure 3: Asynchronous batch computing for inner products on GPUs.

idea, batches are assigned to two CUDA streams alternately so that the overhead of memory copies and reductions on the CPU can be reduced.

The method of batch computing can also be applied to the updating of $\vec{w}$ in the loop. In PETSc's implementation, it calls cublasDaxpy one by one to update $\vec{w}$ for the CUDA vector type or updates $\vec{w}$ with a batch size of 2 for the ViennaCL [36] vector type. We develop several CUDA kernels to use a large batch size instead of calling cublasDaxpy in our implementation. The template of the kernel is shown in Algorithm 3. VecAXPYKernel$_j$ is developed to compute a linear combination of $x_j$ using coefficients $a_j$. Compared to visiting $\vec{w}$ once using cublasDaxpy at a time, calling VecAXPYKernel$_j$ in a batch reduces the number of times of fetching $\vec{w}$ from the global memory. For a given number of loops, the kernels need to be called in a sequential order since $\vec{w}$ that is updated in one batch has to be used in the following batch.

*3.3. Preconditioning.* To implement equations (5) and (6) on a parallel system with distributed GPUs, we consider the block Jacobi preconditioner as

$$A_B^{(i)} \approx L_B^{(i)} U_B^{(i)} = M_B^{(i)}, \quad i = 0, 1, 2, \ldots, l-1, \tag{7}$$

$$\begin{aligned} \left(M_B^{(i)}\right)^{-1} \vec{v}^{(i)} = \vec{z}^{(i)} &\longrightarrow L_B^{(i)} \vec{f}^{(i)} = \vec{v}^{(i)}, \\ U_B^{(i)} \vec{z}^{(i)} &= \vec{f}^{(i)}, \quad i = 0, 1, 2, \ldots, l-1, \end{aligned} \tag{8}$$

where $A_B^{(i)}$ and $M_B^{(i)}$ represents the main diagonal part of the partitioned matrix and the preconditioning matrix in the $i^{\text{th}}$ process, respectively, $\vec{z}^{(i)}$, $\vec{f}^{(i)}$, and $\vec{v}^{(i)}$ are the partitioned vectors in the $i^{\text{th}}$ process, and $l$ is the total number of processes.

For pointwise matrices, the level-scheduling-based parallel methods [2, 21] for the preconditioning step depend heavily on the matrix pattern. Therefore, some cases failed to gain speedups on GPUs over the CPU implementation. To keep a flexible choice of computing the preconditioning step, we investigate three implementations of computing equation (8) for block sparse matrix based on PETSc in this section and make performance comparisons in Section 5.

PETSc implements several efficient subroutines to conduct the ILU factorizations and preconditioning operations for block sparse matrices with various block sizes. In the first implementation, we consider performing (8) using PETSc's API on CPUs and copying the preconditioned vector to the device memory at every iteration of Algorithm 1. The developed function is listed in Algorithm 4. The function accepts four input variables as *ksp*, *vecv*, *vecz*, and *dv*, where *ksp* is an instance of the PETSc's KSP structure, *vecv* and *vecz*, encapsulated by the PETSc's Vec structure in the host memory, are the left- and right-side vectors in equation (8), and *dv* is the device pointer to the left side vector on the GPU. The function outputs the pointer to the right-side vector. To obtain an explicit pointer to the low-level array data of the Vec object in PETSc, we employ two

```
(1) int tid = threadIdx.x + blockDim.x × blockI dx .x;
(2) while tid < len do
(3)     w[tid]+ = a_1 x_1 [tid] + ··· + a_i x_j [tid];
(4)     tid+ = gridDim.x × blockDim.x;
(5) end
```

Algorithm 3: __global__ void VecAXPYKernel$_j$ ($w$, len, $a_1$, ..., $a_j$ , $x_1$, $x_2$, ..., $x_j$).

```
(1) VecGetArray(vecv, & hv);
(2) cudaMemcpy(hv, dv, sizeof(vecv), cudaMemcpyDeviceToHost);
(3) VecRestoreArray(vecv, & hv);
(4) PCApply(ksp ⟶ pc, vecv, vecz);
(5) VecGetArray(vecz, & hz);
(6) cudaMemcpy(dz, hz, sizeof(vecz), cudaMemcpyHostToDevice);
(7) VecRestoreArray(vecz, & hz);
```

Algorithm 4: PETScPrecond(in: *ksp*, *vecv*, *vecz*, *dv*, out: *dz*).

APIs, VecGetArray and VecRetoreArray, before and after using the object. Then, the preconditioning procedure can be divided into three steps. The data transfer is firstly launched to copy the left-side vector from the device to host using cudaMempcy from the CUDA library. This is followed by a call of PCApply in PETSc to apply the declared preconditioner to *vecv*. Inside PCApply, it performs two block sparse triangular solves in which the calculations are conducted in blockwise. Figure 4 shows the pattern of solving a lower block sparse triangular system with block size $s$ in PETSc. The preconditioned vector, *vecz*, is then copied to the device space which *dz* points to.

In the second implementation, we utilize the efficient triangular solvers in the BCSR format provided by the cuSPARSE library. Algorithm 5 lists the main steps of the function (cuSPARSEPrecond) which we develop and integrate into PETSc. The first step (lines 2–7), also called the preprocessing phase, is to estimate the memory requirement, allocate adequate spaces and extract possible parallelism for the subsequent solve phase. The preprocessing step is required to be executed only once on the local GPUs because the lower ($L_B^{(i)}$) and upper ($U_B^{(i)}$) factors remain unchanged during the iterative process of GMRES once the factors has been constructed by the ILU factorization. The second step is to perform two block sparse triangular systems by calling cusparseDbsrsv2_solve twice with the parameters of ($L_B^{(i)}$) and ($U_B^{(i)}$).

Since the ILU factorization of $A_B^{(i)}$ equation (7) is conducted outside Algorithm 1, the structure compatibility between the ILU factorization and triangular solves has to be studied. Specifically, when the block ILU factorization using PETSc on CPUs is preferred, the solving of the upper triangular system has to be transformed into

$$\widetilde{U}_B^{(i)}\overrightarrow{z}^{(i)} = \left(\mathrm{diag}\left(U_B^{(i)}\right)\right)^{-1}\overrightarrow{f}^{(i)}, \tag{9}$$

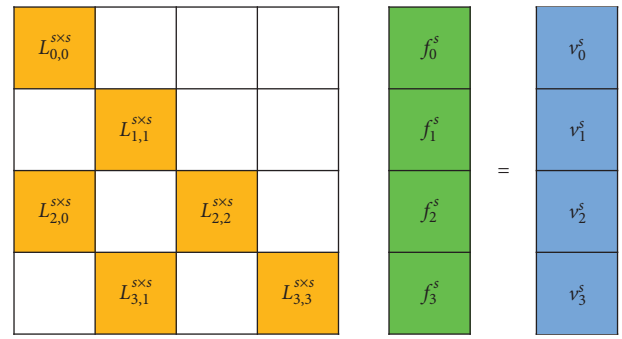


Figure 4: Pattern of the lower block sparse triangular solver in PETSc.

where $\widetilde{U}_B^{(i)} = (\mathrm{diag}(U_B^{(i)}))^{-1}U_B^{(i)}$. This is because cusparseDbsrsv2_solve ignores all the values in the lower part of the input matrix according to the pointwise diagonal. And, $U_B^{(i)}\overrightarrow{z}^{(i)} = \overrightarrow{f}^{(i)}$ is equivalent to the modified system above with all identity matrices in the diagonal blocks. The process can be illustrated by Figure 5(a).

By contrast, the block-based ILU factorization using cuSPARSE can accept $A_B^{(i)}$ in the BCSR format as the input matrix and replace $A_B^{(i)}$ with the lower and upper factors. In this case, it conducts a pointwise ILU factorization by treating $s^2$ values in each block as independent scalars, even though $A_B^{(i)}$ is expressed in the BCSR format. Therefore, the replaced $A_B^{(i)}$ can be directly used as the input matrix for the two calls of cusparseDbsrsv2_solve (lines 9-10 in Algorithm 5) where the pointwise lower and upper factors can be automatically extracted. This strategy is shown in Figure 5(b).

