

# Point-block incomplete LU preconditioning with asynchronous iterations on GPU for multiphysics problems

The International Journal of High Performance Computing Applications 1–15

© The Author(s) 2020

Article reuse guidelines:

sagepub.com/journals-permissions

DOI: 10.1177/1094342020981153

journals.sagepub.com/home/hpc



Wenpeng Ma<sup>1</sup> and Xiao-Chuan Cai<sup>2</sup> 

## Abstract

Point-block matrices arise naturally in multiphysics problems when all variables associated with a mesh point are ordered together, and are different from the general block matrices since the sizes of the blocks are so small one can often invert some of the diagonal blocks explicitly. Motivated by the recent works of Chow and Patel and Chow et al., we propose an efficient incomplete LU (ILU) preconditioner for point-block matrices targeting applications on GPU. The construction of the preconditioner involves two critical steps: (1) the initial guessing of values for the lower and upper triangular matrices; and (2) several sweeps of asynchronous updating of the triangular matrices. Three representative problems are studied to show the advantage of the proposed point-block approach over the standard point-wise approach in terms of the number of GMRES iterations and also the total compute time. Moreover, we compare the proposed algorithm with the level-scheduling based parallel algorithm employed in NVIDIA's cuSPARSE library as well as the serial method implemented in Intel MKL library, and the experiments show that a  $2\times-5\times$  speedup can be achieved over the block-based ILU( $p$ ) factorizations from the cuSPARSE library.

## Keywords

Point-block matrix, inexact ILU preconditioning, GPU architecture, asynchronous iteration, multiphysics problems

## Introduction

Incomplete LU factorization (ILU) is one of the most important building blocks for constructing preconditioners for solving large linear system of equations. However, the parallelization of ILU is still a bottleneck in high performance computing, especially on GPU accelerators, because it is inherently a sequential process. In order to improve the parallel performance of ILU, several versions of the algorithm have been introduced. These algorithms can be classified into three categories. One category is often referred to as the parallel level-scheduling algorithms studied in Saad (2003), Pakzad et al. (1997) that aim to efficiently solve the upper/lower sparse triangular systems; see Anderson and Saad (1989). This approach involves two steps, a pre-processing step that analyzes the parallelism of the problem and groups the components into different level sets, and then parallel solve that carries out the actual parallel computations on each level. Since the computing between different level sets is performed sequentially, the efficiency of this approach depends heavily on the sparsity pattern of the matrix. NVIDIA's cuSPARSE (2014) library has a GPU implementation of this algorithm. One drawback of this method is that it sometimes takes a considerable amount of time to find the parallelism in the

pre-processing step, which significantly affects the overall performance.

The second category is based on a technique that rearranges the ordering of the matrix into a new matrix by using some multi-coloring schemes, as in Saad (2003), Poole and Ortega (1987), for example red-black coloring when the number of desired colors is 2. Compared to the level-scheduling algorithms, this method is capable of capturing more parallelism but the resulting ILU preconditioner may become less effective than the preconditioner constructed based on the original matrix. GPU performance of the method can be found in Li and Saad (2013).

The third category consists of "inexact" ILU algorithms developed for the purpose of achieving higher parallel performance by not satisfying some constraints required by the regular ILU factorization. Chow and Patel (2015) proposed

<sup>1</sup> School of Computer and Information Technology, Xinyang Normal University, Henan, China

<sup>2</sup> Department of Mathematics, University of Macau, Macau, China

## Corresponding author:

Xiao-Chuan Cai, Department of Mathematics, University of Macau, Macau, China.

Emails: xccai@um.edu.mo; cai@cs.colorado.edu

an iterative method in which nonzeros in  $L$  and  $U$  are updated asynchronously without any data dependency. Compared with the traditional exact ILU, their experiments showed that the inexact ILU algorithm often requires five sweeps to achieve the same convergence level as the regular ILU. Chow et al. (2015) then implemented their algorithm on GPUs with highly efficient light-weighted threads, and their experiments show a  $2.0 \times$ – $28.9 \times$  speedup for some matrices from Davis and Hu (2011) over NVIDIA's cuSPARSE library. This algorithm is already integrated into popular parallel numerical packages including PETSc Balay et al. (2020) and ViennaCL Rupp et al. (2016). There are many other works on parallel factorizations on shared or distributed memory architectures, for examples Abdelfattah et al. (2016a), Rennich et al. (2016) and Anzt et al. (2017).

In this paper, we address matrices from multiphysics problems in which each mesh point has more than one unknowns. For example, in a two-dimensional flow field, there are two velocity variables and one pressure variable, with proper ordering, the resulting matrix would have  $3 \times 3$  blocks. Different from traditional block matrices with large blocks, this kind of block matrix has sufficiently small blocks so that the inverse of the diagonal blocks can be hard-coded. In the rest of paper, we refer to this type of matrices as point-block matrices. It is important to point out that for all algorithms to be introduced in this paper for point-block matrices, only matrix–vector and matrix–matrix operations are required, and there is no scalar operations. Note that a block is considered as a non-zero block if one of its elements is non-zero. On the other hand, we refer to a general sparse matrix as a point-wise matrix. Note that algorithms involving point-wise matrices require many scalar operations.

Point-block matrices arise from multiphysics problems naturally. Mathematically speaking, they can be treated as point-wise matrices when every element in a block is independently considered and zero values are not allocated. But numerical instability may occur if all zero elements in blocks are eliminated, because some coupling between the variables associated with the same mesh point is removed consequently. While with the point-block format, each block is supposed to be stored and computed as a unit without removing the zeros in a non-zero block, then the coupling feature of the physics is preserved. This is equivalent to say that fill-ins are allowed between unknowns associated with the same point-block or mesh point. Generally speaking, this consumes more memory, also involves more arithmetical operations, but for some applications this offers more stability and parallelism, thus more robust and faster in terms of the total compute time. Note that there are some classical papers for ILU factorizations of block matrices; for examples Saad and Zhang (1999a, 1999b) and Axelsson et al. (1989), and the parallelization of block ILU factorization is realized from either level-scheduling or multi-coloring algorithm on point-wise matrices in Chen et al. (2018) and Yang and Liu (2015). Some other parallel block ILU schemes are studied in Luo et al. (2015a), Luo et al. (2015b), Kim and Yun (2000) and Yun (2000).

The level of parallelism for exact ILU for certain sparse matrices may not be enough for the high concurrency of a GPU card, therefore, following Chow et al. (2015) we only attempt to compute an inexact ILU. Unlike the point-wise operations in Chow et al. (2015), for some applications the point-block matrix can offer more parallelism. Unlike standard block algorithms in which the inversion of the diagonal blocks requires LU factorizations, for point-block matrices, we invert the diagonal blocks by an explicit formula. Generally speaking, as a preconditioner, inexact incomplete LU is not as strong as the corresponding exact incomplete LU for many problems, therefore, some sweeps are often performed. As expected, synchronization hurts the performance of GPU badly, therefore we apply these sweeps asynchronously.

The rest of the paper is organized as follows. We first provide a quick review of incomplete LU factorization in point-block format in the second section. In the third section, an asynchronous point-block ILU( $p$ ) factorization is introduced, then an efficient GPU implementation of the proposed algorithm is discussed in detail. Three multicomponent problems are studied and the performance comparisons are reported in the fourth section. Some concluding remarks are given in the final section.

## Standard point-block ILU( $p$ ) factorization

We consider a  $n \times n$  sparse matrix  $A_{n \times n} = \{A_{i,j}, i, j = 1, \dots, m\}$  as a  $m \times m$  point-block matrix. For simplicity we assume all blocks are of the same size  $l \times l$ , and we also assume  $l$  is small since it is often related to the number of variables associated with a single point (or an element) of the finite element mesh. For example,  $l = 3$  for two-dimensional Navier–Stokes equations with two velocity variables and one pressure variable. The complete point-block LU factorization of  $A_{n \times n}$  is expressed as

$$A = L^{\text{exact}} U^{\text{exact}}$$

where  $L^{\text{exact}} = \{L_{i,j}, i, j = 1, \dots, m\}$  is a block lower triangular matrix,  $U^{\text{exact}} = \{U_{i,j}, i, j = 1, \dots, m\}$  is a block upper triangular matrix.

When applying the factorization as a preconditioner, we are interested in an incomplete version of the point-block LU factorization (ILU) with a sparsity pattern  $S_p = \{(i, j), 1 \leq i \leq m, 1 \leq j \leq m\}$

$$A = LU - R$$

where  $L = \{L_{i,j}, i, j = 1, \dots, m\}$  is a block lower triangular matrix,  $U = \{U_{i,j}, i, j = 1, \dots, m\}$  is a block upper triangular matrix, and  $L_{i,j} = U_{i,j} = 0, \forall (i, j) \notin S_p$ , and  $R_{i,j} = 0, \forall (i, j) \in S_p$ . In general, the sparsity pattern is chosen to be the same as the non-zero block positions in  $A$ , and the resulting factorization is known as the point-block ILU(0) factorization. The sparsity pattern can be extended by defining the level of fill, as in Saad (2003), which produces the point-block ILU( $p$ ) factorization with  $p$  fill-in levels.

It is straightforward to write the constraints of incomplete LU factorization (Chow and Patel, 2015; Chow et al., 2015) in the point-block format as

$$\sum_{k=1}^{\min(i,j)} L_{i,k} U_{k,j} = A_{i,j}, \quad (i, j) \in S_p \quad (1)$$

Then we can give the explicit expressions of  $L_{i,j}$  and  $U_{i,j}$  as

$$L_{i,j} = \left( A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} U_{k,j} \right) U_{j,j}^{-1} \quad (i > j) \quad (2)$$

$$U_{i,j} = A_{i,j} - \sum_{k=1}^{i-1} L_{i,k} U_{k,j} \quad (i \leq j)$$

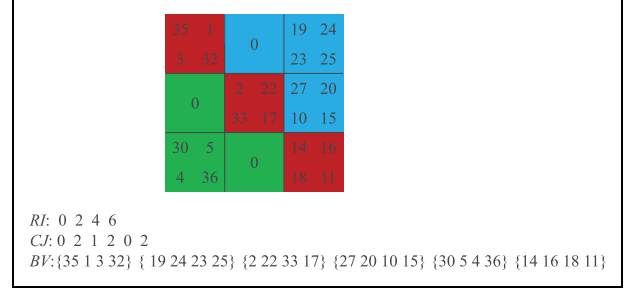
To keep the coupled feature of the physical quantities associated with a mesh point, we treat a block in the point-block matrix as an unit and store the values consecutively. Compared to the point-wise format, (2) looks similar but contains totally different computation: All scalars are changed into  $l \times l$  block matrices, the multiplication of two scalars becomes a matrix-matrix multiplication and the inversion of a scalar is substituted by the inverse of a  $l \times l$  block matrix. Furthermore, the data structures for point-wise matrices and point-block matrices are also different in the software implementation.

