DBSR: An Efficient Storage Format for Vectorizing Sparse Triangular Solvers on Structured Grids

Xiaojian Yang[1,§], Shengguo Li[1,§], Fan Yuan[2], Dezun Dong[1,*]
[1]*National University of Defense Technology,* China
[2]*Xiangtan University,* China
{yangxj, nudtlsg, dong}@nudt.edu.cn, fyuan@smail.xtu.edu.cn

***Abstract*—The Sparse Triangular Solver (SPTRSV) plays a critical role in solving structured grid problems. Yet, the commonly used sparse matrix storage formats for structured grid methods do not efficiently support SPTRSV in utilizing the instruction parallelism offered by modern multi-core CPUs. We introduce DBSR, a new sparse storage format to enable SPTRSV to take advantage of the SIMD instructions. DBSR promotes contiguous memory access and vectorized computation, while also optimizing memory usage. We evaluate DBSR by applying it within multigrid algorithms and the zero fill-in incomplete LU preconditioner. Our evaluation, conducted on four architectures – three ARMv8 systems and one x86 system – demonstrates that DBSR consistently outperforms mainstreamed storage formats across evaluation workloads and platforms.**

***Index Terms*—Structured grids, Sparse triangular solver, Vectorization**

I. Introduction

Sparse linear solvers are typically applied to find solutions from structured grids which are widely used in computational problems. Examples of such problems include electromagnetism [1], astrophysics [2], radiative transfer [3] and atmospheric modeling [4]. Structured grids are characterized by their evenly spaced divisions, creating a uniform pattern of squares, rectangles, or cubes. This organization simplifies data management and algorithm design, lowering memory use and improving computation efficiency.

Sparse linear solvers, especially iterative methods, can be adapted to the specific needs of structured grids to provide accurate solutions. A key component in an iterative-based sparse linear solver and its associated components is the Sparse Triangular Solver (SPTRSV). This method is used in key algorithms like Gauss-Seidel (GS) and incomplete LU (ILU). A properly designed and well-optimized SPTRSV can greatly improve the computation of the linear solver [5], [6]. It also plays a crucial role in smoothing operations and coarse-grid corrections, enhancing the robustness and speed of multi-grid (MG) methods with linear solvers. Therefore, the efficiency of SPTRSV directly impacts the overall performance of these solvers, making fast SPTRSVs crucial.

Parallelization is key to achieving good performance on multi-core processors. Unfortunately, efficient parallelization of SPTRSV can be more challenging compared to many other heavily-studied sparse linear algebra subroutines [7], [8] like the sparse matrix-vector multiplication (SPMV) [9], [10] and sparse matrix-matrix multiplication (SPGEMM) [11]. While the elements of the input, x, in SPTRSV can be computed in parallel, there often exist non-trivial data dependencies among matrix elements.

Although several techniques were proposed to parallelize SPTRSV [12]–[15], most of the methods do not capitalize the SIMD instructions of modern processors [16]–[18]. Some limited work has attempted to leverage SIMD instructions on the Xeon Phi processor for the symmetric Gauss-Seidel (SYMGS) smoother [19]. This technique is based on the sliced ELLPACK (SELL) storage format [20]. As we will show later in this paper (Section V), the sparse matrix storage format plays a key role in the parallelization efficiency of SPTRSV, but mainstreamed sparse matrix storage formats, including SELL, are inadaquate for the problem.

This paper investigates ways to allow SPTRSV to take advantage of SIMD instructions of modern multi-core CPUs through instruction vectorization. To this end, we propose a new storage format for the SPTRSV input. Our storage format, namely *diagonal block compressed sparse row* (DBSR), is specifically tailored for SPTRSV and structured grids. It is a combination of the block compressed sparse row (BCSR) [21] format and diagonal (DIA) format [6], where each block is stored in the DIA format. This customized data structure reduces the overhead of storing column indices, since it only needs to record the position of each block. By reusing an index, the location of other data within the block can be calculated. At the same time, it facilitates contiguous memory accesses, reducing the number of indirect addresses and thus reducing the high cost introduced by other storage formats. This, in turn, improves the cache locality and the application performance. DBSR adopts a reordering technique to improve instruction parallelism while maintaining a convergence rate for the algorithm. As we will show later in the paper, DBSR is more efficient than mainstreamed sparse storage like SELL and SELL-C-σ specifcially designed for vectorization [10].

We demonstrate the benefit of DBSR by integrating it into geometric multigrid (GMG) algorithms and the High Performance Conjugate Gradient (HPCG) benchmark [22], [23]. HPCG provides a representative implementation of structured grid problems and is commonly used to evaluate the performance of high-performance systems for real-world applications. We showcase that DBSR complements state-of-

[§]Equal contribution.
[*]Corresponding author.

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/31.00 ©2024 IEEE

the-art optimization techniques [24], [25] to further enhance application performance through SIMD instructions and vectorization.