The asynchronous techniques [22, 23, 25–27] where the ILU factorizations and triangular solves are calculated iteratively and inexactly have been attractive. In these

```
(1) if firstcall then
(2)     cusparseDbsrsv2_bufferSize(..., Ldescr, Ablkval, Arowptr, Acolval, s, ..., & Lbufsz);
(3)     cusparseDbsrsv2_bufferSize(..., Udescr, Ablkval, Arowptr, Acolval, s, ..., & U bufsz);
(4)     cudaMalloc(& pLbuf, Lbufsz);
(5)     cudaMalloc(& pUbuf, Ubufsz);
(6)     cusparseDbsrsv2_analysis(..., Ldescr, Ablkval, Arowptr, Acolval, s, ..., pLbuf);
(7)     cusparseDbsrsv2_analysis(..., Udescr, Ablkval, Arowptr, Acolval, s, ..., pUbuf);
(8) end
(9)   cusparseDbsrsv2_solve(..., Ldescr, Ablkval, Arowptr, Acolval, s, ..., dv, df, ..., pLbuf);
(10)  cusparseDbsrsv2_solve(..., Udescr, Ablkval, Arowptr, Acolval, s, ..., df, dz, ..., pUbuf);
```

Algorithm 5: cuSPARSEPrecond(in: *dv*, *df*, *firstcall*, out: *dz*).

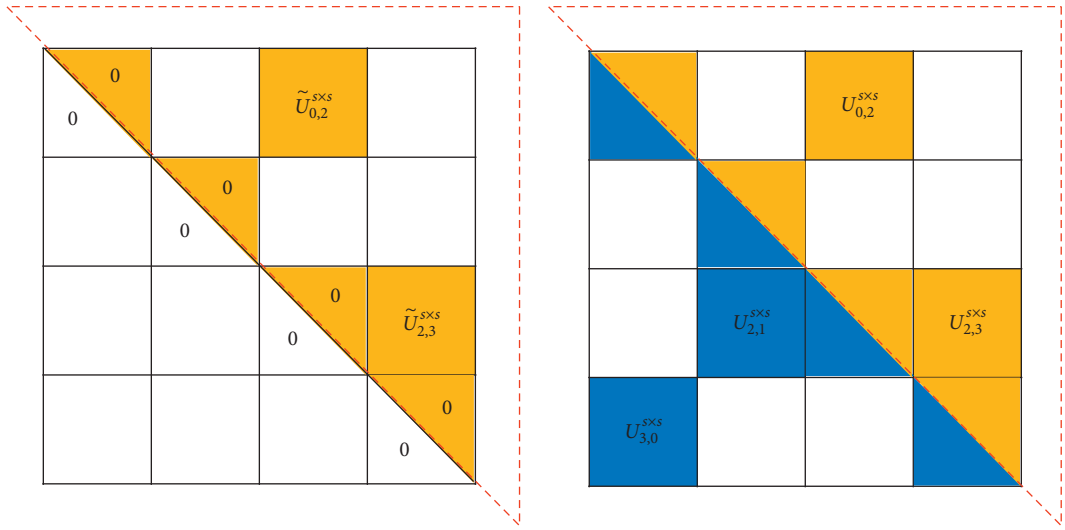


Figure 5: Two different patterns of the upper block sparse triangular solver using cusparseDbsrsv2_solve: (a) $\widetilde{U}_B^{(i)}$; (b) the upper part of (b) $A_B^{(i)}$ after the cuSPARSE's ILU factorization.

methods, although a linear system may need more iterations to converge, the overall performance can be improved because the overhead of the preconditioning step at every step is reduced. In the last implementation, we investigate the inexact triangular solves for block sparse systems on a multi-GPU system.

In the $i^{\text{th}}$ process, by considering the upper block triangular system $U_B^{(i)}\overrightarrow{z}^{(i)} = \overrightarrow{f}^{(i)}$, we explicitly express the block vector $\overrightarrow{z}_j^{(i)}$ in an iterative way as

$$\left[\overrightarrow{z}_j^{(i)}\right]_{it+1} = \left[U_B^{(i)}\right]_{j,j}^{-1}\left(\overrightarrow{f}_j^{(i)} - \sum_{k>j}\left[U_B^{(i)}\right]_{j,k}^{-1}\left[\overrightarrow{z}_k^{(i)}\right]_{it}\right),$$
$$(k, j) \in S_p, j = r-1, r-2, \ldots, 0, \tag{10}$$

where $[\overrightarrow{z}_j^{(i)}]_{\text{it}+1}$ represents the $j^{\text{th}}$ block vector at the $(\text{it}+1)^{\text{th}}$ iteration. And the lower block triangular system can be written in a similar way. By removing the data dependencies in the $r$ constraints, we can update $\overrightarrow{z}_j^{(i)}$ iteratively without the synchronization of data to approximate the exact solutions. Compared to the scalar version for pointwise matrices, equation (10) contains totally different computations. The scalar multiplication is changed to the small dense matrix $(s \times s)$ vector multiplication, and the inverse of a scalar is replaced with the inverse of a $s \times s$ matrix.

It is straightforward to implement the scalar version of equation (10) on GPUs since each thread can be assigned to the calculation of a scalar solution. However, assigning the computation of $\overrightarrow{z}_j^{(i)}$ to a single CUDA thread is not preferred on GPUs because the global memory accesses are far from coalesced when a dense block matrix multiplies a block vector [29]. Inspired by the fine-grained idea for the block sparse matrix-vector multiplication on GPUs [29], we develop a CUDA kernel shown in Algorithm 6 to update $\overrightarrow{z}_j^{(i)}$ in an iteration.

As illustrated in Figure 6, we use 32 threads within a warp to accumulate $[U_B^{(i)}]_{j,k}[\overrightarrow{z}_k^{(i)}]_{\text{it}}$ over all blocks at the $j^{\text{th}}$ row of $U_B^{(i)}$ that satisfy $k > j$ and $(j, k) \in S_p$. Specifically, each thread in the warp computes the multiplication of only one element in $[U_B^{(i)}]_{j,k}$ and the corresponding element in the block vector $[\overrightarrow{z}_k^{(i)}]_{\text{it}}$ (lines 18–20 in Algorithm 6) and stores the scalar result in a register variable. However, 32 threads in

```
(1) tid = blockDim.x × blockIdx.x + threadIdx.x;
(2) __shared__ double sh[warps_per_block][warp_size];
(3) __shared__ double dg[warps_per_block][s^2];
(4) irow_forward = (tid/warp_size);
(5) if irow_forward < r then
(6)    irow = (r − 1) − irow_forward;
(7)    sidx = rowptr[irow]; eidx = rowptr[irow + 1];
(8)    rpos = threadIdx.x % warp_size; warp_id = (threadIdx.x/warp_size);
(9)    cl = (rpos/s)%s; rw = rpos % s;
(10)   range = (warp_size/s^2 × s^2);
(11)   idx = sidx + 1 + (rpos/s^2);
(12)   sum = 0.0; tmp = 0.0; sh[warp_id][rpos] = 0.0;
(13)   if rpos < range then
(14)      if rpos < s^2 then
(15)         dg[warp id][rpos] = blkval[sidx × s^2 + rpos];
(16)      end
(17)      while idx < eidx do
(18)         col = col_val[idx];
(19)         sum = sum − blkval[idx × s^2 + cl × s + rw] × z[col × s + cl];
(20)         idx = idx + ( warp_size/s^2);
(21)      end
(22)      s[warp_id][rpos] = sum;
(23)      for step ≥ 1; step/ = 2 do
(24)         if rpos < step × s and rpos + step × s < warp_size then
(25)            sh[warp_id][rpos]+ = sh[warp_id][rpos + step × s];
(26)         end
(27)      end
(28)      if rpos < s then
(29)         tmp = 0.0; sh[warp_id][rpos]+ = df[irow × s + rpos];
(30)         for i = 0; i < s; i + + do
(31)            tmp+ = dg[warp_id][rpos + i × s] × sh[warp_id][i];
(32)         end
(33)         dz[irow × s + rpos] = tmp;
(34)      end
(35)   end
(36) end
```

Algorithm 6: KernelBtrisvUpper(*urowptr*, *ucolval*, *ublkval*, *df*, *dz*, *r*, *stride*).