The popular storage format for point-block matrices is known as the Block Compressed Sparse Row (BCSR) format, which is employed in libraries like PETSc (Balay et al., 2020) and NVIDIA's cuSPARSE (2014). Given a  $n \times n$  point-block matrix  $A_{nn}$  with  $m$  block rows and columns, block size  $l$  and  $nnzb$  non-zero blocks, the BCSR format stores the matrix using three arrays ( $RI, CJ, BV$ ) as following.

- $RI$  is an integer array with length  $m + 1$  where the  $i^{th}$  element indicates the starting column index of the  $i^{th}$  row in  $CJ$ . The last element equals to  $nnzb$ .
- $CJ$  is an integer array with length  $nnzb$ , and stores all column indices of non-zero blocks by block rows.
- $BV$  is an array with length  $nnzb \times l \times l$ , and stores all non-zero block values by block rows. The  $l^2$  values are consecutively stored either by rows or columns within a block.

Figure 1 shows a  $6 \times 6$  matrix written as a  $3 \times 3$  point-block matrix with block size 2. Below the matrix, the BCSR format is illustrated in three arrays. Within each block, we employ a row-major ordering, which is also the ordering we use for the rest of the paper.

Algorithm 1 lists the steps of a point-block ILU factorization with fill-in levels. The algorithm requires the BCSR format of the input matrix. To perform the operations in (2), it is convenient to express  $L$  as row blocks and  $U$  as column blocks during the computation. Therefore, the algorithm outputs a lower triangular matrix  $L$  in BCSR format and an upper triangular matrix  $U$  in Block Compressed Sparse Column (BCSC) format. The factorization process is



**Figure 1.** A sample point-block matrix with  $n = 6$ ,  $m = 3$ ,  $l = 2$ ,  $nnzb = 6$ , and  $S = \{(0, 0), (0, 2), (1, 1), (1, 2), (2, 0), (2, 2)\}$ .

---

#### Algorithm 1. Standard point-block ILU( $p$ ) factorization.

---

##### Input:

1. a point-block matrix in BCSR format,  $A$ .
2. fill-in level,  $p$ .

##### Output:

1. a lower triangular matrix in BCSR format,  $L$ .
  2. an upper triangular matrix in BCSC format,  $U$ .
- 1: Symbolic factorization: estimate non-zero block positions, with level  $p$  in  $L$  and  $U$ .
  - 2: Allocate memory for  $L$  and  $U$ , respectively.
  - 3: Initialize  $L$  by  $L_{i,i} = I_{l,l}$ , and let  $L_{i,j} = A_{i,j}$  ( $i > j$ ). Initialize  $U_{i,j}$  with  $A_{i,j}$  ( $i \leq j$ ).
  - 4: **for**  $s = 1$  to  $m$  **do**
    - 4.1: compute the  $s^{th}$  row of  $L_{s,j}$  ( $s > j$ ):
$$L_{s,j} = (A_{s,j} - \sum_{k=1}^{j-1} L_{s,k} U_{k,j}) U_{j,j}^{-1}$$
    - 4.2: compute the  $s^{th}$  column of  $U_{i,s}$  ( $i \leq s$ ):
$$U_{i,s} = A_{i,s} - \sum_{k=1}^{i-1} L_{i,k} U_{k,s}$$
- end for**
- 

described as four steps. The first step is known as symbolic factorization (Balay et al., 2020; Saad, 2003; Yang and Liu, 2015) which sets up a non-zero pattern for  $L$  and  $U$  and incorporates the fill-in level  $p$ . The second step is to identify the number of non-zero blocks in  $L$  and  $U$  and allocate memory for both matrices. In the third step, the values of  $L$  and  $U$  are initialized by the lower and upper part of  $A$ , respectively. More precisely, we set the block-diagonal of  $L$  to  $l \times l$  identity matrices that serves as an initial condition for nonlinear constraints in (2). The fourth step is the numerical factorization which performs a loop over all blocks in a specific order to compute the corresponding block values in  $L$  and  $U$ . Inside the loop, it updates the  $s^{th}$  block-row of  $L$  first and then the  $s^{th}$  block-column of  $U$ , which is a sequentially dependent step that is necessary to obtain the correct factorization. Thus, in this updating order, it requires only one loop to calculate the exact lower and upper factors with  $p$  fill-in levels.

## Asynchronous point-block ILU( $p$ ) factorization for GPUs

Algorithm 1 is not suitable for fine-grained parallel computing using GPUs because of the sequential nature of

**Algorithm 2.** Asynchronous point-block ILU( $p$ ) factorization.**Input:**

1. a point-block matrix in BCSR format,  $A$ ;
2. the non-zero pattern with  $p$  fill-in levels,  $S_p$ ;
3. the maximum number of iterations,  $it_{max}$ ;
4. a tolerance for the residual,  $\epsilon$ ;

**Output:**

1. a lower triangular matrix in BCSR format,  $L$ ;
2. an upper triangular matrix in BCSC format,  $U$ ;
- 1: Symbolic factorization: estimate non-zero block positions, with level  $p$  in  $L$  and  $U$ .
- 2: Allocate memory for  $L$  and  $U$ , respectively.
- 3: Initialize  $L$  by  $L_{i,i} = I_{l,l}$ , and let  $L_{i,j} = A_{i,j} (i > j)$ . Initialize  $U_{i,j}$  with  $A_{i,j} (i \leq j)$ .  
 $it = 0$ ;
- 4: **loop**  
 $it = it + 1$ ;
- 4.1: map  $(i, j) \in S_p$  to an available computing unit
- 4.2: update  $L_{i,j} (i > j)$ :  
 $L_{i,j}^{it} = (A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}^{curr} U_{k,j}^{curr}) (U_{j,j}^{curr})^{-1}$
- 4.3: update  $U_{i,j}^{it} (i \leq j)$ :  
 $U_{i,j}^{it} = A_{i,j} - \sum_{k=1}^{i-1} L_{i,k}^{curr} U_{k,j}^{curr}$
- 4.4: compute  $r$  in (3)
- 5: **until**  $r < \epsilon$  or  $it \geq it_{max}$

Steps 4.1 and 4.2. For point-wise ILU, a powerful idea was introduced in Chow and Patel (2015) that updates  $L_{i,j} (i > j)$  and  $U_{i,j} (i \leq j)$  in an order that avoids the data dependency. Such an approach makes it possible to compute many entries of  $L$  and  $U$  in parallel. But, at the same time, the updating order makes it impossible to compute the exact values in  $L$  and  $U$ , as a result, the incomplete factorization is inexact. To make the ‘‘inexact and incomplete’’ factorization useful as a preconditioner, Chow and Patel (2015) suggests to include an outer loop to improve the exactness of the method without sacrificing the parallelism. In this paper, we extend the point-wise algorithm to point-block matrices, which is referred to as the asynchronous point-block ILU( $p$ ) described in Algorithm 2.

Since the constraints in (2) are nonlinear, the algorithm is nonlinear and the iteration is controlled by two input parameters,  $it_{max}$  and  $\epsilon$ .  $it_{max}$  specifies the maximum number of sweeps that the constraint equations are executed, and  $\epsilon$  is a tolerance that determines the inexactness of the resulting ILU factorization. Similar to the definition of the residual of ILU factorization in Chow and Patel (2015), we define

$$r = \left( \sum_{(i,j) \in S_p} \left\| A_{i,j} - \sum_{k=1}^{\min(i,j)} L_{i,k} U_{k,j} \right\|_F^2 \right)^{\frac{1}{2}} \quad (3)$$

as the residual of the point-block inexact ILU factorizations. Note that  $r = 0$  when the factorization is exact.

One important step for the fine-grained parallelization of the algorithm is to map chunks of the computing tasks determined by non-zero blocks of  $S_p$  to the computing

units. Then each computing unit is responsible for updating tasks in a sub-set of  $S_p$ . Different hardware architecture or programming models lead to different tasks to computing unit mapping strategies. For example, on CPUs or Intel MIC (Nguyen, 2017) architecture, a computing unit usually refers to a thread, and a computing unit on GPUs could be a thread or a warp (all threads in a warp execute the same instruction). The task of mapping  $S_p$  to threads can be easily accomplished by using OpenMP directives on CPUs or Intel MIC. Specifically, the key word *parallel for* can be added before Step 4 in Algorithm 1 to distribute the computing task to a number of OpenMP threads automatically, and it is always followed by the key word *schedule(mode)* to control how the iterations are divided into chunks and how the chunks are assigned to threads. On the GPU architecture, however, it is the programmer’s responsibility to work out a strategy for distributing  $\|S_p\|$  tasks of Steps 4.2–4.3 in Algorithm 2 to GPU threads. This is implemented by developing a GPU kernel to specify a proper workload of Steps 4.2–4.3 for each thread according to thread identifiers. Therefore, instead of executing Step 4.2 and Step 4.3 in a sequential order, they are executed concurrently by launching the kernel in thousands of light weighted threads on GPUs.

As described in the exact point-block ILU( $p$ ) algorithm (Steps 4.1–4.2 of Algorithm 1),  $L_{i,j}$  (or  $U_{i,j}$ ) is updated in an order that  $L_{i,j}$  (or  $U_{i,j}$ ) is not computed until the data it depends on has been calculated. In this paper, to gain more parallelism, we remove the data dependency by updating all  $L_{i,j}$  and  $U_{i,j}$  in several computing units simultaneously without any synchronizations. Therefore, using different number of computing units yields different updating orders, and then different factorization results. Note that even when the number of computing units is fixed, the results of the computation from two different test runs could also be different because the sequence of data accesses could be different. When the task is divided by blocks, asynchronous happens only among different blocks because the calculations within a block are sequential.