We evaluate our approach on four architectures, including one x86 and three ARMv8 platforms. Compared to other state-of-the-art techniques for optimizing MG algorithms [24], [25], DBSR has improved its performance by 1.19x$\sim$1.24x. When integrating DBSR with these techniques, our implementation of GMG and HPCG achieves a speedup between 1.47x to 3.40x over the engineer- and vendor-tuned HPCG implementations on x86 and ARM platforms [26], [27]. Tests using a distributed setup with 256 nodes show that DBSR supports good weak scalability. We also integrate DBSR into the zero fill-in ILU (ILU(0)) preconditioner [6] to effectively support the implementation of a vectorized ILU(0) algorithm. ILU(0) is one of the most widely used preconditioners [28]. While there are limited works on vectorizing ILU(0) [29], [30], our DBSR-based vectorization version leads to better performance, delivering an average improvement of 1.17x to 1.31x in double-precision and 1.22x to 1.46x in single-precision across different platforms compared to other best strategies.

This paper makes the following contributions:

1) We introduce a new storage format based on reordering, DBSR, tailored for structured grid problems. DBSR supports contiguous memory access and minimizes memory overhead.
2) We demonstrate how DBSR can be applied to a variety of preconditioners or smoothers[1]. Multiple types of datasets have been evaluated on multiple platforms.

II. Background

A. Sparse Matrix Storage Format

Sparse matrices typically only contain a few non-zero elements per row, and only these elements need to be stored. Many excellent storage formats have been adopted that can greatly improve access efficiency. For example, the Coordinate (COO) format is the default storage format for `.mtx` text and can be quickly converted to other formats. The Compressed Sparse Row (CSR) format is the simplest and most commonly used sparse format, universal and efficient. The Sliced ELLPACK (SELL) format divides the matrix into segments by rows [20]. Each segment is compressed to the left and stored in columns, with zeros added to the shortfall. The SELL format is commonly used for vectorized calculations. Diagonal (DIA) format applies to sparse matrices where the distribution of non-zero elements is parallel to the diagonal. The Block Compressed Sparse Row (BCSR) format applies to sparse matrices with dense submatrices and is more efficient in some cases (see [31] for a more detailed illustration). Fig. 1 shows an example implementation of these storage formats. Due to the irregular distribution of non-zero elements of sparse matrices, the choice of storage format has a direct impact on the computational efficiency. More efficient storage formats

[1]Code and data available at https://github.com/YXJ-123/DBSR.

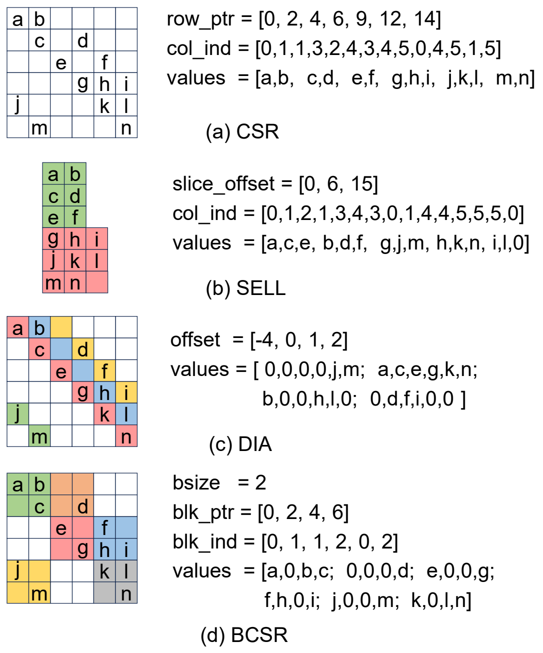

row_ptr = [0, 2, 4, 6, 9, 12, 14]
col_ind = [0,1,1,3,2,4,3,4,5,0,4,5,1,5]
values = [a,b, c,d, e,f, g,h,i, j,k,l, m,n]
(a) CSR
slice_offset = [0, 6, 15]
col_ind = [0,1,2,1,3,4,3,0,1,4,4,5,5,5,0]
values = [a,c,e, b,d,f, g,j,m, h,k,n, i,l,0]
(b) SELL
offset = [-4, 0, 1, 2]
values = [0,0,0,0,j,m; a,c,e,g,k,n; b,0,0,h,l,0; 0,d,f,i,0,0]
(c) DIA
bsize = 2
blk_ptr = [0, 2, 4, 6]
blk_ind = [0, 1, 1, 2, 0, 2]
values = [a,0,b,c; 0,0,0,d; e,0,0,g; f,h,0,i; j,0,0,m; k,0,l,n]
(d) BCSR

Fig. 1. Several efficient sparse matrix storage formats.

Algorithm 1: Solving SPTRSV in CSR format.