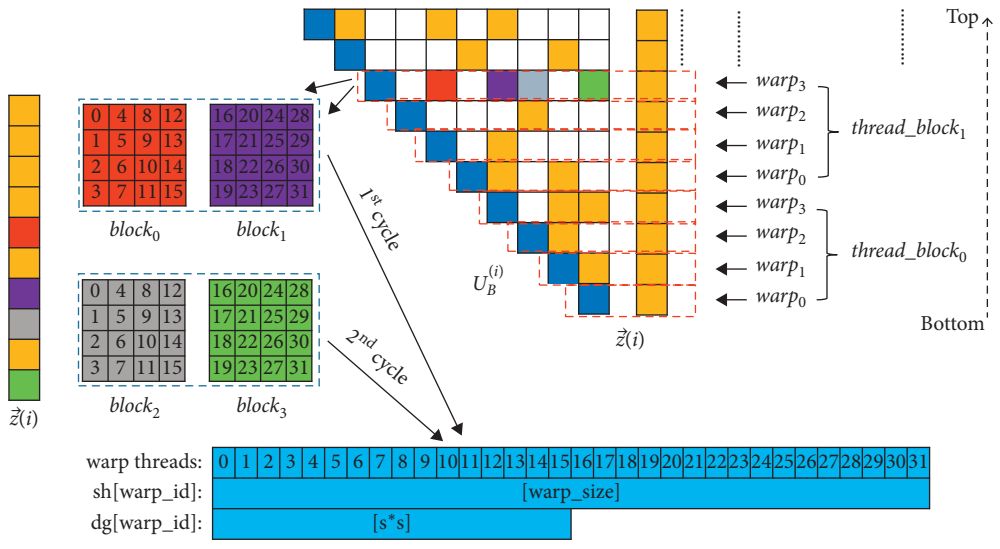


Figure 6: Illustration of the CUDA kernel.

a warp may be unable to cover all blocks simultaneously; then, each thread accumulates the results one by one via performing several cycles on the remaining blocks. To make the thread private variable visible to each other, the threads in each warp share a space which is declared as the type of shared memory (line 2 in Algorithm 6). After that, the scalar result in each thread is copied into the shared memory, and a serial of reduction operations (lines 23–27 in Algorithm 6) are launched to summarize the results for the corresponding rows of the resulting block vector of $\sum_{k>j}[U_B^{(i)}]_{j,k}[\overrightarrow{z}_k^{(i)}]_{\text{it}}$. This is followed by the operation of vector addition on $\overrightarrow{f}^{(i)}$ and $sh$ with the first $s$ elements. The last task of a warp is to conduct a matrix-vector multiplication using the inverse of $[U_B^{(i)}]_{j,j}$ and the resulting vector in the shared memory from the last step. The inverse of $[U_B^{(i)}]_{j,j}$ can be obtained from the block ILU factorization in PETSc because PETSc replaces all diagonal blocks of $U_B^{(i)}$ with their inversions to prepare for the subsequent block triangular solves. We take advantage of this fact by copying the lower and upper factors in PETSc from the host to device before the kernel is executed. And, the first $s^2$ threads in a warp load the diagonal block into the shared memory for the matrix-vector multiplication at line 31 in Algorithm 6.

For the lower block triangular system, the iterative process is written as

$$\left[\overrightarrow{f}_j^{(i)}\right]_{it+1} = \overrightarrow{v}_j^{(i)} - \sum_{k<j}\left[L_B^{(i)}\right]_{j,k}\left[\overrightarrow{f}_k^{(i)}\right]_{it}, \quad (k,j) \in S_p,\ j = 0,1,2,\ldots,r-1, \tag{11}$$

and the kernel to iterate equation (11) for one step is developed in a similar way. Slightly different from the upper one, the kernel maps $r$ rows of $L_B^{(i)}$ from top to bottom. In addition, the matrix-vector multiplication can be discarded because the diagonal blocks of $L_B^{(i)}$ are set to identity matrices in PETSc. We summarize the preconditioning step in Algorithm 7 by performing the two kernels several times.

## 4. Experiments

*4.1. Test Platform.* In our experiments, the tests were conducted on a heterogeneous cluster at Computer Network Information Center, Chinese Academy of Sciences. The cluster consists of 270 blades nodes, 30 GPGPU nodes, and 40 MIC nodes. Each GPGPU node is configured with 2 Intel E5-2680 V2 (Ivy Bridge | 10C | 2.8 GHz) CPUs, 128 GB DDR3 ECC 1866 MHz memory, and two NVIDIA Tesla K20 GPGPU cards. Each card is configured by 320 bit GDDR5 5 GB memory with a clock rate of 2.6 GHz and a bandwidth of 208 GB/sec. In each card, there are 14 stream multiprocessors (SMs) containing 2496 processors in total. The peak single and double precision floating point performance is 3.52 and 1.17 TFLOPS, respectively.

All algorithms introduced in the present paper were developed and tested based on PETSc-3.10.2 that was compiled with "- -with-debugging = 0" option by GCC-4.4.7 and CUDA-aware OpenMPI-3.1.4 [37]. As the optimized PETSc is generally 2 times faster than the debug version of PETSc, we use the optimized PETSc on CPUs as reference when we analyze speedups between CPUs and GPUs. The developed CUDA kernels are compiled by *nvcc* from CUDA 6.5.14. And, all the floating point operations were performed in double precision.

*4.2. Test Matrices.* The test matrices were selected from the SuiteSparse Matrix Collection [38], which are shown in Figure 7. Each pointwise sparse matrix was transformed into the block sparse matrix with the BCSR format using cusparseDcsr2bsr routine from the NVIDIA's cuSPARSE library [28] with a given block size. Table 1 shows the descriptions before and after the transformation. The strategy we use to partition a matrix is

$$\text{nrows}_i = \begin{cases} \left(\frac{r}{l}\right) + 1 & \text{mod}(r,l) \neq 0,\ 0 \leq i \leq \text{mod}(r,l), \\ \frac{r}{l}, & \text{mod}(r,l) \leq i \leq l, \end{cases} \tag{12}$$

where $r$ is the total number of block rows and columns of the matrix, $l$ is the number of MPI processes (GPUs), and $\text{rows}_i$ is the number of block rows of the partitioned matrix assigned to the $i^{\text{th}}$ process (GPU). All processes call MatCreateMPIAIJWithArrays simultaneously to construct a parallel object of the PETSc matrix using their local parts.

*4.3. Results.* We use 4 matrices from *G3_circuit* to *atmosmodl* in Table 1 to investigate the performance of GPUDirect communication in BSpMV and batch computing on vector operations in GMRES restarted with 30.

Figure 8 shows a comparison between the synchronized and asynchronized inner products with two batch sizes. All cases are conducted using 8 MPI processes and GPUs. Each MPI process is associated with a CPU core and controls a GPU card. Each process accumulates the wall clock time of computing the local inner products (line 9 in Algorithm 1) on the GPU until the case has converged. And, the wall clock times are averaged over all processes and shown in the $y$-axis of Figure 8. In both the synchronized and asynchronized cases, results using a batch size of 4 outperform that using a batch size of 2. This is to be expected because the larger batch size can reduce the global memory accesses in the loop. We notice that the asynchronized inner products with batch size 2 perform better than the synchronized ones using batch size 4. This demonstrates that overlapping the frequent memory copies and the computing on the host is effective for a smaller batch size. And, around 2.0x speedup can be obtained using asynchronized inner products with batch size 4 compared to using the synchronized method with batch size 2.