To describe an asynchronous version of Algorithm 1, we define two random matrices

- $L_{i,j}^{curr} (i, j = 1, \dots, m)$ : The current version of the matrix  $L_{i,j}$  whose values are being calculated or determined by the hardware.
- $U_{i,j}^{curr} (i, j = 1, \dots, m)$ : The current version of the matrix  $U_{i,j}$  whose values are being calculated or determined by the hardware.

Following (2), the process for computing  $L_{i,j}^{curr}$  and  $U_{i,j}^{curr}$  is given by

$$L_{i,j}^{curr} = \left( A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}^{curr} U_{k,j}^{curr} \right) (U_{j,j}^{curr})^{-1} \quad (i > j)$$

$$U_{i,j}^{curr} = A_{i,j} - \sum_{k=1}^{i-1} L_{i,k}^{curr} U_{k,j}^{curr} \quad (i \leq j)$$

but the outcome of the process is not necessarily the same as that of (2). The resulting values in  $L$  and  $U$  could be far from the desired values, especially in the point-block case ( $l \geq 1$ ) since some of the calculations are “unfinished” before they are fetched by the neighboring computing units. To improve the results, some sweeps are performed as

$$\begin{aligned} & \text{For } it = 1 \text{ to } it_{\max} \\ & L_{i,j}^{curr} = \left( A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}^{curr} U_{k,j}^{curr} \right) (U_{j,j}^{curr})^{-1} \quad (i > j) \\ & U_{i,j}^{curr} = A_{i,j} - \sum_{k=1}^{i-1} L_{i,k}^{curr} U_{k,j}^{curr} \quad (i \leq j) \\ & \text{Endfor} \end{aligned}$$

Here  $it$  is a loop identifier from 1 to  $it_{\max}$ . If  $it_{\max}$  is sufficiently large, then the results of the loop is usually the same as (2). The details of the algorithm are provided in Algorithm 2.

For point-wise matrices, Chow et al. (2015) presented a GPU implementation in which a CUDA thread is responsible for computing a scalar value  $l_{i,j}$  or  $u_{i,j}$ . This strategy is effective and efficient because it ensures that each CUDA thread is assigned certain amount of workload and writes the newest data back into the GPU’s global memory quickly for the use by other threads. Similarly, the idea can be extended to point-block matrices by letting a CUDA thread compute and update a block in  $L$  or  $U$ . However, this strategy is not suitable for Algorithm 2 on GPUs because the matrix-matrix multiplication involves much more computations than the scalar multiplication, which is more than what a lightweight CUDA thread can handle. Moreover, if one CUDA thread writes  $l^2$  values in a block simultaneously but the two adjacent threads fail to access adjacent global memory addresses, the global memory accesses would be far from coalesced. The uncoalesced global memory accesses will lead to poor performance of writing new data, and this keeps the running threads from using the newest data from other threads.

The works about block sparse matrix–vector multiplication by Abdelfattah et al. (2016b) and Eberhardt and Hoemmen (2016) suggest that mapping a block to consecutive CUDA threads can offer a lot of benefits in performance improvement on GPUs. To make full use of the fine-grained feature of GPUs for computing all blocks of  $L$  and  $U$ , we decide to assign 32 threads in a warp to compute and update a block-row in both  $L$  and  $U$  collaboratively. This is summarized in detail in Algorithm 3.

In the GPU architecture, CUDA threads are dispatched in warps implicitly, while a kernel that states the computing tasks on threads is explicitly configured with two main parameters, the number of threads in a thread block and the total number of thread blocks. To identify which warp corresponds to a specific block-row in the point-block matrix, we manually map a global thread identifier to the warp identifier through dividing the global thread identifier by the warp size. For example, as shown in Figure 2, when the number of threads in a thread block is set to 128, the

### Algorithm 3. GPU implementation of Algorithm 2.

---

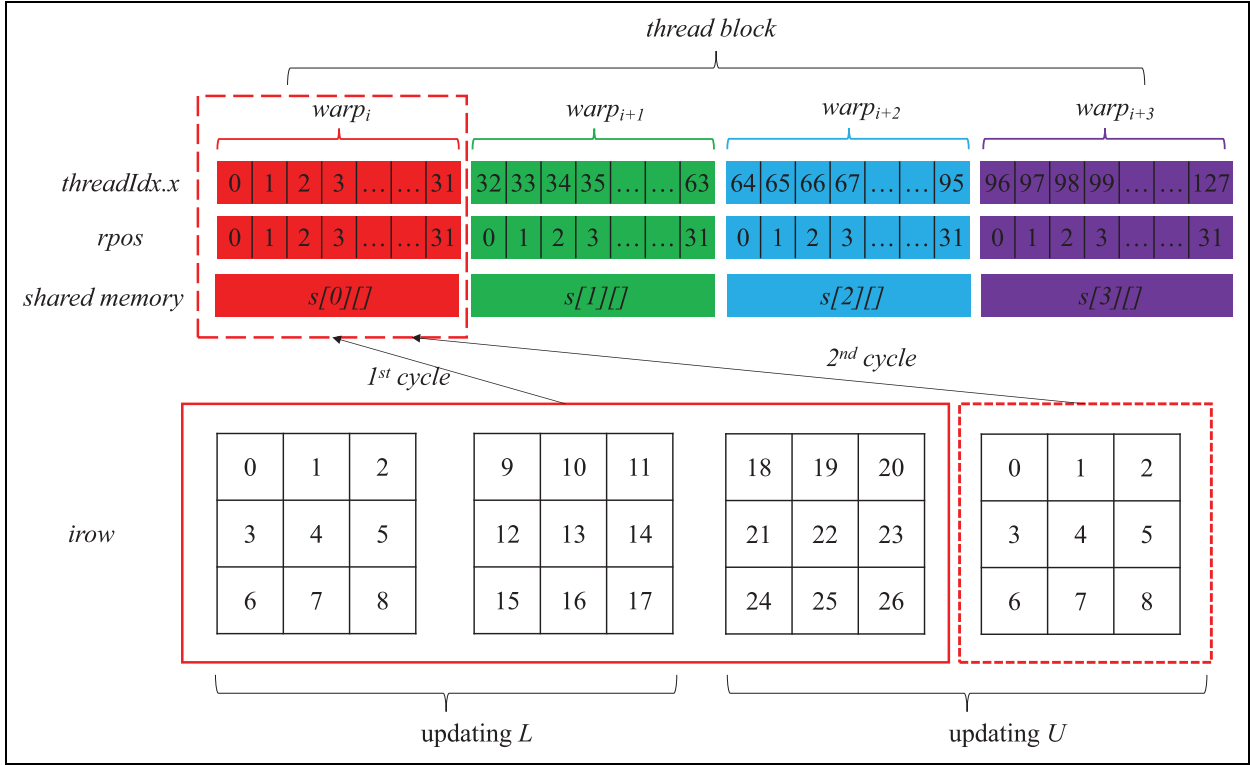
```

1: tid = blockDim.x × blockIdx.x + threadIdx.x;
2: __shared__ double s[warps_per_block][warp_size];
3: irow = tid/warp_size;
4: if irow < size then
5:   stidx = Arowptr[irow];
6:   edidx = Arowptr[irow + 1];
7:   rpos = threadIdx.x%warp_size;
8:   gpos = stidx × sbz2 + rpos;
9:   r = (rpos/sbz)%sbz;
10:  c = rpos%sbz;
11:  range = (warp_size/sbz2) × sbz2;
12:  if rpos < range then
13:    while gpos < edidx × sbz2 do
14:      idx = gpos/sbz2;
15:      jcol = Acolval[idx];
16:      lidx = Lrowptr[irow];
17:      ledidx = Lrowptr[irow + 1];
18:      uidx = Ucolptr[jcol];
19:      uedidx = Ucolptr[jcol + 1];
20:      s[warp_id][rpos] = 0.0;
21:      kmax = irow > jcol?jcol : irow;
22:      while lidx < ledidx && uidx < uedidx do
23:        kcol = Lcolval[lidx];
24:        krow = Urowval[uidx];
25:        if kcol == krow then
26:          if kcol < kmax then
27:            s[warp_id][rpos] += ∑ LtidUuidx
28:          end if
29:        end if
30:        if (kcol ≤ krow) lidx ++;
31:        if (kcol ≥ krow) uidx ++;
32:      end while
33:      s[warp_id][rpos] = Ablkval[gpos] −
34:        s[warp_id][rpos];
35:      if irow > jcol then
36:        all threads in the same block compute the
37:        determinant of Ujj: det
38:        det = 1.0/det; det = det × [s × Uj,j*]r,c;
39:        Lblkval[(lidx − 1)sbz2 + r × sbz + c]
40:          = det;
41:      else
42:        Ublkval[(uidx − 1)sbz2 + r × sbz + c]
43:          = s[warp_id][rpos];
44:      end if
45:      offset += range;
46:    end while
47:  end if

```

---

total 128 threads in a thread block are divided into four warps that correspond to four block rows of  $L$  and  $U$ , respectively. By knowing a certain identifier, i.e. a warp identifier, all threads in a warp can confirm the starting and ending indices of data segment they need to handle in a BCSR formatted matrix. Within each warp, the local thread identifier, expressed as  $rpos$ , can be calculated as the remainder of dividing thread identifier in a thread block ( $threadIdx.x$ ) by the warp size. By obtaining the local thread identifier in a warp, we explain how the threads