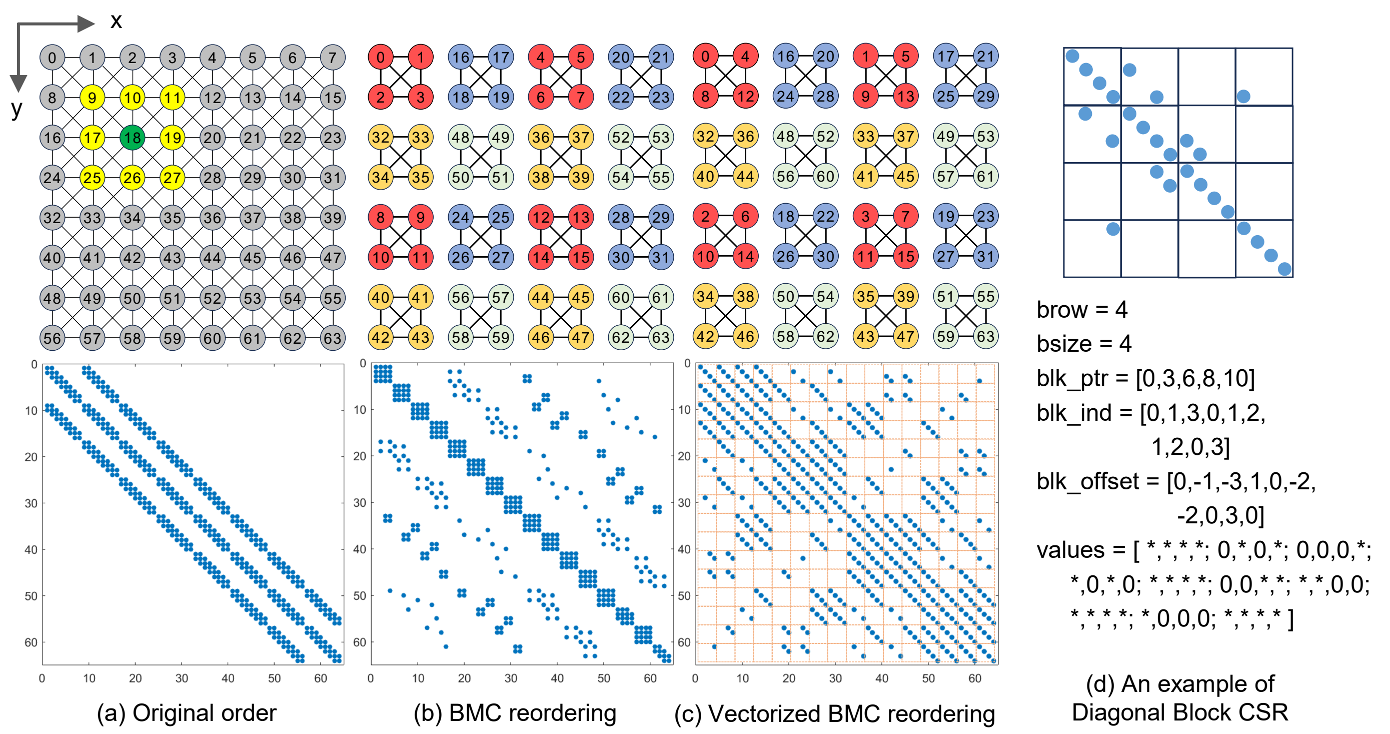
```
Input: $L, b, x$                    ▷ $L$ is the lower triangular matrix.
Output: $x$
1 for $i = 0$ to $n - 1$ do
2     double $temp = b[i]$
3     for $j = row\_ptr[i]$ to $row\_ptr[i+1] - 1$ do
4         $temp -= values[j] * x[col\_ind[j]]$
5     end
6     $x[i] = x[i] + temp/D[i]$        ▷ $D[i]$ is the $i$-th diagonal.
7 end
```

are constantly being created to accommodate a wide variety of computing environments.

B. Block Multi-Color Ordering

Fig. 2(a) shows a two-dimensional 8x8 structured grid of 9-point stencil discretized, where the numbers represent the order of serial processing. For example, the point 18 is connected to 8 other points surrounding it. The figure also shows the adjacency matrix of this grid. Algorithm 1 outlines the procedure for solving a lower triangular linear system, $Lx = b$. For such matrices, the algorithm has a strict order of dependence and low parallelism.

Altering the processing sequence of grid points is a common strategy to enhance parallelism in sparse computations. The multi-color (MC) ordering technique [6] sacrifices some of the convergence rate to improve parallelism. This method categorizes points into colors where points of the same color are independent and can be calculated concurrently. Points with lower color priority rely on neighbors with higher priority. However, this coloring with points as the basic unit is difficult to control. Too few colors can cause more of the original dependencies to be broken, leading to a decrease in the


x
y
(a) Original order
(b) BMC reordering
(c) Vectorized BMC reordering
(d) An example of Diagonal Block CSR

```
brow = 4
bsize = 4
blk_ptr = [0,3,6,8,10]
blk_ind = [0,1,3,0,1,2,
           1,2,0,3]
blk_offset = [0,-1,-3,1,0,-2,
              -2,0,3,0]
values = [ *,*,*,*; 0,*,0,*; 0,0,0,*;
   *,0,*,0; *,*,*,*; 0,0,*,*; *,*,0,0;
   *,*,*,*; *,0,0,0; *,*,*,* ]
```

Fig. 2. The evolution of our vectorized BMC reordering. Fig. (a) displays the processing order of the original structured grid and the layout of the sparse matrix, with each number denoting the processing order. Fig. (b) shows the classical BMC reordering process. Fig. (c) illustrates our vectorized BMC reordering with vector length 4. The sorted sparse matrix is partitioned based on the vector length. Within each data block, the non-zero elements are arranged along the parallel lines of diagonal. In Fig. (d), we extract multiple data blocks to introduce the diagonal block CSR (DBSR) storage format.

convergence rate, while too many colors can lead to poor locality and excessive synchronization time between colors.

Block multi-color ordering [15], on the other hand, divides the grid into multiple blocks and applies multi-color to the blocks. The classical BMC ordering process is shown in Fig. 2(b). The BMC method processes the grid blocks in color priority order while ensuring that the points within each grid block are processed sequentially, thus improving data locality and convergence. Grid blocks of the same color can be processed in parallel, so each thread can process one or more grid blocks.