For updating $\overrightarrow{w}$ at line 10 of Algorithm 1, Figure 9 shows the wall clock comparison between the implementation using cublasDaxpy from the cuSPARSE library and that using $\text{VecAXPYKernel}_j\,(j \leq 8)$. Since the operations are memory bound, reducing the number of global access to $\overrightarrow{w}$

```
(1) for it = 0; it < itmax; it++ do
(2)     KernelBtrisvLower <<< nblocksl, nthreadsl >>> (lrowptr, lcolval, lblkval, dv, df, r, stride);
(3) end
(4) for it = 0; it < itmax; it++ do
(5)     KernelBtrisvUpper <<< nblocksu, nthreadsu >>> (urowptr, ucolval, ublkval, df, dz, r, stide);
(6) end
```

Algorithm 7: IterativePrecond(in: *lrowptr*, *lcolval*, *lblkval*, *urowptr*, *ucolval*, *ublkval*, *dv*, *df*, *r*, *stride*, $it_{max}$ out: *dz*).

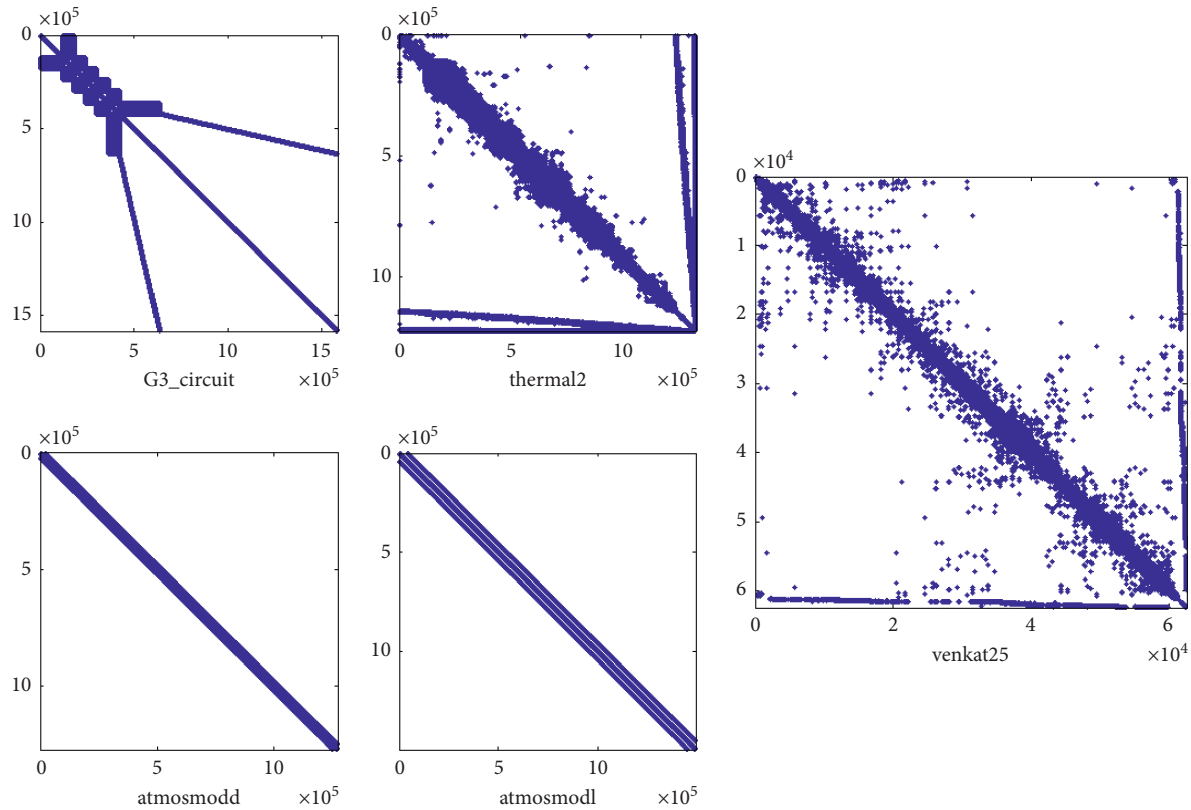


Figure 7: The patterns of the selected matrices.

Table 1: Descriptions of the test matrices.

| Matrix | *n* | *nnz* | *r* | *nnzb* | *s* |
|---|---|---|---|---|---|
| G3_circuit | 1585478 | 7660826 | 792739 | 5230049 | 2 |
| thermal2 | 1228045 | 8580313 | 245609 | 3791545 | 5 |
| atmosmodd | 1270432 | 8814880 | 317608 | 2190844 | 4 |
| atmosmodl | 1489752 | 10319760 | 496584 | 3429888 | 3 |
| venkat25 | 62424 | 1717763 | 15606 | 107362 | 4 |

improves the performance. We see from Figure 9 that over 2.5x speedup can be obtained for all four cases.

Figure 10 shows the performance for the BSpMV on the multi-GPUs. For each matrix, two strategies are studied and compared. One is using the BSpMV on the MDP to overlap with the GPU-Direct communication that is prepared for the BSpMV on the ODP. The other is to perform the BSpMV on the MDP, the communication, and the BSpMV on the ODP in a sequential order. The wall clock time for each configuration is estimated as the maximum value over all GPUs within 30 GMRES iterations (line 7 during the nested loop (lines 5–18) with $m = 30$) because the performance is determined by the slowest one. We observe that the overlapping strategy has shown a significant improvement in wall clock times for *G3_circuit* and *thermal2*. This is because these two matrices have much larger bandwidth that leads to

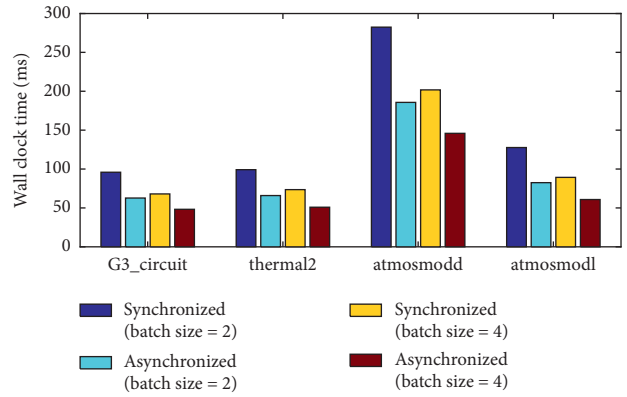


Figure 8: Comparison between the synchronized and asynchronized inner products with different batch sizes.

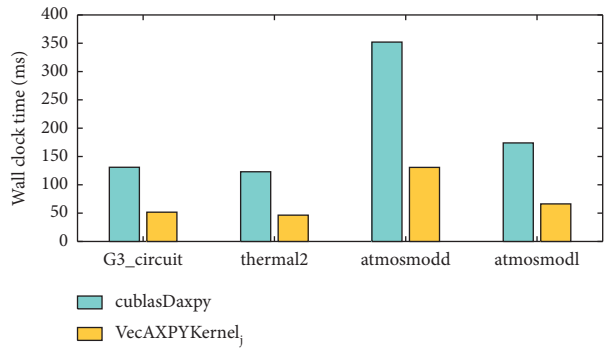


Figure 9: Comparison between cublasDaxpy and VecAXPYKernel$_j$ ($j \leq 8$).

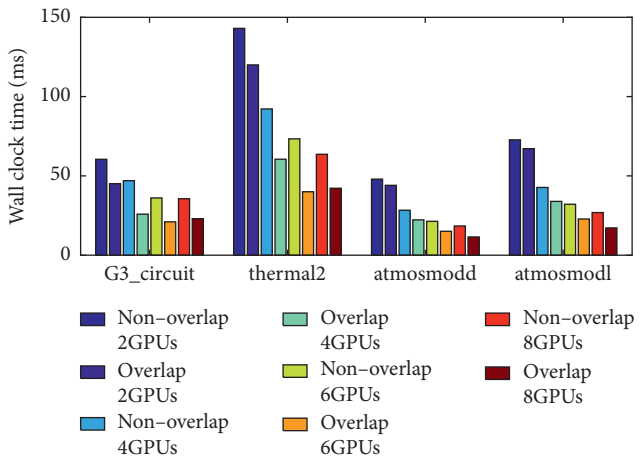


Figure 10: Performance of BSpMV with and without overlapping strategies.

more overhead in MPI communication than the other two matrices. Especially, when the number of GPUs increases to 8, the MPI overhead accounts for the major proportion of the overall time for these two cases and the decreasing computational workload is unable to overlap the overhead effectively. This is why we observe that 8 GPUs costs more time than 6 GPUs in the two cases. When less than 6 GPUs are used for *atmosmodd* and *atmosmodl*, the communication accounts for around 20% of the total GPU time due to the small bandwidth; therefore, we see relatively small gaps between the overlap bars and the nonoverlap bars. However, the proportions of the communication increase to nearly 30% and 40% when 6 GPUs and 8 GPUs are used, which makes the overlapping strategy has an obvious advantage over the nonoverlapping method.