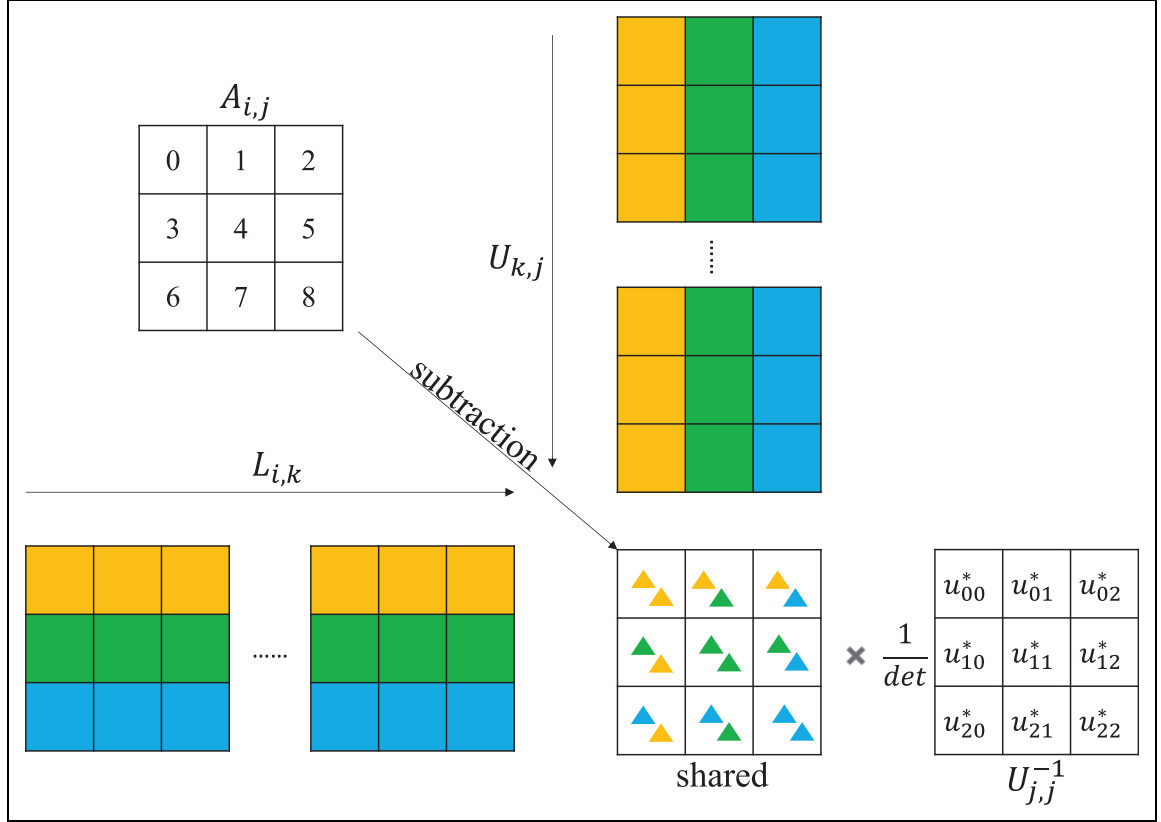
**Figure 2.** Illustration of updating the  $irow^{th}$  block-row on GPU. The first warp (32 threads) in a thread block performs two cycles to update the four blocks of the row: (1) the first 27 threads in the warp update the first three blocks (two blocks in  $L$  and one block in  $U$ ) showing in the box with red solid border; then (2) the first nine threads in the warp update the fourth block showing in the box with red dashed border.

in a warp work independently and collaboratively for a block-row.

In the rest of this section, we discuss the application of Algorithm 3 for a more concrete example with  $3 \times 3$  blocks. As illustrated in Figure 2, our strategy is to make a CUDA thread responsible for computing a single element of a block. Specifically, each element in a block is assigned to a thread in a warp, and elements are assigned by rows within a block. Different blocks are assigned from left to right. However, the total number of elements in a block-row may exceed the warp size, and the threads in a warp can cover only three complete blocks in this particular example. The assignment procedure is restarted at the fourth block and a thread in a warp may compute more than one element. For example, there are 4 blocks in total at the  $i^{th}$  block-row of  $L$  and  $U$ , and 27 threads are used in the first assignment cycle while only the first 9 threads will work in the second cycle. The second cycle does not start until all the 27 threads in the first cycle has completed their task. In the pseudo code of Algorithm 3,  $range$  is declared to estimate the maximum number of threads in a warp that can be used to cover complete blocks. Then all threads within  $range$  are activated to start the computation concurrently. Starting from line 14 in Algorithm 3, each thread begins to solve one element in  $L_{i,j}$  or  $U_{i,j}$  according to the constraints at Steps 4.2–4.3 of Algorithm 2.

We demonstrate the process of executing operation (2) in Figure 3. Each thread has to perform a multiplication of a row-vector and a column-vector when it encounters a pair of blocks,  $L_{i,k}$  and  $U_{k,j}$ . The result is accumulated as each thread goes over all block pairs  $\{L_{i,k}, U_{k,j}\}$  that satisfy  $k < \min\{i, j\}$ . In (2), the constraint for  $L_{i,j}$  has one more operation than that for  $U_{i,j}$ , and the extra operation involves a matrix–matrix multiplication which uses the accumulated values across the threads. To make the result from each private thread visible to other threads, we exploit shared memory on GPUs instead of registers to save the intermediate results. Unlike registers, shared memory is allocated per thread block and can be accessed by all threads in the same thread block. Therefore, we allocate a space with the thread block size on shared memory to guarantee that each thread can store a value with double precision type. In line 2 in Algorithm 3, the space is arranged by a two-dimensional array for accessing shared data in warps.

To update  $L_{i,j}$  according to the first constraint in (2), the inverse matrix of  $U_{j,j}$  is needed by the threads that handle the same block. This can be accomplished by two steps. First, a local thread identifier, i.e.  $rpos$ , is transformed into a  $(r, c)$  pair where  $r$  and  $c$  represent the row and column index within the corresponding  $l \times l$  block respectively. The transformation step is listed as line 9 and 10 in Algorithm 3. Second, the computation of the element at  $(r, c)$  of  $(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} U_{k,j}) U_{j,j}^{-1}$  is assigned to the thread



**Figure 3.** Solving the constraints of  $L_{i,j}$  on GPU using nine threads via three steps: (1) each thread computes one scalar element of  $\sum_{k=1}^{j-1} L_{i,k} U_{k,j}$  and stores the result in the shared memory showing as two coloring triangles. The color of the upper and lower triangle shows the scalar rows of  $L_{i,k}$  and columns of  $U_{k,j}$  each thread needs to go over, respectively; (2) each threads accesses one element of  $A_{i,j}$  to perform an in-place block subtraction in the shared memory; and (3) perform a block–block multiplication between the shared results and  $U_{j,j}^{-1}$  with the explicit expression.

corresponding to  $(r, c)$ . According to line 33 in Algorithm 3, the results of  $A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} U_{k,j}$  in a warp for 3 blocks are stored in the shared memory ( $s$ ) which can be accessed by all threads in the warp. But when performing the matrix–matrix multiplication, the thread corresponding to  $(r, c)$  needs not only the element at  $(r, c)$  of  $U_{j,j}^{-1}$  but other two elements at the  $c^{\text{th}}$  column of  $U_{j,j}^{-1}$ . This issue can be solved by giving a complete symbolic expression of  $U_{j,j}^{-1}$ . Suppose  $U_{j,j}$  and its adjoint matrix  $U_{j,j}^*$  are expressed as

$$U_{j,j} = \begin{bmatrix} u_{00} & u_{01} & u_{02} \\ u_{10} & u_{11} & u_{12} \\ u_{20} & u_{21} & u_{22} \end{bmatrix} \quad U_{j,j}^* = \begin{bmatrix} u_{00}^* & u_{01}^* & u_{02}^* \\ u_{10}^* & u_{11}^* & u_{12}^* \\ u_{20}^* & u_{21}^* & u_{22}^* \end{bmatrix}$$

we compute the symbolic expression of  $u_{rc}^*$  in  $U_{j,j}^*$  as

$$u_{rc}^* = u_{ab}u_{de} - u_{fg}u_{hk},$$

where  $a = \text{mod}(c + 1, 3)$ ,  $b = \text{mod}(r + 1, 3)$ ,  $d = \text{mod}(c + 2, 3)$ ,  $e = \text{mod}(r + 2, 3)$ ,  $f = \text{mod}(c + 1, 3)$ ,  $g = \text{mod}(r + 2, 3)$ ,  $h = \text{mod}(c + 2, 3)$ ,  $k = \text{mod}(r + 1, 3)$ . Then the determinant of  $U_{j,j}$  can be computed by

$$\begin{aligned} \det &= u_{00}u_{11}u_{22} - u_{00}u_{12}u_{21} - u_{01}u_{10}u_{22} \\ &\quad + u_{01}u_{12}u_{20} + u_{02}u_{10}u_{21} - u_{02}u_{11}u_{20} \end{aligned}$$

All threads in a block compute the determinant of  $U_{j,j}$  and save it as a register variable. Then each thread associated with a specific  $(r, c)$  accesses the elements in  $U$  and  $s$  to perform one row–column computation. For example, the thread at  $(1, 2)$  firstly computes  $u_{02}^*$ ,  $u_{12}^*$ , and  $u_{22}^*$  and then accesses the second (index starting from 0) row in  $s$  to perform a multiply–add computation. The result is written at  $(r, c)$  of block  $L_{irow,jcol}$  in Algorithm 3.

## Numerical experiments

In this section, we present some numerical experiments by using the proposed GPU version of point-block ILU factorization as a preconditioner for several multicomponent problems arising in incompressible flow calculations and phase field method for the modeling of crystal growth. We also provide some comparisons with results obtained using the NVIDIA’s cuSPARSE (2014) and Intel MKL (2017) libraries.

The experiments are carried out on a cluster of computing nodes, each consisting of 2 Intel E5-2680 V2 (Ivy Bridge, 10C, 2.8 GHz) CPUs, with 128 GB DDR3 ECC 1866 MHz physical memory and 2 Nvidia Tesla K20 GPGPU cards. The GPU card has the compute capability

of 3.5, 320-bit GDDR5 5 GB memory, 14 stream multi-processors (SMs), 2496 processors in total. Each SM has 65536 registers, and a thread can use a maximum number of 255 registers. The cluster is also configured with libraries CUDA Toolkit 6.5.14 and Intel MKL 11.1.

NVIDIA’s cuSPARSE (2014) library provides two ILU factorization functions. One is point-wise, and the other is block-based. The point-wise routine requires the calls of *cusparseDcsrilu02\_bufferSize*, *cusparseDcsrilu02\_analysis*, and *cusparseDcsrilu02* one by one in order to compute an ILU factorization for an input matrix. The first function calculates the memory requirement, the second function is responsible for extracting possible parallelism, and the last function computes the ILU factorization. cuSPARSE also offers the ILU factorizations of point-block matrices in the BCSR format, and the corresponding functions are *cusparseDbsrilu02\_bufferSize*, *cusparseDbsrilu02\_analysis* and *cusparseDbsrilu02*.

The Intel MKL 11.1 provides a function named *dcsrilu0* which computes point-wise ILU factorizations based on the algorithm proposed in Saad (2003).