In addition to the number of colors, the size of blocks is another critical variable in the BMC method. Adjusting the block size can help mitigate problems caused by the number of colors.

C. Optimization of HPCG

HPCG serves as a benchmark for solving large sparse linear systems, derived from the second-order 27-point stencil discretization of a 3D Poisson equation [22], [23]. It employs the preconditioned conjugate gradient (PCG) method to solve the linear system, with a four-level V-cycle geometric multigrid method serving as the preconditioner and SYMGS utilized as the smoother. Various techniques have been proposed for optimizing HPCG on different platforms [19], [32], [33].

On multi-core CPUs, the most recent state-of-the-art version, as proposed in [24], introduces methods to enhance the computation and parallelization efficiency of the SYMGS and MG algorithms. These methods include optimizing SYMGS to reduce both computation and memory access, as well as employing deep kernel fusion and adaptive block partitioning strategies for BMC. However, it uses the CSR format, which is difficult to vectorize. In this work, we combine these novel optimization strategies and propose a vectorization-friendly approach. The performance of HPCG is significantly further improved, and numerical results are shown in section V-B.

III. Reordering Optimization for Structured Grids

A. Reordering for Vectorization

Based on the fact that grid blocks of the same color perform the same operation, we vectorize the BMC method to take advantage of its parallelism. As shown in Fig. 2(c), we vectorize the grid points occupying the same position in multiple grid blocks of the same color. Here we set the vector length to 4. Then, in four grid blocks of the same color, the points at the same position are numbered consecutively. All grid points are repositioned according to this criterion. Note that the color priority will remains the same.

Fig. 2(c) also shows the structure of sparse matrix after our vectorized BMC reordering. The matrix is segmented by vector length, resulting in blocks where non-zero elements are aligned only along the main diagonal or lines parallel to it. This pattern stems from the fact that for points numbered consecutively in multiple grid blocks, the numbering of their neighbors in the same direction is also consecutive. However, due to grid boundaries, some points will exhibit different neighboring states, resulting in elements within their corresponding matrix blocks being distributed with offsets along the diagonal. For example, the points $\{40, 41, 42, 43\}$ are adjacent to the points $\{0, 1, 2, 3\}$ with only two points. In contrast, the matrix elements corresponding to the interior points within the grid are distributed along the diagonal of the matrix block without offset. For example, the points $\{32, 33, 34, 35\}$ are adjacent to all four points of the grid points $\{8, 9, 10, 11\}$.

B. Diagonal Block Compressed Sparse Row

Based on the characteristics of such sparse matrix element distribution, we have developed a novel storage format, diagonal block compressed sparse row (DBSR). In Fig. 2(d), we extract some data blocks to demonstrate our DBSR format. The storage information includes: (1) `brow` and `bsize`: denotes the number of rows and the block size, respectively, where $bsize$ is equal to the vector length; (2) `*blk_ptr`: marks the offset of the starting block in each row and the total block count, with $brow + 1$ entries; (3) `*blk_ind`: records the column index of each data block and the amount is num_Blocks, where num_Blocks is the total number of data blocks that satisfy $num\_Blocks = blk\_ptr[brow]$; (4) `*blk_offset`: captures the intra-block diagonal offset for each block, again of length num_Blocks; and (5) `*values`: stores the matrix elements of length $num\_Block * bsize$. DBSR is similar to BCSR in that it stores only blocks containing non-zero elements. A notable difference is that each data block is stored using the DIA format, which retains only a single diagonal rather than the entire block. This therefore requires an additional array, equivalent in size to the column index array, to capture the diagonal offsets. There are two advantages to storing only one diagonal line within each data block. First, uniform element counts across data blocks set to $bsize$ facilitate efficient data retrieval. In addition, both the right-hand-side array b aligned with rows and the left-hand-side array x correlated with columns of non-zero elements can be accessed seamlessly in contiguous $bsize$ space. This breaks the shackles of discontinuous memory access in sparse arithmetic and greatly improves data locality. Once the vector length is determined, multiple vectors can be processed simultaneously if the number of parallelizable blocks per color is sufficient. Note that $bsize$ is not limited by the length of SIMD-bits supported by the hardware platform, and SIMD commands can be executed multiple times per block. When $bsize = 1$, our vectorized BMC will be converted to a classic BMC.

Our DBSR format reduces row index storage to $1/bsize$ and column index storage to $2/bsize$ compared to the commonly used CSR format. Although the value array will hold extra zeros, the padding will be much less than the index reduction if $bsize$ is chosen appropriately. In fact, the range of diagonal offsets within each block is determined entirely by the block size, so `*blk_offset` can be expressed using only $\log_2 bsize$-bits integer and 1-bit symbol, without the need for the `int` type. Then the memory overhead can be further minimized.