By employing the optimization techniques discussed above, we report the strong scalability for all the four cases in Figure 11. The wall clock times are estimated from lines 3 to 21 (without the preconditioning step) in Algorithm 1 and averaged over three runs. For *G3_circuit* and *thermal2* cases, we observe that only 33.3% and 35.8% parallel efficiencies can be obtained on 16 GPUs compared to 2 GPUs, respectively. This can be explained by the fact that the two matrices have large bandwidths and irregular sparsity patterns which results in unbalanced workload of matrix-vector multiplication and unbalanced MPI communication. Although the overlapping strategy is employed, the decreasing computational workload fails to hide the increasing communication overhead. The parallel efficiency for both *atmosmodd* and *atmosmodl* cases is over 50% on 16GPUs, which benefits from the smaller bandwidth and structured sparsity pattern.

Then, we investigate the performance of preconditioned GMRES with different preconditioning algorithms. Since the existing works of preconditioned GMRES focused either on scalar matrices or on single-GPU implementation, we compare our algorithm to the implementations based on PETSc and cuSPARSE libraries where block matrices are efficiently structured. *Thermal2*, *mosmodd*, *atmosmodl*, and *venkat25* are selected for the experiments, and the results for the first three matrices are shown from Tables 2–4 and the result for *venkat25* is shown in Figure 12. In each table, the first two row sections show the performance results for Algorithms 4 and 5. And, the remaining sections are the results for Algorithm 7 with different choices of the maximum number of iterations ($it_{max}$) used in Algorithm 7. In each section, the first three rows list the minimum, maximum, and average wall clock times spent on performing the preconditioning step in a restarted loop (lines 5–18 in Algorithm 1). The fourth row shows the total wall clock times spent on executing GMRES until the given relative tolerance is satisfied. And, the fifth row shows the total number of iterations required for the convergence tolerance.

In Table 2, the *thermal2* case shows that the preconditioning step using cuSPARSEPrecond is much slower than PETScPrecond. This demonstrates that the matrix pattern of *thermal2* is not suitable for level-scheduling-based algorithms. To estimate the degree of load balance for the preconditioning step, we introduce the load balance factor as

$$\mathrm{LBF} = \frac{T_{\mathrm{precond\_max}}}{T_{\mathrm{precond\_avg}}}. \tag{13}$$

We observe that the LBFs for cuSPARSEPrecond increases from 1.09 to 1.58 as the number of GPUs increases, which is larger than that for PETScPrecond. This can be

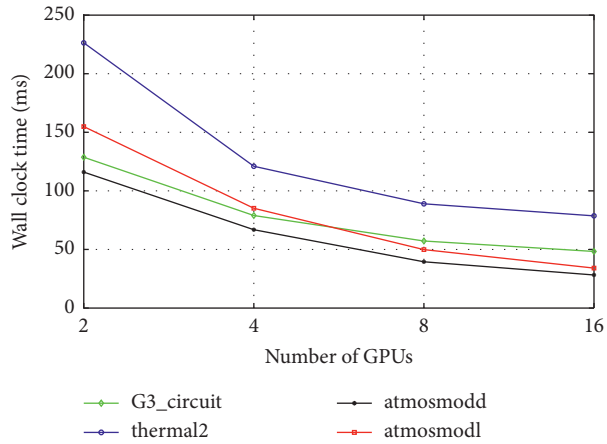


Figure 11: Strong scaling for all the four cases.

Table 2: *thermal2*: comparisons among three implementations on the preconditioning step of GMRES(30) with the relative tolerance, $\epsilon = 10^{-3}$.

| Algorithms | | 2GPUs | 4GPUs | 6GPUs | 8GPUs |
|---|---|---|---|---|---|
| PETScPrecond | $T_{\text{precond\_min}}$ (ms) | 1085 | 523 | 327 | 196 |
| | $T_{\text{precond\_max}}$ (ms) | 1125 | 569 | 408 | 295 |
| | $T_{\text{precon\_avg}}$ (ms) | 1105 | 550 | 358 | 260 |
| | $T_{\text{gmres}}$ (ms) | 2760 | 2031 | 1524 | 1168 |
| | GMRES iterations | 58 | 83 | 86 | 86 |
| cuSPARSEPrecond | $T_{\text{precond\_min}}$ (ms) | 2839 | 1081 | 689 | 490 |
| | $T_{\text{precond\_max}}$ (ms) | 3429 | 1966 | 1431 | 1254 |
| | $T_{\text{precon\_avg}}$ (ms) | 3134 | 1569 | 1053 | 789 |
| | $T_{\text{gmres}}$ (ms) | 7378 | 6036 | 4564 | 4012 |
| | GMRES iterations | 58 | 83 | 86 | 86 |
| IterativePrecond ($\text{it}_{\text{max}} = 3$) | $T_{\text{precond\_min}}$ (ms) | 356 | 172 | 110 | 72 |
| | $T_{\text{precond\_max}}$ (ms) | 365 | 185 | 128 | 93 |
| | $T_{\text{precon\_avg}}$ (ms) | 361 | 177 | 118 | 87 |
| | $T_{\text{gmres}}$ (ms) | 1385 | 1080 | 800 | 644 |
| | GMRES iterations | 66 | 96 | 99 | 100 |
| IterativePrecond ($\text{it}_{\text{max}} = 4$) | $T_{\text{precond\_min}}$ (ms) | 474 | 229 | 146 | 96 |
| | $T_{\text{precond\_max}}$ (ms) | 485 | 246 | 170 | 124 |
| | $T_{\text{precon\_avg}}$ (ms) | 480 | 236 | 157 | 116 |
| | $T_{\text{gmres}}$ (ms) | 1505 | 1192 | 883 | 705 |
| | GMRES iterations | 60 | 88 | 92 | 94 |
| IterativePrecond ($\text{it}_{\text{max}} = 5$) | $T_{\text{precond\_min}}$ (ms) | 572 | 286 | 188 | 120 |
| | $T_{\text{precond\_max}}$ (ms) | 586 | 308 | 212 | 155 |
| | $T_{\text{precon\_avg}}$ (ms) | 579 | 294 | 196 | 145 |
| | $T_{\text{gmres}}$ (ms) | 1727 | 1329 | 996 | 760 |
| | GMRES iterations | 59 | 86 | 89 | 89 |

explained by the fact that the level-scheduling algorithm extracts much different levels of parallelism from the partitioned matrices $A_B^{(i)}$ with different matrix patterns. The LBFs for IterativePrecond do not exceed 1.1, which is better than that for both PETscPrecond and cuSPARSEPrecond. This is due to the fully parallelized feature of IterativePrecond. In all three implementations, the total number of GMRES iterations used to converge to the given tolerance increases with increasing the number of GPUs. This is to be expected because the number of iterations usually increases when the block Jacobi method is applied to increasing partitions. Compared to the exact triangular solver using PETScPrecond and cuSPARSEPrecond, the iterative, inexact solver using IterativePrecond makes GMRES(30) require more iterations to converge to the same tolerance. And, the number of iterations for GMRES(30) can be reduced by increasing the number of times the iterative process is executed. Despite more GMRES (30) iterations are required, IterativePrecond shows an advantage over the other two methods in terms of wall clock times. When IterativePrecond with $\text{it}_{\text{max}} = 3$ is used for comparisons, we obtain 1.81x and 6.23x on 8 GPUs over the implementations of GMRES using PETScPrecond and cuSPARSEPrecond, respectively.