To demonstrate the advantage of the proposed algorithm over the point-wise version implemented with the CUDA kernel by Chow et al. (2015), we denote the implementations as GPU\_PBILU and GPU\_PWILU, respectively. The difference between the two versions is that a non-zero block sub-matrix in a point-block format is treated as a dense matrix with all elements stored and computed, but in the point-wise format all zeros are removed and do not participate in any computation.

cuSPARSE and Intel MKL libraries only offer ILU factorization without fill-in levels (ILU(0)). To compare the performance of the aforementioned approaches with higher fill-in levels, we perform the symbolic step to obtain the non-zero pattern  $S_p$  on CPUs and the actual factorization is executed on GPUs.

In the experiments, we solve the linear system of equations with a right-preconditioned GMRES restarted at 30 until the following condition (Saad, 2003) is satisfied

$$\| (b - Ax^k)M^{-1} \| \leq r_{tol} \| b \|$$

where  $b$  is the right-hand side of the linear system,  $M^{-1}$  is the ILU preconditioner discussed in the previous sections and is used here as a right preconditioner, the initial guess  $x^0$  is chosen as 0, and some different values of the tolerance  $r_{tol}$  will be tested in the experiments.

For each experiment, we report the number of GMRES iterations denoted as “GMRES”, the total compute time of GMRES denoted as “GMRES<sub>time</sub>”, and the unpreconditioned residual  $\| b - Ax^k \|_2$  denoted as “GMRES<sub>resid</sub>”. The residual of the point-block inexact ILU factorization, as defined in (3), is denoted as “ILU<sub>resid</sub>”, and the compute time for the ILU factorization is denoted as “ILU<sub>time</sub>”. We use “ $it_{impv}$ ” to denote the number of sweeps used in the asynchronous ILU factorization. All results in our experiments are averaged over three runs.

**Table 1.** Comparisons of GPU\_PWILU and GPU\_PBILU when different number of sweeps is applied for two different relative tolerances of GMRES(30).

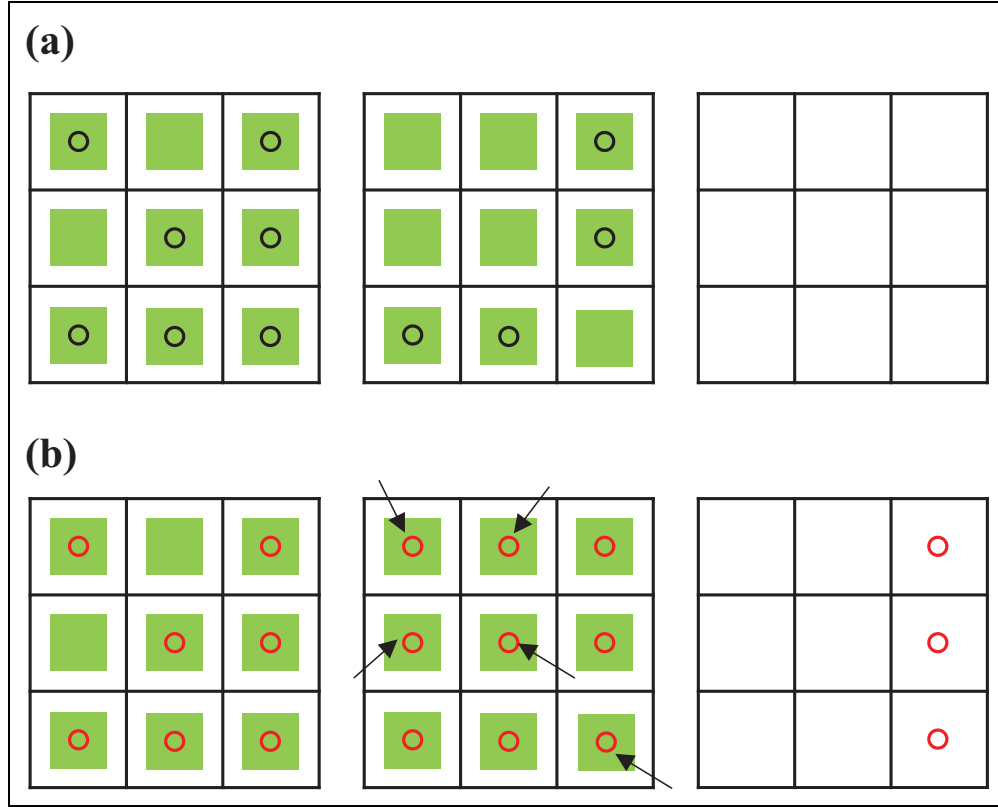
$it_{impv}$	GMRES	GMRES <sub>time</sub>	GMRES <sub>resid</sub>	ILU <sub>resid</sub>	ILU <sub>time</sub>
$r_{tol} = 10^{-5}$					
GPU_PWILU					
3	263	522 ms	2.11E-3	31.0	1.95 ms
4	292	576 ms	2.12E-3	9.54	2.56 ms
5	270	538 ms	2.13E-3	3.54	3.17 ms
GPU_PBILU					
3	172	410 ms	2.16E-3	2.48	1.71 ms
4	160	369 ms	2.09E-3	1.03	2.23 ms
5	148	344 ms	2.03E-3	0.58	2.76 ms
$r_{tol} = 10^{-6}$					
GPU_PWILU					
3	410	808 ms	2.15E-4	31.0	1.95 ms
4	410	811 ms	2.10E-4	9.61	2.56 ms
5	382	749 ms	2.11E-4	3.54	3.17 ms
GPU_PBILU					
3	246	560 ms	2.10E-4	2.47	1.71 ms
4	208	475 ms	2.04E-4	1.03	2.23 ms
5	200	456 ms	2.11E-4	0.58	2.76 ms

### A two-dimensional stokes problem

We consider a matrix from the finite element discretization of a two-dimensional Stokes problem on a mesh with 10201 mesh points. Each mesh point contains three unknowns including two velocity components ( $u, v$ ) and one pressure ( $p$ ), the resulting matrix  $A$  is of size  $30603 \times 30603$  with  $3 \times 3$  blocks.

Table 1 shows the performance of the ILU(0) factorizations as well as the number of GMRES(30) iterations when different number of asynchronous sweeps is applied. Two different convergence tolerances are considered as shown in the top and bottom part of Table 1. In this case, the ILU(0) sparsity pattern for the point-block matrix contains 70593 blocks (635337 values) while the sparsity pattern for the point-wise matrix only contains 388625 non-zero values. Although the ILU(0) factorization of the point-block matrix needs to solve 60% more constraints than that of the point-wise matrix, we observe from Table 1 that GPU\_PBILU still performs slightly better than GPU\_PWILU in terms of GPU factorization time. For GPU\_PBILU, the number of GMRES iterations decreases with increasing  $it_{impv}$ , thus the compute time for GMRES is reduced gradually. But in the GPU\_PWILU case, when  $it_{impv} = 3$  is applied, GMRES uses 263 iterations to converges to  $10^{-5}$ , which is 29 and 7 fewer than when  $it_{impv} = 4$  and  $it_{impv} = 5$  although the ILU<sub>resid</sub> is larger. This is due to the stochastic feature of applying the inexact factorization to GMRES. We also observe that GPU\_PBILU outperforms GPU\_PWILU in terms of GMRES<sub>time</sub> even though GMRES in the





**Figure 4.** A comparison of the sparsity patterns of the upper triangular  $U$  matrix from three methods: (1) the point-block ILU(0) showing as green squares in both (a) and (b); (2) the point-wise ILU(0) showing as the black circles in (a); and (3) the point-wise ILU(1) showing as the red circles in (b). The five elements pointed by arrows are additional nonzeros provided by the point-wise ILU(1) and the point-block ILU(0), not by the point-wise ILU(0).

point-block format is more expensive in terms of the arithmetic complexity, this is because point-block ILU(0) factorization yields fewer GMRES iterations compared to the point-wise ILU(0) factorization in this case.

Note that with the point-block ILU(0) preconditioner GMRES converges faster in terms of the number of iterations. This can be explained by Figure 4 which shows three consecutive blocks from a block-row of the upper triangular factor. To understand the faster convergence, we compare three preconditioners: point-block ILU(0), point-wise ILU(0), and point-wise ILU(1), and show that point-block ILU(0) retains more nonzeros than point-wise ILU(0) and is actually closer to point-wise ILU(1). Green squares in Figure 4(a) and Figure 4(b) show the non-zero  $3 \times 3$  blocks, and 18 constraints need to be solved for these three blocks in the point-block ILU(0). In the case of point-wise factorizations, the black circles in Figure 4(a) show the sparsity pattern of ILU(0) whereas the red circles indicate the sparsity pattern of ILU(1) in Figure 4(b). We can see clearly in Figure 4(b) that five circles pointed by arrows in the point-wise ILU(1) pattern overlap with the green squares in the second block and three circles are outside the two blocks showing in the third block, which is not covered by the point-block ILU(0). That means that the constraints of ILU(0) in the point-block format partially

cover the constraints of ILU(1) in the point-wise format, which makes the effectiveness of the point-block ILU(0) behave between the point-wise ILU(0) and ILU(1).