Algorithm 2: Solving SPTRSV in DBSR format.

```
Input: $L, b, x$                ▷ $L$ is the lower triangular matrix.
Output: $x$
1  // for $color$ from 0 to $NumColors - 1$
2  // #pragma omp parallel for
3  // for $t$ from $BlocksNum[color]$ to $BlocksNum[color + 1] - 1$
4  for $i = t * BlockSize$ to $(t + 1) * BlockSize - 1$ do
5      vdouble $vec\_temp = load(b + i * bsize)$
6      for $j = blk\_ptr[i]$ to $blk\_ptr[i + 1] - 1$ do
7          int $ind\_val = j * bsize$
8          int $ind\_x = blk\_ind[j] * bsize + blk\_offset[j]$
9          vdouble $vec\_vals = load(values + ind\_val)$
10         vdouble $vec\_x = load(x + ind\_x)$
11         $vec\_temp- = vec\_vals * vec\_x$
12     end
13     $vec\_store(x + i * bsize, vec\_temp)$
14 end
```

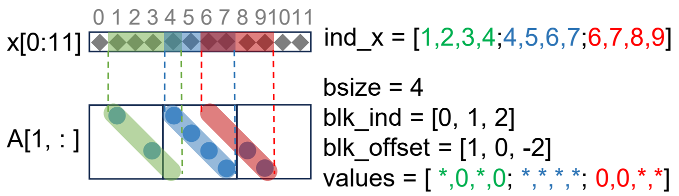


Fig. 3. Contiguous memory accesses and intra-block offsets in DBSR format.

C. Algorithm Description

DBSR is suitable for solving structured grid problems using the BMC reordering strategy. The acceleration process of vectorized BMC consists of three steps: (1) determining the BMC scheduling scheme, (2) ordering the sparse matrices and constructing the storage structure, and (3) computing to solve the problem. First, determine $bsize$ to establish the reordering relationship. Then, construct the DBSR storage structure for the adjusted sparse matrix. Since the shift details are available, the sparse matrix needs to be traversed only once. Also, both the solution array x and the right-hand side array b must follow to the specific order. Once the DBSR storage structure has been determined, it can be used repeatedly for calculations.

Algorithm 2 demonstrates how a vectorized version of DBSR accelerates SPTRSV. Line 3 of the algorithm is a parallel processing of multiple vectors of the same color. Each vector in the algorithm is loaded and stored by accessing $bsize$ consecutive addresses (lines 5, 9, 10). The positions of the matrix elements involved in the update are looked up directly in line 7. Line 8 calculates the column index of the first element within the block. Note that the intra-block offset must be taken into account when determining the starting index of each data block. These offsets may cause the indexes of values loaded into vectors to exceed the boundaries of individual blocks when accessing the solution array x. For example, this scenario occurs in both the first and third blocks of Fig. 3. Since the quantity of stored values within each block matches those on the diagonal, the solution array being accessed spans two blocks. However, DBSR sets the values of the overstore to zero and the calculations they participate in will not change the stored values.

D. `Gather`*-Free*

It is well known that sparse operations are difficult to optimize due to their discontinuous memory access and low computational access ratio. When the non-zero elements of a sparse matrix are widely scattered, access to the solution array becomes highly irregular, resulting in poor data locality. Although the `gather` command has been developed in various SIMD instruction sets, it is difficult to achieve the desired results using vectorized sparse operations due to its high overhead. Our DBSR allows values to be taken consecutively from an array because the indexes of the rows and columns of the elements within each block are contiguous. Therefore, in line 10 of Algorithm 2, the `gather` command can be replaced with the `load` command. This advantage is illustrated in Fig. 3 when accessing the x-values corresponding to the three data blocks, all of which are contiguous and can perform vector operations without conflict.

Algorithm 2 describes the vectorized version of the DBSR computational framework. The serial loop version can still use one index to calculate the remaining $bsize - 1$ positions without access operations.

E. Applicability

DBSR, though inspired by existing BCSR and DIA formats, is specifically designed for solving sparse linear equations on structured grids. This new format leverages SIMD instructions and is suitable for a wide range of sparse triangular solvers, such as SYMGS and ILU. In contrast, BCSR introduces excessive zero-value padding for sparse operations, and DIA is unsuitable for solving sparse triangular systems in parallel. Moreover, DBSR outperforms mainstream sparse storage formats like SELL and SELL-c-σ [10] without additional format conversion overhead. Our reordering strategy and DBSR format are specifically designed for structured grids, and they are applicable to both equidistant and non-equidistant structured grids in 2D or 3D.

IV. Optimization for ILU Preconditioner

We emphasize that the DBSR technique can be applied to a variety of smoothing operators. In this study, we focus on the incomplete LU(0) preconditioner as a demonstration to illustrate the acceleration potential of DBSR in both the factorization phase and the smoothing phases of structured sparse matrices.

A. Zero Fill-in Incomplete LU (ILU(0))

ILU preconditioner is the approximate factorization of a non-singular sparse matrix A into the product of a sparse lower triangular matrix L and a sparse upper triangular matrix U, $A \approx LU$, where the certain positions are allowed to be non-zero or zero_fill only. ILU(0) is the simplest ILU

Algorithm 3: General ILU(0) Factorization.

```
1 for $i = 1$ to $n - 1$ do
2     for $k = 0$ to $i - 1$ and $(i, k) \in NZ(A)$ do
3         Compute $a_{ik} = a_{ik}/a_{kk}$
4         for $j = k + 1$ to $n - 1$ and $(i, j) \in NZ(A)$ do
5             Compute $a_{ij} := a_{ij} - a_{ik}a_{kj}$.
6         end
7     end
8 end
```