Table 3: *atmosmodd*: comparisons among three implementations on the preconditioning step of GMRES(30) with the relative tolerance, $\epsilon = 10^{-5}$.

| Algorithms | | 2GPUs | 4GPUs | 6GPUs | 8GPUs |
|---|---|---|---|---|---|
| PETScPrecond | $T_{\text{precond\_min}}$ (ms) | 526 | 262 | 180 | 127 |
| | $T_{\text{precond\_max}}$ (ms) | 580 | 293 | 203 | 142 |
| | $T_{\text{precon\_avg}}$ (ms) | 553 | 282 | 191 | 135 |
| | $T_{\text{gmres}}$ (ms) | 3311 | 1453 | 1085 | 765 |
| | GMRES iterations | 136 | 116 | 124 | 120 |
| cuSPARSEPrecond | $T_{\text{precond\_min}}$ (ms) | 285.3 | 173 | 148.3 | 135.7 |
| | $T_{\text{precond\_max}}$ (ms) | 285.5 | 173.4 | 149 | 137.3 |
| | $T_{\text{precon\_avg}}$ (ms) | 285.4 | 173.2 | 148.5 | 136.7 |
| | $T_{\text{gmres}}$ (ms) | 1886 | 969 | 854 | 741 |
| | GMRES iterations | 136 | 116 | 124 | 120 |
| IterativePrecond ($\text{it}_{\max} = 3$) | $T_{\text{precond\_min}}$ (ms) | 215.2 | 108.3 | 72.4 | 55 |
| | $T_{\text{precond\_max}}$ (ms) | 216.7 | 108.8 | 73 | 55.4 |
| | $T_{\text{precon\_avg}}$ (ms) | 216 | 108.6 | 72.7 | 55.2 |
| | $T_{\text{gmres}}$ (ms) | 1814 | 824 | 661 | 469 |
| | GMRES iterations | 160 | 137 | 157 | 143 |
| IterativePrecond ($\text{it}_{\max} = 4$) | $T_{\text{precond\_min}}$ (ms) | 286 | 143.9 | 146 | 96 |
| | $T_{\text{precond\_max}}$ (ms) | 286.4 | 144.6 | 170 | 124 |
| | $T_{\text{precon\_avg}}$ (ms) | 286.2 | 144.2 | 157 | 116 |
| | $T_{\text{gmres}}$ (ms) | 1900 | 934.3 | 883 | 705 |
| | GMRES iterations | 138 | 128 | 92 | 94 |
| IterativePrecond ($\text{it}_{\max} = 5$) | $T_{\text{precond\_min}}$ (ms) | 356.9 | 179.5 | 120.5 | 91 |
| | $T_{\text{precond\_max}}$ (ms) | 357.4 | 180.2 | 121.2 | 91.4 |
| | $T_{\text{precon\_avg}}$ (ms) | 357.2 | 179.9 | 120.8 | 91.2 |
| | $T_{\text{gmres}}$ (ms) | 2043 | 1056 | 897 | 646 |
| | GMRES iterations | 125 | 124 | 154 | 148 |

Table 4: *atmosmodl*: comparisons among three implementations on the preconditioning step of GMRES (30) with the relative tolerance, $\epsilon = 10^{-5}$.

| Algorithms | | 2GPUs | 4GPUs | 6GPUs | 8GPUs |
|---|---|---|---|---|---|
| PETScPrecond | $T_{\text{precond\_min}}$ (ms) | 539 | 276 | 183 | 132 |
| | $T_{\text{precond\_max}}$ (ms) | 583 | 293 | 199 | 146 |
| | $T_{\text{precon\_avg}}$ (ms) | 561 | 283 | 193 | 141 |
| | $T_{\text{gmres}}$ (ms) | 1628 | 944 | 690 | 559 |
| | GMRES iterations | 63 | 70 | 74 | 79 |
| cuSPARSEPrecond | $T_{\text{precond\_min}}$ (ms) | 408.8 | 230.8 | 191.2 | 175.0 |
| | $T_{\text{precond\_max}}$ (ms) | 409 | 231.4 | 192.5 | 176.2 |
| | $T_{\text{precon\_avg}}$ (ms) | 408.9 | 231.1 | 191.8 | 175.7 |
| | $T_{\text{gmres}}$ (ms) | 1236 | 768 | 656 | 621 |
| | GMRES iterations | 63 | 70 | 74 | 79 |
| IterativePrecond ($\text{it}_{\max} = 2$) | $T_{\text{precond\_min}}$ (ms) | 182.2 | 92.1 | 62.1 | 47.1 |
| | $T_{\text{precond\_max}}$ (ms) | 182.9 | 92.6 | 62.5 | 47.6 |
| | $T_{\text{precon\_avg}}$ (ms) | 182.5 | 92.3 | 62.2 | 47.4 |
| | $T_{\text{gmres}}$ (ms) | 887 | 497 | 382 | 327 |
| | GMRES iterations | 77 | 82 | 87 | 94 |
| IterativePrecond ($\text{it}_{\max} = 3$) | $T_{\text{precond\_min}}$ (ms) | 272.1 | 137.4 | 92.4 | 70.8 |
| | $T_{\text{precond\_max}}$ (ms) | 272.5 | 138.3 | 92.8 | 70.4 |
| | $T_{\text{precon\_avg}}$ (ms) | 272.3 | 137.7 | 92.7 | 70.3 |
| | $T_{\text{gmres}}$ (ms) | 1001 | 580 | 422 | 355 |
| | GMRES iterations | 68 | 75 | 78 | 82 |
| IterativePrecond ($\text{it}_{\max} = 4$) | $T_{\text{precond\_min}}$ (ms) | 361.5 | 182.8 | 123.1 | 93.3 |
| | $T_{\text{precond\_max}}$ (ms) | 362.6 | 183.4 | 123.4 | 93.6 |
| | $T_{\text{precon\_avg}}$ (ms) | 362.1 | 183.1 | 123.3 | 93.4 |
| | $T_{\text{gmres}}$ (ms) | 1172 | 667 | 494 | 402 |
| | GMRES iterations | 65 | 72 | 76 | 80 |

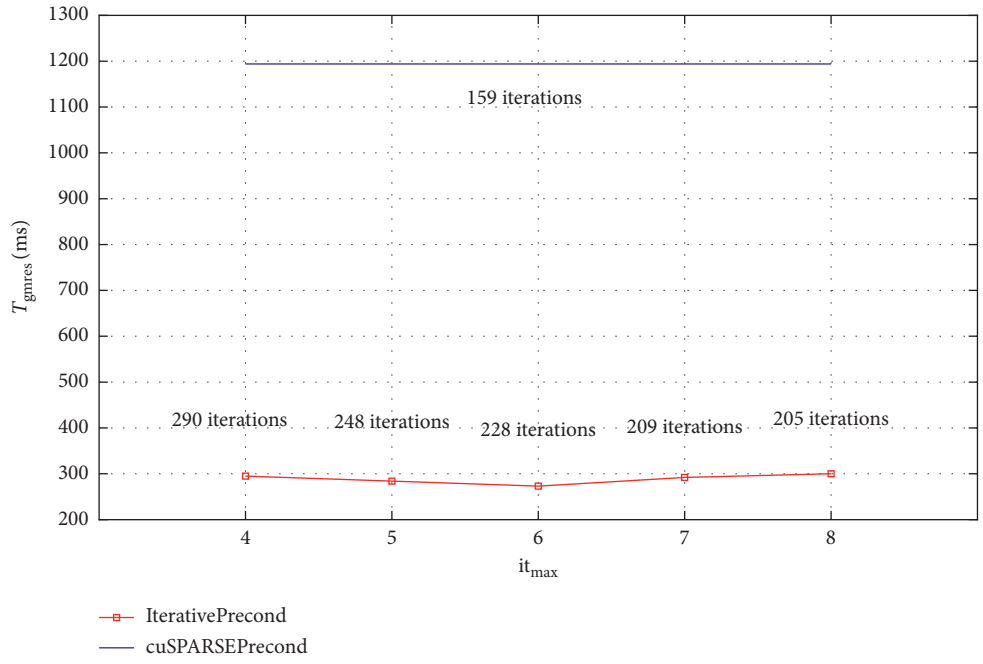


FIGURE 12: venkat25: comparison of $T_{gmres}$ with increasing $it_{max}$ (relative tolerance: $10^{-5}$).