For ILU factorizations with higher fill-in levels, we compare the performance of our algorithm to that of popular libraries including cuSPARSE and Intel MKL. The first two parts of Table 2 show the execution times of three functions that accomplish the point-wise ILU factorization in the cuSPARSE library. They are followed by the corresponding results from point-block factorizations. We observe from the four parts that the level-scheduling scheme implemented in the cuSPARSE library is effective for this case in both point-wise and point-block formats, and the factorization time can be reduced by spending a small amount of time on analyzing the parallelism. The last two parts give the numbers of GMRES iterations and factorization times when the two inexact ILU factorizations are applied to solving the linear equations. We note that the exact point-block ILU(1), ILU(2) and ILU(3) factorization produced by cuSPARSE makes GMRES converge to  $10^{-3}$  at 24<sup>th</sup> step, 17<sup>th</sup> step and 11<sup>th</sup> step, respectively. And 5 sweeps of our GPU kernel can achieve the same convergence level for GMRES using 24 steps, 17 steps and 11 steps. However, compared to the point-wise factorization from cuSPARSE, GPU\_PWILU fails to offer effective

**Table 2.** 2D Stokes case: comparisons to cuSPARSE, Intel MKL libraries and GPU\_PWILU with higher fill-in levels. The tolerance of GMRES(30) is set to  $10^{-3}$ .

cuSPARSE (point-wise): with level-scheduling			
Time	ILU(1)	ILU(2)	ILU(3)
$T_{buffer\ size}$	0.13 ms	0.10 ms	0.12 ms
$T_{analysis}$	3.96 ms	5.38 ms	6.16 ms
$T_{csr\ ilu}$	11.17 ms	18.77 ms	28.17 ms
cuSPARSE (point-wise): without level-scheduling			
Time	ILU(1)	ILU(2)	ILU(3)
$T_{buffer\ size}$	0.14 ms	0.10 ms	0.12 ms
$T_{analysis}$	0.22 ms	0.22 ms	0.24 ms
$T_{csr\ ilu}$	21.53 ms	40.95 ms	44.20 ms
cuSPARSE (block-based): with level-scheduling			
Time	ILU(1)	ILU(2)	ILU(3)
$T_{buffer\ size}$	0.14 ms	0.16 ms	0.14 ms
$T_{analysis}$	1.21 ms	1.23 ms	1.51 ms
$T_{bsr\ ilu}$	17.49 ms	28.37 ms	51.67 ms
cuSPARSE (block-based): without level-scheduling			
Time	ILU(1)	ILU(2)	ILU(3)
$T_{buffer\ size}$	0.14 ms	0.16 ms	0.16 ms
$T_{analysis}$	0.15 ms	0.16 ms	0.18 ms
$T_{bsr\ ilu}$	41.89 ms	52.23 ms	68.83 ms
Intel MKL			
Time	ILU(1)	ILU(2)	ILU(3)
$T_{dcsr\ ilu}$	33.16 ms	82.03 ms	200.47 ms
GPU_PWILU			
$i_{impv} = 5$	ILU(1)	ILU(2)	ILU(3)
GMRES	150	NAN occurred	NAN occurred
$ILU_{time}$	11.79 ms		
GPU_PBILU			
$i_{impv} = 5$	ILU(1)	ILU(2)	ILU(3)
GMRES	24	17	11
$ILU_{time}$	3.72 ms	6.6 ms	12.18ms

factorizations by five sweeps, and NAN (Not A Number) occurs in the process of ILU(2) and ILU(3). This can be explained by the fact that the positions in  $S_p$  of ILU( $p$ ) ( $p \geq 1$ ) but not in that of ILU(0) are initialized as zero values, and the simple and straightforward way to give initial guesses in the lower(L) and upper(U) factors makes the point-wise cycles unstable. But this doesn't have any impact on GPU\_PBILU. Concerning the total compute time, ILU(1), ILU(2) and ILU(3) of GPU\_PBILU achieve  $5.06\times$ ,  $4.51\times$  and  $4.38\times$  over the point-block ILU factorization with the level-scheduling algorithm provided in the cuSPARSE library, respectively.

We now consider a tighter tolerance of  $10^{-10}$  for GMRES(40) with ILU( $p$ ) ( $p = 1, 2, 3$ ). Similar to the results for GPU\_PWILU in Table 2, GMRES with ILU(1)

**Table 3.** 2D Stokes case: GMRES(40) iterations and  $ILU_{time}$  using GPU\_PBILU. The tolerance of GMRES(40) is set to  $10^{-10}$ .

GPU_PBILU			
$i_{impv} = 8$	ILU(1)	ILU(2)	ILU(3)
GMRES(40)	1693	319	80
$ILU_{time}$	5.96 ms	10.58 ms	19.5 ms

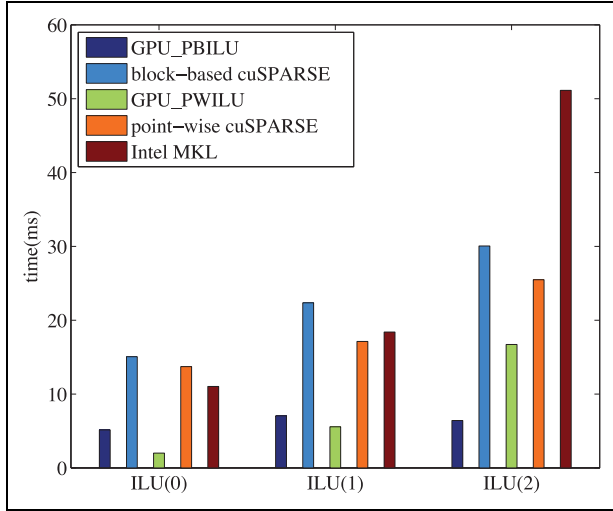
obtained by GPU\_PWILU fails to converge even with the increasing number of sweeps, and NAN occurs in GMRES with ILU(2) and ILU(3). By using GPU\_PBILU, we show the number of GMRES iterations and  $ILU_{time}$  in Table 3. The exact point-block ILU( $p$ ) ( $p = 1, 2, 3$ ) results in 1755, 295 and 80 GMRES iterations on the CPU. From Table 3, we observe that ILU(1) on GPU leads to fewer GMRES iterations compared to the CPU result, whereas ILU(2) on GPU leads to more iterations. And increasing the number of sweeps is not helpful to reduce the number of GMRES iterations for the ILU(2) case. This is probably because the success of solving the nonlinear constraints (Steps 4.2 and 4.3 in Algorithm 2) depends on the initial guesses, but our initial guesses are not good enough.

### A two-dimensional, nonlinear driven cavity flow problem

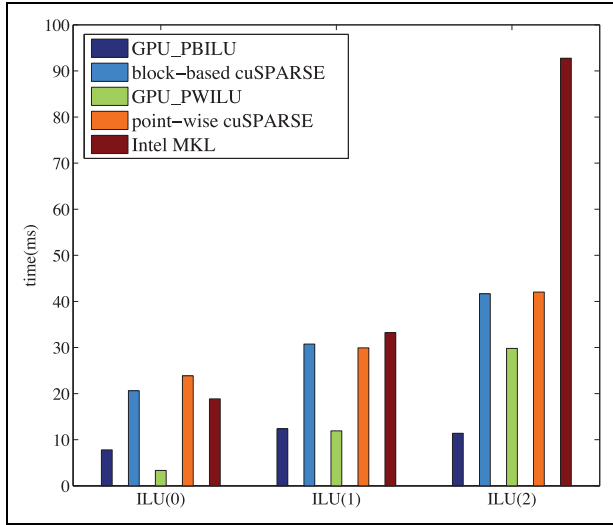
In this experiment, we investigate the effects of ILU( $p$ ) factorizations on solving linear systems in a nonlinear solver. The nonlinear algebraic system arises from the finite difference discretization of the 2D driven cavity flow problem modeled by the incompressible Navier–Stokes equations in the velocity-vorticity formulation. The Reynolds number is 1. The system is solved with an inexact Newton method in which the linear Jacobian systems are solved by a right-preconditioned GMRES. The Jacobian matrix is stored as a point-block matrix with  $3 \times 3$  blocks. The inexact Newton iteration is stopped when the following condition is satisfied

$$\|F(x^k)\| \leq 10^{-8} \|F(x_0)\|$$

where  $x_0$  is the initial guess. We consider two uniform meshes of size  $150 \times 150$  and  $200 \times 200$ . For these meshes, the number of inexact Newton iterations is 4. For each of the iterations, the Jacobian matrix is constructed and solved with a relative tolerance of  $10^{-3}$ . The linear systems for this experiment are extracted from the third step of the Newton iterations. The timing results are shown in Figure 5, Figure 6, Table 4 and Table 5. For the  $150 \times 150$  mesh, by employing the exact point-block ILU( $p$ ) ( $p \leq 2$ ) preconditioners on CPUs, GMRES(30) uses 365, 212 and 163 iterations to converge to  $10^{-3}$ , respectively. For point-block ILU(0) and ILU(1) on GPU, it requires 7 and 6 sweeps of GPU\_PBILU to make GMRES(30) converge with the same number of iterations obtained on the CPU. We observe that due to the stochastic



**Figure 5.**  $150 \times 150$  mesh: comparisons among GPU\_PBILU, GPU\_PWILU and popular libraries.



**Figure 6.**  $200 \times 200$  mesh: comparisons among GPU\_PBILU, GPU\_PWILU and popular libraries.

**Table 4.**  $it_{impv}$ ,  $ILU_{resid}$ , GMRES, and  $GMRES_{time}$  for GPU\_PBILU on  $150 \times 150$  mesh. “\” means no convergence.

Fill-in levels	GPU_PBILU			GPU_PWILU		
	ILU(0)	ILU(1)	ILU(2)	ILU(0)	ILU(1)	ILU(2)
$it_{impv}$	7	6	4	5	5	$\geq 5$
$ILU_{resid}$	7.54	5.58	171	60.2	538	\
GMRES	365	212	149	365	169	\
$GMRES_{time}$	1651 ms	1089 ms	872 ms	1374 ms	740 ms	\

feature of our algorithm, 4 sweeps are not only sufficient for ILU(2) to make GMRES(30) converge but also requires 14 fewer iterations. A speedup of  $2.93\times$ ,  $3.17\times$  and  $4.68\times$  can be obtained by our algorithm compared to the block-based version in cuSPARSE. GMRES(30) uses 367, 213 and 164 iterations respectively to converge using the exact

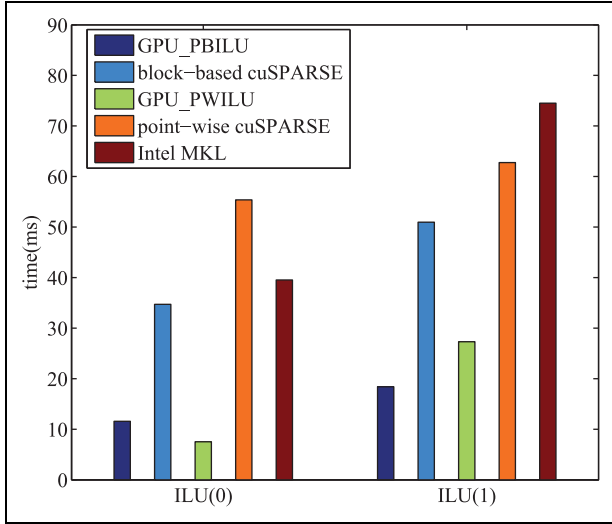
**Table 5.**  $it_{impv}$ ,  $ILU_{resid}$ , GMRES, and  $GMRES_{time}$  for GPU\_PBILU on  $200 \times 200$  mesh. “\” means no convergence.