Algorithm 4: ILU(0) Factorization in DBSR format.

```
1 // for $color$ from 0 to $NumColors - 1$
2 // #pragma omp parallel for
3 // for $t$ from $BlocksNum[color]$ to $BlocksNum[color + 1] - 1$
4 for $i = t * BlockSize + 1$ to $(t + 1) * BlockSize - 1$ do
5     for $p = blk\_ptr[i]$ to $dia\_ptr[i] - 1$ do
6         int $k = blk\_ind[p]$
7         vdouble $vec\_A_{ik} = load(values + p * bsize)$
8         vdouble $vec\_A_{kk} =$
            $load(values + dia\_ptr[k] * bsize + blk\_offset[p])$
9         $vec\_A_{ik} = vec\_A_{ik}/vec\_A_{kk}$
10        while $r = dia\_ptr[k] + 1$ to $blk\_ptr[k + 1] - 1$ and
            $q = p + 1$ to $blk\_ptr[i + 1] - 1$ do
11            if $blk\_ind[r] == blk\_ind[q]$ and
                $blk\_offset[p] + blk\_offset[r] == blk\_offset[q]$
                then
12                vdouble $vec\_A_{ij} = load(values + q * bsize)$
13                vdouble $vec\_A_{kj} =$
                    $load(values + r * bsize + blk\_offset[p])$
14                $vec\_A_{ij} -= vec\_A_{ik} * vec\_A_{kj}$
15            end
16        end
17    end
18 end
```

preconditioner, with no extra padding beyond the non-zero element positions of the original sparse matrix. Compared to other types of ILU preconditioners, ILU(0) trades a cheaper preprocessing cost for a better convergence rate [6]. Algorithm 3 describes the factorization procedure for ILU(0). It is critical to determine whether the location to be updated is a non-zero element (lines 2 and 4). Each element below the diagonal is updated with the corresponding diagonal element in its column (line 3), and then this result is utilized to modify all elements to the right (lines 4 and 5). Since the positions that need to be modified are predetermined, there is no need to change the storage structure of this sparse matrix. After completing the ILU(0) factorization, the approximate solution is obtained through multiple iterations of solving the upper and lower triangular linear systems using iterative algorithms.

Similar to other factorization techniques, the general ILU(0) follows a strict serial processing order. The MC-ILU(0) (multi-color ILU(0)) approach is commonly used to improve parallelism. However, MC-ILU(0) tends to discard many non-zeros during the factorization process, which leads to a significant reduction in the convergence rate [34]. Our DBSR is based on

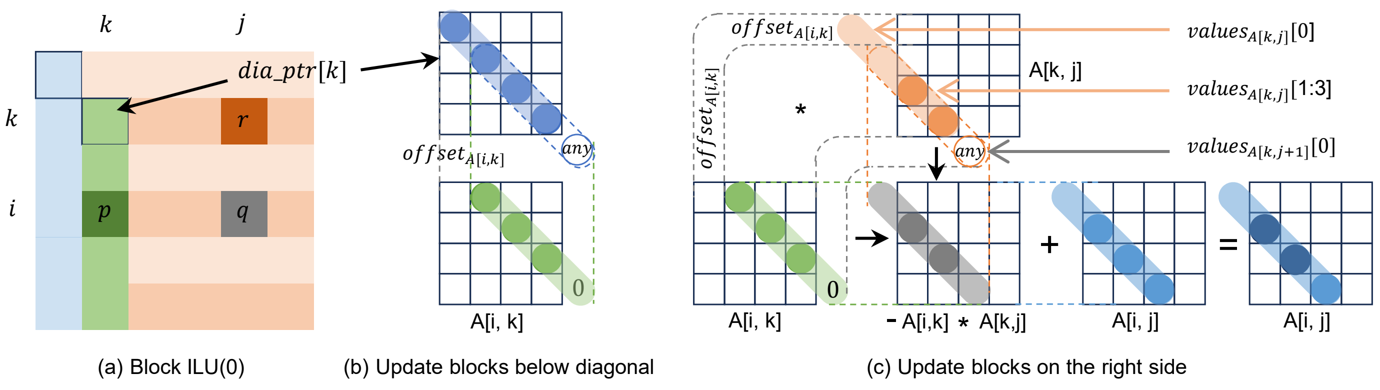


Fig. 4. The computational flow of the block ILU(0) factorization. Fig. (a) shows the positions of the data blocks. Fig. (b) updates data blocks below the diagonal. Fig. (c) updates the data blocks on the right side, which is a general sparse matrix block-block multiplication in DBSR format.

the BMC method and improves parallelism while maintaining a similar convergence rate as the original sequential preconditioner. Owing to its distinctive element distribution, ILU(0) preconditioner can be performed faster in the DBSR format.

B. Apply DBSR to ILU(0)

In the DBSR format, the smallest storage unit of the matrix is the block, so the ILU(0) factorization is converted to a block ILU(0) algorithm. This modification enhances the efficiency of updating the necessary data blocks. Fig. 4 depicts the computational flow of the block ILU(0) factorization in DBSR format. Algorithm 4 describes the corresponding pseudocode, where $*dia\_ptr$ indicating the position of the diagonal blocks in $*blk\_ind$ and $*blk\_offset$. Similarly, grid blocks of the same color can be factorized in parallel (line 3). The elements within each data block are diagonally distributed, so data updates occur only between data blocks, not within them. When updating the data blocks below the diagonal of the sparse matrix, the vector loading of the diagonal elements must take into account the offsets within the data blocks below (line 8). As in Fig. 4(b), the lower data block $A[i,k]$ is updated with the diagonal block. The offset $offset_{A[i,k]}$ causes part of the diagonal block's neighboring data to be loaded into the vector as interfering data. However, this does not affect the result because the corresponding padding value in block $A[i,k]$ is 0. This situation is analogous to Fig. 3.