We also compare our CPU + GPU-based GMRES (30) with IterativePrecond to the corresponding CPU only implementation in PETSc, which is shown in Figure 13. For a number of $N$ in the $x$-axis of Figure 13, the baseline bar in blue is obtained by running the PETSc's CPU-only (pure MPI) implementation on $N$ CPU cores each of which is associated with a MPI process, while the other bar is obtained by running the CPU + GPU implementation on $N$ GPUs each of which is controlled by a MPI process. By using 8 GPUs, we obtain over 4.4x speedups compared to 8 CPU cores, which greatly reduces the computing time of GMRES(30) algorithm for the problem with a fixed size.

For the *atmosmodd* case, the right-hand vector comes from the accompanying data of the case. In Table 3, the *atmosmodd* case shows that both cuSPARSEPrecond and IterativePrecond are faster than PETScPrecond. The pointwise matrix pattern of *atmosmodd* has been reported to be suitable for level-scheduling algorithms [2], and we notice in this experiment that the benefit of the pattern is still kept when *atmosmodd* is transformed into the BCSR format. We observe that $T_{gmres}$ using cuSPARSEPrecond is about 1.76x faster than that using PETScPrecond when 2 GPUs are used. However, the speedup reduces as the number of GPUs increases, and $T_{gmres}$ with cuSPARSEPrecond does not show persuasive advantage over $T_{gmres}$ with PETScPrecond when 8 GPUs are used. By contrast, although IterativePrecond results in more GMRES iterations, $T_{gmres}$ using IterativePrecond with $it_{max} = 3$ maintains more than 1.6x speedups compared to $T_{gmres}$ using PETScPrecond when the number of GPUs increases from 2 to 8. In terms of *LBFs* for the preconditioning step, both cuSPARSEPrecond and IterativePrecond performs better than PETScPrecond because of the parallelism and the relatively balanced partitions. As we expected, the number of GMRES iterations is gradually reduced with the increasing

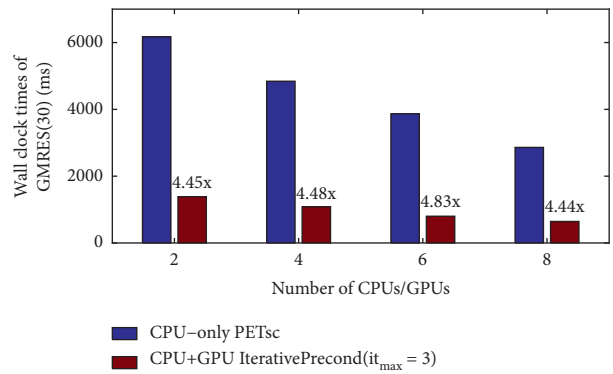


FIGURE 13: *thermal2*: comparisons of the wall clock times of GMRES(30) between the CPU-only implementation using PETSc and out CPU + GPU implementation.

value of $it_{max}$, but leading to more overhead in every 30 preconditioning steps, and thus makes the total wall clock time of GMRES increase. Due to the stochastic feature of updating solutions in IterativePrecond, the GMRES iterations could be less than that using the exact preconditioners. In this case, for example, when IterativePrecond with $it_{max} = 5$ is employed, only 125 iterations are required to make GMRES(30) converge, which is 11 less than that using PETScPrecond. Figure 14 shows the speedups obtained by the CPU + GPU implementation over the PETSc implementation on CPUs. And, above 3.0 x speedup can be obtained for all configurations.

The right-hand vector for the *atmosmodl* case also comes with the matrix. The results are shown in Table 4. We find that IterativePrecond using $it_{max} = 4$ is sufficient for GMRES (30) to converge with nearly the same number of iterations as that obtained by the exact preconditioners. The scalability

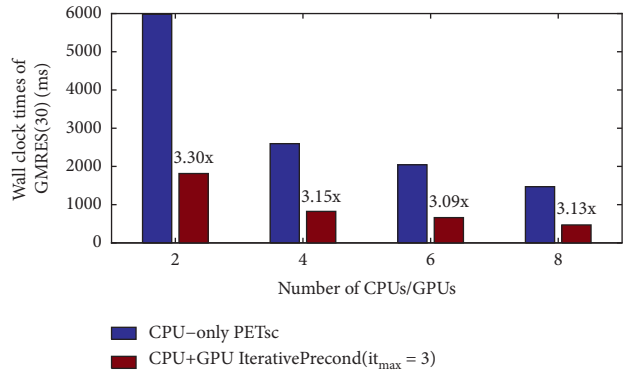


Figure 14: *atmosmodd*: comparisons of the wall clock times of GMRES (30) between the CPU-only implementation using PETSc and out CPU + GPU implementation.

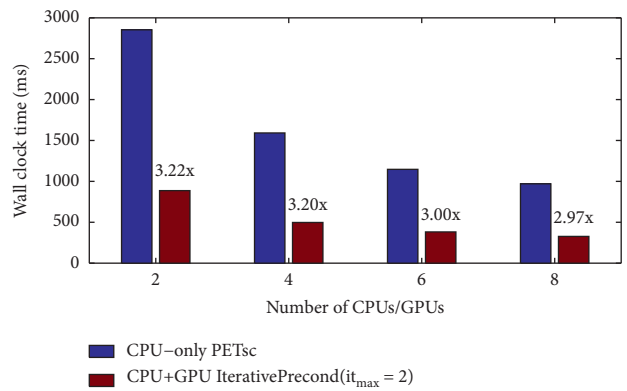


Figure 15: *atmosmodl*: comparisons of the wall clock times of GMRES(30) between the CPU-only implementation using PETSc and out CPU + GPU implementation.

of IterativePrecond is much better than cuSPARSEPrecond because there is no data dependencies when IterativePrecond is applied, and cuSPARSEPrecond is slower than PETScPrecond when 8 GPUs are used. $T_{\text{gmres}}$ using IterativePrecond with $\text{it}_{\text{max}} = 2$ shows a speedup of 1.71x and 1.90x over that using PETScPrecond and cuSPARSEPrecond, respectively. Compared to the CPU-only implementation using 8 MPI processes, 2.97x speedup can be obtained using 8 GPUs, which is shown in Figure 15.

Since the scale of *venkat25* is not large enough to gain speedups on GPUs over the pure MPI implementation, we use this case only to demonstrate the advantage of the inexact triangular solve over the cuSPARSE-based triangular solve. Figure 12 shows a comparison of $T_{\text{gmres}}$ with increasing $\text{it}_{\text{max}}$. We observe a optimal value of 6 for $\text{it}_{\text{max}}$ because $T_{\text{gmres}}$ attains the minimum value at $\text{it}_{\text{max}} = 6$. Although IterativePrecond results in 69 more GMRES iterations to converge, it still outperforms cuSPARSEPrecond. When 4 GPUs are used, GMRES with IterativePrecond performs 4.37x faster than that with cuSPARSEPrecond.

## 5. Conclusion and Future Work

In this work, focusing on the block sparse matrices, we present the development of a PETSc-enabled GMRES algorithm with various optimizations and preconditioning implementations on a GPU cluster. Motivated by the fined-grained feature of the inexact preconditioners in recent years, we propose a GPU algorithm to conduct a blockwise triangular solve inexactly and iteratively. On the block matrices selected from the SuiteSparse matrix collection, our experiments have shown that the preconditioned GMRES with inexact triangular solves outperform that with exact ones implemented by the cuSPARSE library and achieve up to 4.4x speedup using 8 GPUs over the CPU-only implementation using 8 MPI processes. Efforts are undergoing to study the preconditioned GMRES algorithm using the Additive Schwarz Method with the proposed inexact triangular solves on multi-GPUs.