Fill-in levels	GPU_PBILU			GPU_PWILU		
	ILU(0)	ILU(1)	ILU(2)	ILU(0)	ILU(1)	ILU(2)
$it_{impv}$	6	6	4	5	6	$\geq 5$
$ILU_{resid}$	30.5	10.5	498	79.7	470	\
GMRES	468	260	135	469	189	\
$GMRES_{time}$	3750 ms	2424 ms	1421 ms	3122 ms	1619 ms	\

point-wise ILU( $p$ ) ( $p \leq 2$ ) preconditioners on CPUs. We observe from Figure 5 and Table 4 that GPU\_PWILU has some advantage over GPU\_PBILU in terms of  $GMRES_{time}$  and  $ILU_{time}$  using ILU(0) and ILU(1), but GPU\_PBILU is more stable than GPU\_PWILU when ILU(2) is used because a number of sweeps of GPU\_PWILU can’t provide an effective preconditioner for GMRES to converge.

For the  $200 \times 200$  mesh, 486, 260 and 169 iterations are needed for GMRES(30) to converge using the exact point-block ILU(0), ILU(1) and ILU(2) on CPUs, respectively. It is shown in Table 5 that 6 sweeps of GPU\_PBILU are needed to make the inexact ILU(0) and ILU(1) on GPU comparable with exact ones on CPUs. Similar to the results of ILU(2) for the  $150 \times 150$  mesh, 34 GMRES iterations can be saved when 4 sweeps are employed for ILU(2) on GPU. And the present method performs about  $2.64\times$ ,  $2.48\times$  and  $3.66\times$  faster than the function in the cuSPARSE library. Employing the exact point-wise ILU(0), ILU(1) and ILU(2), GMRES uses 471, 262 and 170 iterations to converge, respectively. Figure 6 and Table 5 show the detailed comparisons. We also see the advantage of GPU\_PWILU with ILU(0), but it takes almost the same time for GPU\_PWILU and GPU\_PBILU to perform 6 sweeps. Since ILU(2) for GPU\_PWILU results in no convergence, we see GPU\_PBILU shows more stability than GPU\_PWILU with increasing fill-in levels.

We then increase the mesh size to  $300 \times 300$ . The point-block matrix contains 448800 and 627602 non-zero blocks for ILU(0) and ILU(1) patterns whereas the point-wise matrix contains 1871424 and 3289466 non-zero values for point-wise ILU(0) and ILU(1) patterns. On CPUs, when the exact point-block ILU(0) and ILU(1) are applied, it takes GMRES(30) 768 and 407 iterations to converge to  $10^{-3}$ , whereas GMRES(30) uses 757 and 410 iterations to converge to the same tolerance using exact point-wise ILU(0) and ILU(1). We compare GPU\_PBILU to all point-wise versions where zero values are ignored in blocks as well as the block-based version provided by the cuSPARSE library in Figure 7. The detailed comparison between GPU\_PBILU and GPU\_PWILU is shown in Table 6. Note that since neither GPU\_PBILU nor GPU\_PWILU with ILU(2) makes GMRES converge, we only report the results using ILU(0) and ILU(1). With point-block ILU(0) obtained by 4 sweeps, GMRES(30) uses 19 fewer iterations to converge to  $10^{-3}$  by using inexact factorizations on GPUs than using exact factorizations on CPUs. Compare



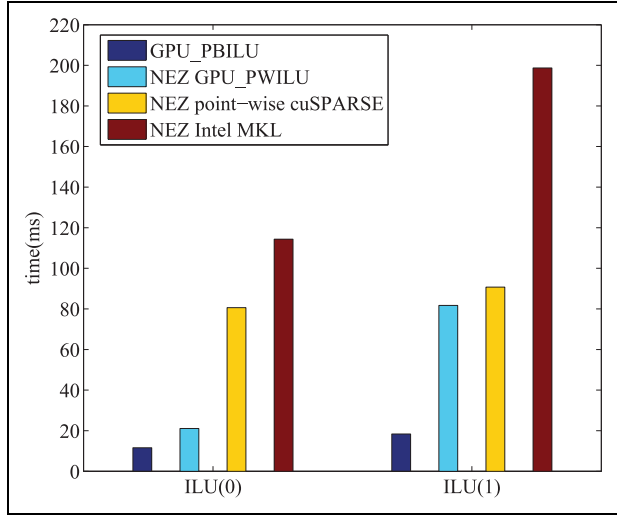
**Figure 7.**  $300 \times 300$  mesh: comparisons among GPU\_PBILU, GPU\_PWILU and popular libraries.

**Table 6.**  $it_{impv}$ ,  $ILU_{resid}$ , GMRES, and  $GMRES_{time}$  for GPU\_PBILU and GPU\_PWILU on  $300 \times 300$  mesh.

Fill-in levels	GPU_PBILU		GPU_PWILU	
	ILU(0)	ILU(1)	ILU(0)	ILU(1)
$it_{impv}$	4	4	5	6
$ILU_{resid}$	527	68	123	754
GMRES	749	407	752	386
$GMRES_{time}$	13825 ms	8543 ms	11276 ms	6818 ms

to the exact point-wise ILU(0), 5 sweeps for point-wise ILU(0) on GPU make GMRES(30) converge with 5 fewer iterations. Note that GPU\_PWILU performs better than GPU\_PBILU in terms of the compute time and  $GMRES_{time}$ . However, GPU\_PWILU fails to be the better one when we switch to ILU(1). No more sweeps are required for point-block ILU(1) to make GMRES converge within 407 steps. In contrast, the point-wise ILU(0) obtained by 5 sweeps of GPU\_PWILU can't make GMRES(30) converge within 410 steps and one more step is required. The extra step even makes the GPU\_PWILU better than the exact one and improves the convergence of GMRES much. Although GPU\_PWILU results in less compute time for GMRES, we still observe GPU\_PBILU with much more constraints to be solved performs better than GPU\_PWILU in terms of the factorization time, which is attractive for conducting ILU factorizations in parallel.  $3.0\times$  and  $2.77\times$  speedups can be obtained by GPU\_PBILU over the block-based implementation in the cuSPARSE library.

Furthermore, we consider another point-wise version by not eliminating zero values (NEZ) in blocks. A matrix in BCSR firstly transformed into CSR by counting all values in blocks, and then factorized by GPU\_PWILU and subroutines in MKL and cuSPARSE libraries. On CPUs, both the point-block and point-wise NEZ versions of ILU result in the



**Figure 8.**  $300 \times 300$  mesh: comparisons among GPU\_PBILU, GPU\_PWILU and popular libraries by not eliminating zeros (NEZ) in blocks.

same number of GMRES iterations. The number of iterations is 768 using ILU(0) and 407 using ILU(1), respectively. The clock times of factorizations are shown in Figure 8. For ILU(0) on GPU, both the times of GPU\_PBILU and GPU\_PWILU are obtained by 4 sweeps. GPU\_PBILU results in 749 GMRES iterations while GPU\_PWILU results in 747 GMRES iterations. But GPU\_PBILU is  $1.82\times$  faster than GPU\_PWILU. For ILU(1) on GPU, 4 sweeps are still sufficient for GPU\_PBILU to make GMRES converge. The number of iterations is 407. The number of sweeps is tested from 4 to 7 for GPU\_PWILU, but none of them provides an effective preconditioner to make GMRES converge. The increasing non-zero patterns that are initialized by zeros might cause GPU\_PWILU unstable with increasing fill-in levels in this case. We still list the time of GPU\_PWILU obtained by 7 steps in Figure 8 even though that bar is meaningless. We observe that GPU\_PBILU achieves  $6.96\times$  and  $4.93\times$  speedup over cuSPARSE for ILU(0) and ILU(1), which demonstrates that our algorithm is more stable to maintain good performance with increasing fill-in levels compared to GPU\_PWILU.

### A problem from a sixth-order crystal growth problem

In this experiment, we consider a  $99372 \times 99372$  matrix from the phase field method for solving a sixth order partial differential equation for the modeling of crystal growth. As reported in Yang and Cai (2014) and Yang et al. (2013) standard GMRES preconditioned with point-wise ILU( $p$ ) doesn't work for the system. In this problem, each mesh point has three variables  $(v_1, v_2, v_3)$ , standard point-wise approaches arrange the variables as  $[v_1^{(1)}, v_1^{(2)}, \dots, v_1^{(n)}, v_2^{(1)}, v_2^{(2)}, \dots, v_2^{(n)}, v_3^{(1)}, v_3^{(2)}, \dots, v_3^{(n)}]^T$  where  $v_i^{(j)}$  represents the  $i^{th}$  variable of the  $j^{th}$  point. In the point-block approach, we order the variables as  $[v_1^{(1)}, v_2^{(1)}, v_3^{(1)}, v_1^{(2)}, v_2^{(2)}, v_3^{(2)}, \dots, v_1^{(n)}, v_2^{(n)}, v_3^{(n)}]^T$ , the resulting matrix is a point-block matrix

**Table 7.** The crystal growth problem: comparisons to cuSPARSE, Intel MKL libraries and GPU\_PWILU with higher fill-in levels. The tolerance of GMRES(30) is set to  $10^{-3}$ . “\” means no convergence.

cuSPARSE (point-wise): with level-scheduling		
Time	ILU(0)	ILU(1)
$T_{buffer\ size}$	0.15 ms	0.17 ms
$T_{analysis}$	13.69 ms	16.07 ns
$T_{csrilu}$	12.13 ms	18.34 ms
cuSPARSE (point-wise): without level-scheduling		
Time	ILU(0)	ILU(1)
$T_{buffer\ size}$	0.15 ms	0.17 ms
$T_{analysis}$	0.42 ms	0.46 ms
$T_{csrilu}$	90.38 ms	106.32 ms
cuSPARSE (block-based): with level-scheduling		
Time	ILU(0)	ILU(1)
$T_{buffer\ size}$	0.14 ms	0.12 ms
$T_{analysis}$	2.84 ms	3.04 ms
$T_{csrilu}$	17.05 ms	26.15 ms
cuSPARSE (block-based): without level-scheduling		
Time	ILU(0)	ILU(1)
$T_{buffer\ size}$	0.14 ms	0.13 ms
$T_{analysis}$	0.23 ms	0.23 ms
$T_{csrilu}$	175.10 ms	201.20 ms
Intel MKL		
Time	ILU(0)	ILU(1)
$T_{dcsrilu}$	39.45 ms	99.52 ms
GPU_PWILU		
$i_{t_{impv}} = 5$	ILU(0)	ILU(1)
GMRES	\	\
$ILU_{time}$	11.20 ms	32.01 ms
$ILU_{resid}$	\	\
GPU_PBILU		
$i_{t_{impv}} = 5$	ILU(0)	ILU(1)
GMRES	15	6
$ILU_{time}$	8.55 ms	11.79 ms
$ILU_{resid}$	0.0193	13.7

with  $3 \times 3$  blocks. GMRES(30) takes 15 and 6 iterations to converge with the tolerance  $10^{-3}$  on the CPU when ILU(0) and ILU(1) are applied, respectively. It is shown in Table 7 that GPU\_PBILU with 5 sweeps makes GMRES(30) converge with the same number of iterations and tolerance. To show the advantage of our algorithm over point-wise ILU factorizations, we show the time for the factorizations obtained by point-wise versions from cuSPARSE, Intel MKL and GPU\_PWILU in Table 7. Our algorithm is faster than all point-wise versions and achieves  $2.34\times$  and  $2.48\times$  speedup over the block-based version from cuSPARSE. Then we consider a tighter tolerance of  $10^{-10}$  for GMRES.