To update the data blocks on the right side, first find the non-zero data blocks corresponding to these blocks in the upper triangular matrix (line 11). These two data blocks will perform the general sparse matrix-matrix multiplication (SPGEMM) operation. In this process, since ILU(0) does not add new zero elements, there is no need to perform the multiplication of these two data blocks if their result does not match the non-zero position in the target data block. In Fig. 4(c), we show an example about the multiplication of two data blocks. In DBSR format, SPGEMM can be simplified to a dot product of two vectors if the positions of the multiplied data are correctly aligned. To achieve this, ensure that the offsets of the two data blocks add up to the same offset as the target data block (line 11). The left data block $A[i,k]$ in

TABLE I
HARDWARE PLATFORMS USED IN EVALUATION.

	Intel Xeon	KP 920	Thunder X2	Phytium 2000+
Sockets	2	1	1	8
#Cores	2×28	1×64	1×32	1×64
#NUMAs	2	2	1	8
CPU Freq.	2.6GHz	2.6GHz	2.5GHz	2.2GHz
L1 Cache	80KB	64KB	32KB	32KB
L2 cache	1.25MB	512KB	256KB	2MB
L3 cache	42MB	64MB	32MB	None
SIMD-bits	AVX512-512	NEON-128	NEON-128	NEON-128

SPGEMM determines the row distribution of the result data. The vector loading of the right data block $A[k,j]$ must take into account the offset $offset_{A[i,k]}$ in the left data block in order for SPGEMM to work correctly (line 13). Likewise, the corresponding multiplier for the offset-induced interfering data in the left data block is exactly zero and therefore does not affect the update of the target data block (line 14).

After the LU factorization phase, the smoothing phase for L and U is similar to Algorithm 2. Therefore, the DBSR strategy is applicable in both the factorization and smoothing phases of ILU(0).

V. EVALUATION

A. Setup

We evaluate DBSR strategy on both an x86 and three representative ARMv8 multicore architectures. Table I lists the hardware platforms, including Intel Xeon Gold-6348 CPU, the KunPeng 920 (KP 920) [35], Thunder X2 [36] and Pyhtium 2000+ CPU [37]. All codes are compiled using `icpc` version 2021.8.0 on Intel platforms and GCC version 9.3.0 and MPICH version 3.4.3 on ARMv8 platforms. The `-O3` and `-march=native` options are used for each node.

The experiments focused on evaluating the impact of the DBSR strategy in the HPCG benchmark and ILU(0) factorization and smoothing. The performance improvement provided by DBSR was demonstrated in the HPCG experiments, where comparisons were made with the state-of-the-art work [24] and the optimized versions of the Intel MKL library (release date January 2024) [27] and ARM [38]. The experiments were run

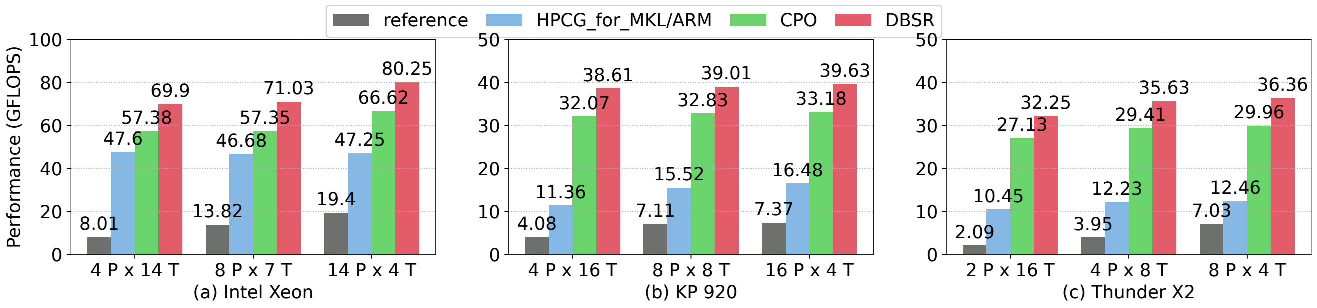


Fig. 5. Overall results of HPCG with different computational allocation schemes when a single node is fully utilized (P: MPI Processes, T: OpenMP Threads).