## Data Availability

The matrix data used to support the findings of this study are available from the SuiteSparse Matrix Collection (https://sparse.tamu.edu/).

## Conflicts of Interest

The authors declare no conflicts of interest.

## Acknowledgments

This work was supported by a grant from the National Key R&D Program of China (no. 2019YFB1704202), National Natural Science Foundation of China (no. 61702438), Nanhu Scholar Program of XYNU, and Innovation Team Support Plan of University Science and Technology of Henan Province (19IRTSTHN014).

## References

[1] Y. Saad and M. H. Schultz, "GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, 1986.

[2] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, no. 2, pp. 443–466, 2013.

[3] L. Z. Khodja, R. Couturier, A. Giersch et al., "Parallel sparse linear solver with GMRES method using minimization techniques of communications for GPU clusters," *Journal of Supercomputing*, vol. 69, no. 1, pp. 200–224, 2014.

[4] I. Yamazaki, H. Anzt, S. Tomov et al., "Improving the performance of CA-GMRES on multicores with multiple GPUs," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, IEEE, Phoenix, AZ, USA, May 2014.

[5] I. Yamazaki, S. Rajamanickam, E. G. Boman, M. Hoemmen, M. A. Heroux, and S. Tomov, "Domain decomposition preconditioners for communication-avoiding krylov methods on a hybrid CPU/GPU cluster," in *Proceedings of the SC14: International Conference for High Performance Computing,*

*Networking, Storage and Analysis*, p. 933944, New Orleans, LA, USA, November 2014.

[6] B. Yang and H. Liu, "Accelerating the GMRES solver with block ILU (K) preconditioner on GPUs in reservoir simulation," *Journal of Geology & Geophysics*, vol. 4, no. 2, pp. 1–7, 2015.

[7] B. Yang, H. Liu, Z. Chen et al., "GPU-accelerated preconditioned GMRES solver," in *Proceedings of the IEEE International Conference on High Performance & Smart Computing & Big Data Security on Cloud*, IEEE, New York, NY, USA, April 2016.

[8] J. Gao, K. Wu, Y. Wang et al., "GPU-accelerated preconditioned GMRES method for two-dimensional Maxwell's equations," *International Journal of Computer Mathematics*, vol. 94, no. 912, pp. 2122–2144, 2017.

[9] K. He, S. X.-D. Tan, H. Zhao, X.-X. Liu, H. Wang, and G. Shi, "Parallel GMRES solver for fast analysis of large linear dynamic systems on GPU platforms," *Integration*, vol. 52, pp. 10–22, 2016.

[10] L. Chen, S. G. Petiton, L. A. Drummond, and M. Hugues, "A communication optimization scheme for basis computation of krylov subspace methods on multi-GPUs," in *Proceedings of the 11th International Conference on High Performance Computing for Computational Science*, pp. 31–36, Eugene, OR, USA, June 2014.

[11] J. Gao, Y. Zhou, G. He, and Y. Xia, "A multi-GPU parallel optimization model for the preconditioned conjugate gradient algorithm," *Parallel Computing*, vol. 63, pp. 1–16, 2017.

[12] M. Ament, G. Knittel, D. Weiskopf et al., "A parallel preconditioned conjugate gradient solver for the Poisson problem on a multi-GPU platform," in *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-Based Processing, PDP 2010*, IEEE Computer Society, Pisa, Italy, February 2010.

[13] H. Ji, M. Sosonkina, and Y. Li, "An implementation of block conjugate gradient algorithm on CPU-GPU processors," in *Proceedings of the Hardware-software Co-design for High Performance Computing*, IEEE, New Orleans, LA, USA, November 2014.

[14] A. Hartwig, S. Tomov, P. Luszczek, I. Yamazaki, D. Jack, and W. Sawyer, "Optimizing krylov subspace solvers on graphics processing units," in *Proceedings of the 2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops*, pp. 941–949, Phoenix, AZ, USA, May 2014.

[15] R. Helfenstein and J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," *Journal of Computational and Applied Mathematics*, vol. 236, no. 15, pp. 3584–3590, 2012.

[16] A. F. P. D. Camargos and V. C. Silva, "Performance analysis of multi-GPU implementations of krylov-subspace methods applied to FEA of electromagnetic phenomena," *IEEE Transactions on Magnetics*, vol. 51, no. 3, pp. 1–4, 2015.

[17] X. X. Liu, H. Wang, and X. D. Tan, "Parallel power grid analysis using preconditioned GMRES solver on CPU-GPU platforms," in *Proceedings of the 2013 IEEE/ACM International Conference on Computer Aided Design*, San Jose, CA, USA, November 2013.

[18] J. I. Aliaga, E. Dufrechou, P. Ezzatti et al., "An efficient GPU version of the preconditioned GMRES method," *The Journal of Supercomputing*, vol. 75, no. 3, pp. 1455–1469, 2018.

[19] V. Minden, B. F. Smith, and M. G. Knepley, "Preliminary implementation of PETSc using GPUs," in *Proceedings of the 2010 Workshop of GPU Solutions to Multiscale Problems in Science and Engineering*, Harbin, China, July 2010.

[20] Y. Saad, *Iterative Methods for Sparse Linear systems*, PWS Pub. Co., Boston, MA, USA, 2009.

[21] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," NVIDIA Technical Report NVR-2011-001, June 2011.

[22] A. Kashi and S. Nadarajah, "Fine-grain parallel smoothing by asynchronous iterations and incomplete sparse approximate inverses for computational fluid dynamics," in *Proceedings of the AIAA Scitech 2020 Forum*, Orlando, FL, USA, January 2020.

[23] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM Journal on Scientific Computing*, vol. 37, no. 2, pp. C169–C193, 2015.

[24] E. Chow, H. Anzt, and J. Dongarra, "Asynchronous iterative algorithm for computing incomplete factorizations on GPUs," *High Performance Computing*, Springer International Publishing, Berlin, Germany, 2015.

[25] M. Dessole and F. Marcuzzi, "Fully iterative ILU preconditioning of the unsteady Navier-Stokes equations for GPGPU," *Computers & Mathematics with Applications*, vol. 77, no. 4, pp. 907–927, 2019.

[26] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *Proceedings of the European Conference on Parallel Processing*, Springer, Vienna, Austria, August 2015.

[27] H. Anzt, E. Chow, D. B. Szyld et al., "Domain overlap for iterative sparse triangular solves on GPUs," *Software for Exascale Computing-SPPEXA 2013-2015*, Springer International Publishing, Berlin, Germany, 2016.

[28] CUDA Toolkit Documentation for cuSPARSE, 2020, https://docs.nvidia.com/cuda/cusparse/.

[29] R. Eberhardt and M. Hoemmen, "Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures," in *Proceedings of the IEEE 2016 International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, Chicago, IL, USA, May 2016.

[30] Developer Reference for Intel oneAPI Math Kernel Library-C, 2020, https://software.intel.com/en-us/onemkldeveloper-reference-c.

[31] S. Balay, S. Abhyankar, M. F. Adams et al., "PETSc web page," 2019, https://www.mcs.anl.gov/PETSc.

[32] X. C. Cai and Y. Saad, "Overlapping domain decomposition algorithms for general sparse matrices," *Numerical Linear Algebra with Applications*, vol. 3, no. 3, pp. 221–237, 1994.

[33] PETSc Documentation: FAQ, 2020, https://www.mcs.anl.gov/petsc/documentation/faq.html.

[34] CUDA Toolkit Documentation for cuBLAS, 2020, https://docs.nvidia.com/cuda/cublas/.

[35] J. Sanders and E. Kandrot, *CUDA by Example—An Introduction to General-Purpose GPU Programming*, Addison-Wesley Professional, Boston, MA, USA, 2010.

[36] K. Rupp, P. Tillet, F. Rudolf et al., "ViennaCL—linear algebra library for multi- and many-core architectures," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S412–S439, 2016.

[37] OpenMPI: Open Source High Performance Computing, 2010, https://www.openmpi.org/doc/v3.1/.

[38] T. A. Davis and Y. Hu, "The university of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, pp. 1–25, 2011.