On the CPU, 59 and 19 GMRES iterations are required to converge for the exact point-block ILU(0) and ILU(1), respectively. For the inexact ILU(0) on GPU, 5 sweeps are needed for GPU\_PBILU to converge in 59 iterations. But for the inexact ILU(1), 6 sweeps are required to converge in 19 iterations, and the speedup reduces to  $2.07\times$ .

## Some concluding remarks

ILU( $p$ ) is a fundamental building block of many preconditioning techniques, such as domain decomposition methods in Kong and Cai (2016) and Luo et al. (2020), for solving linear system of equations, and is also one of the most difficult components to be parallelized on a GPU because it is originally designed for purely sequential computers. In this paper, an inexact ILU( $p$ ) preconditioner in the point-block form is investigated for a GPU. Note that in this paper a “point” block matrix is viewed differently than a standard block matrix in the sense that the size of the block has to be small enough such that the inverse of the block can be written out explicitly. The proposed algorithm is effective for solving linear system of equations arising from the discretization of multicomponent partial differential equations for which the resulting matrix can be arranged in a point-block format. Our experiments show that the point-block ILU is more stable than the point-wise ILU in the sense that the point-wise preconditioned GMRES may fail to converge while the point-block ILU converges because additional coupling between variables associated with the same mesh point is preserved. Even though the point-block version involves more arithmetic operations than the corresponding point-wise version, it could be faster on GPU because of the inherited data locality in the point-block matrix and the proposed fine-grained algorithm for block-block matrix multiplication and block inversion. Our experiments also show that  $2\times$ – $5\times$  speedups can be obtained over the block-based ILU factorizations provided by the cuSPARSE library. In all experiments, we discuss the speedups between the approximate ILU and the exact ILU under the condition that the same number of GMRES iterations is performed to solve the problem, which ensures that the comparisons are fair. The success is due to two key ingredients, namely (1) the point-block format that keeps more coupling of variables associated with a point in the mesh and (2) the asynchronous iterations that enable the efficient use of GPU. Our future work includes the extension of the current approach to a cluster of GPUs.

## Acknowledgment

We thank Yingzhi Liu for his help.


## Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

## Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work of the first author is supported in part by NSFC 61702438, the Nanhu Scholar Program of XYNU and the Innovation Team Support Plan of Science and Technology of Henan Province (19IRTSTHN014).

## ORCID iD

Xiao-Chuan Cai  <https://orcid.org/0000-0003-0296-8640>

## References

- Abdelfattah A, Haidar A, Tomov S, et al. (2016a) Fast Cholesky factorization on GPUs for batch and native modes in MAGMA. *Journal of Computational Science* 20: 85–93.
- Abdelfattah A, Ltaief H, Keyes D, et al. (2016b) Performance optimization of sparse matrix-vector multiplication for multi-component PDE-based applications using GPUs. *Concurrency & Computation Practice & Experience* 28(12): 3447–3465.
- Anderson E and Saad Y (1989) Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing* 1(1): 73–95.
- Anzt H, Gates M, Dongarra J, et al. (2017) Preconditioned Krylov solvers on GPUs. *Parallel Computing* 68: 32–44.
- Axelsson O, Eijkhout V, Polman B, et al. (1989) Incomplete block-matrix factorization iterative methods for convection-diffusion problems. *BIT Numerical Mathematics* 29(4): 867–889.
- Balay S, Abhyankar S, Adams MF, et al. (2020) PETSc web page. Available at: <https://www.mcs.anl.gov/petsc> (accessed 25 October 2020).
- Chen Y, Tian X, Liu H, et al. (2018) Parallel ILU preconditioners in GPU computation. *Soft Computing* 22: 8187–8205.
- Chow E and Patel A (2015) Fine-grained parallel incomplete LU factorization. *SIAM Journal on Scientific Computing* 37(2): C169–C193.
- Chow E, Anzt H and Dongarra H (2015) Asynchronous iterative algorithm for computing incomplete factorizations on GPUs. In: Kunkel JM and Ludwig T (eds) *ISC HIGH PERFORMANCE 2015, Lecture notes in computer science*, Vol. 9137, pp. 1–16, 2015. DOI: 10.1007/978-3-319-20119-1\_1
- Davis TA and Hu Y (2011) The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software* 38(1): 1–25.
- Eberhardt R and Hoemmen M (2016) Optimization of block sparse matrix-vector multiplication on shared-memory parallel architectures. In: *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, Chicago, IL, USA, 23–27 May 2016, pp. 663–672 USA: IEEE.
- Intel Math Kernel Library Documentation (2017) Available at: <https://software.intel.com/en-us/articles/intel-math-kernel-library-documentation> (accessed 30 October 2020).
- Kim SW and Yun JH (2000) Block ILU factorization preconditioners for a block-tridiagonal H matrix. *Linear Algebra and its Applications* 37(1–3): 103–125.
- Kong F and Cai XC (2016) Scalability study of an implicit solver for coupled fluid-structure interaction problems on unstructured meshes in 3D. *The International Journal of High Performance Computing Applications* 32: 207–219.
- Li RP and Saad Y (2013) GPU-accelerated preconditioned iterative linear solvers. *Journal of Supercomputing* 63(2): 443–466.
- Luo L, Liu L, Cai Y, et al. (2020) Fully implicit hybrid two-level domain decomposition algorithms for two-phase flows in porous media on 3D unstructured grids. *Journal of Computational Physics* 409: 109312.
- Luo LX, Edwards JR, Luo H, et al. (2015a) A fine-grained block ILU scheme on regular structures for GPGPUs. *Computers & Fluids* 119: 149–161.
- Luo LX, Edwards JR, Luo H, et al. (2015b) Optimization of a fine-grained BILU by CUDA inter-block synchronization. In: *22nd AIAA computational fluid dynamics conference*, Dallas, TX, 22–26 June 2015, pp. 1–17(2015–3055) USA: AIAA.
- Nguyen Loc Q (2017) Quick start guide for Intel Xeon Phi processor x200 product family. Available at: <https://software.intel.com/en-us/articles/quick-start-guide-for-the-intel-xeon-phi-processor-x200-product-family> (accessed 1 November 2020).
- NVIDIA cuSPARSE library (2014) Available at: <https://developer.nvidia.com/cuda-toolkit-65> (accessed 10 March 2020).
- Pakzad M, Lloyd JL and Phillips C (1997) Independent columns: a new parallel ILU preconditioner for the PCG method. *Parallel Computing* 23(6): 637–647.
- Poole EL and Ortega JM (1987) Multicolor ICCG methods for vector computers. *SIAM Journal on Numerical Analysis* 24(6): 1394–1418.
- Rennich SC, Stosic D and Davis TA (2016) Accelerating sparse Cholesky factorization on GPUs. *Parallel Computing* 59: 140–150.
- Rupp K, Tillet PH, Rudolf F, et al. (2016) ViennaCL—linear algebra library for multi- and many-core architectures. *SIAM Journal on Scientific Computing* 38: S412–S439.
- Saad Y (2003) *Iterative Methods for Sparse Linear Systems*, 2nd edn. Philadelphia, PA: Society for Industrial and Applied Mathematics.
- Saad Y and Zhang J (1999a) BILUM: block versions of multi-elimination and multilevel ILU preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing* 20(6): 2103–2121.
- Saad Y and Zhang J (1999b) BILUTM: a domain-based multilevel block ILUT preconditioner for general sparse matrices. *SIAM Journal on Matrix Analysis and Applications* 21(1): 279–299.
- Yang B and Liu H (2015) Accelerating the GMRES solver with block ILU(k) preconditioner on GPUs in reservoir simulation. *Journal of Geology & Geophysics* 4(2): 1–7.
- Yang C and Cai XC (2014) A scalable implicit solver for phase field crystal simulations. In: *27th IEEE international parallel & distributed processing symposium workshops & PhD forum (IPDPSW'13)*, IEEE: Boston, MA, USA.
- Yang C, Cai XC, Keyes DE, et al. (2013) NKS method for the implicit solution of a coupled Allen-Cahn/Cahn-Hilliard

system. In: *Lecture notes in computational science and engineering*, Vol. 98, pp. 819–827.

Yun JH (2000) Block ILU preconditioners for a nonsymmetric block-tridiagonal M-matrix. *BIT Numerical Mathematics* 40(3): 583–605.

### Author biographies

*Wenpeng Ma* is a lecturer at School of Computer and Information Technology, Xinyang Normal University, China. He received a Master's degree from Shenzhen University in 2011 and PhD from Computer Network and

Information Center, Chinese Academy of Sciences in 2015. His research interests include GPU computing, heterogeneous computing and parallel software for scientific applications.

*Xiao-Chuan Cai* is a Professor at University of Macau. He received PhD and MSc from New York University in 1989 and 1988, and BSc from Peking University in 1984. His research interests include domain decomposition methods for partial differential equations and high performance scientific computing.