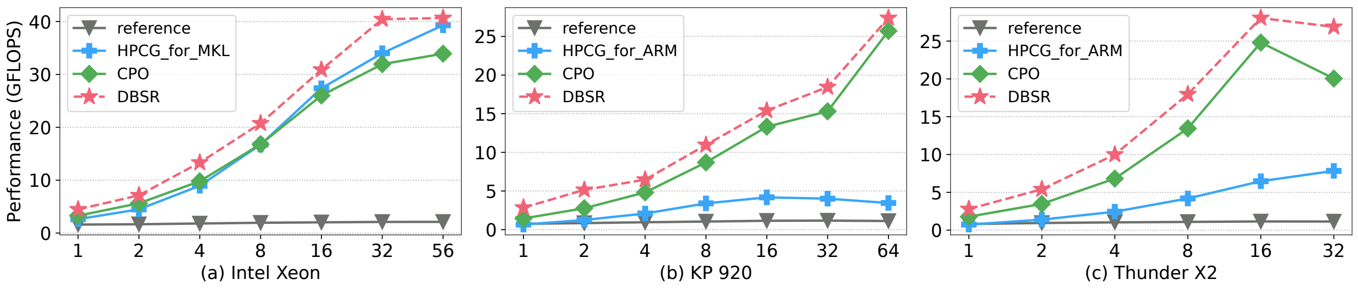


Fig. 6. Performance of HPCG with different number of threads (horizontal axis) on a single node.

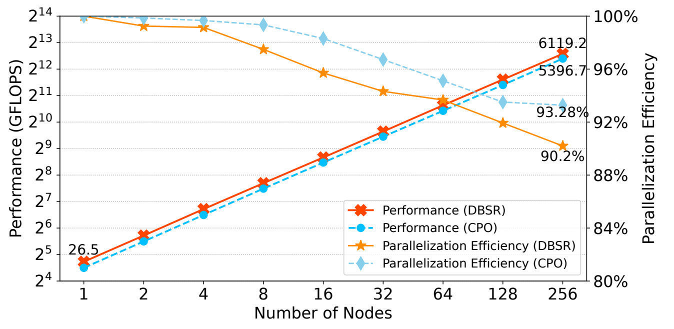


Fig. 7. Performance and parallel efficiency of HPCG optimized with DBSR strategy in the weak scaling sense.

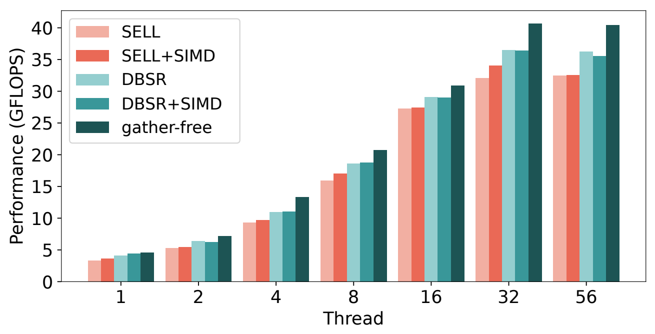


Fig. 8. Performance comparison of HPCG in DBSR format vs. SELL format and the impact of SIMD instructions on Intel Xeon CPU.

with a local problem size of $nx = ny = nz = 192$. For the ILU(0) experiments, various parallelization strategies are used to compare and elucidate the reasons why DBSR exhibits high efficiency in terms of memory usage and parallelism. In addition, we use the MG technique to speed up its convergence. The datasets contain 27-point and 7-point stencil geometric grids of varying precision scaled to $nx = ny = nz = 256$.

Our experiments show the results of HPCG optimized by DBSR in sections V-B, V-C and V-D. Sections V-E, V-F and V-G explore the benefits of DBSR for ILU(0) preconditioner. We show single-node results with Intel Xeon, KP 920 and Thunder X2 CPUs, and evaluate the weak scalability of HPCG at multiple nodes on the Phytium 2000+ platform.

B. HPCG Performance on Single-node

We first present the performance of the improved HPCG on a single computing node using all the physical processor cores. The optimized versions to be demonstrated are outlined below:

- `reference`: Official release version of HPCG-3.1.
- `HPCG_for_MKL/ARM`: Vendor-optimized versions tailored for specific architectures.
- `CPO`: Incorporating state-of-the-art computational and parallel optimization techniques from [24], [25].
- `DBSR`: Our DBSR approach combined with `CPO`.

The experiment will be divided into two parts, one with a multi-process, multi-thread scenario and the other with a single-process, multi-thread scenario. In this experiment, the DBSR method sets $bsize$ to 8 and utilizes SIMD techniques.

Optimal distribution results. We allocate multiple processes and threads to each optimization strategy to maximize the utilization of the nodes. Fig. 5 reports their performance (GFLOPS). The DBSR version achieved the best results in all cases. Overall, the DBSR approach improves the state-of-the-art strategies by 20.5%∼23.9%, 18.8%∼20.4% and 18.9%∼21.4% on Intel, KP 920 and Thunder X2 platforms, respectively. Moreover, the performance improvement achieved by `DBSR` ranges from 1.47x to 1.70x compared to `HPCG_for_MKL`, and from 2.41x to 3.40x compared to `HPCG_for_ARM`.

Parallelism comparison. Fig. 6 illustrates the performance of each method for different numbers of threads. The `DBSR` version always has the best performance. The DBSR technology improves performance by 18.8%∼36.2% and 15.2%∼52.2% compared to `CPO` on x86 and ARM platforms, respectively. Ultimately, the DBSR-optimized HPCG demonstrates a performance boost ranging from 1.03x to 1.70x compared to the MKL version and from 4.32x to 12.39x compared to the ARM version.

C. Scalability Testing

Fig. 7 displays the weak scaling scalability test results of HPCG, optimized with the DBSR policy. The experiments were performed on a 256-node Phytium 2000+ cluster, using a total of $16,384$ cores. The scalability tests extend to 2048 MPI processes, each using eight cores. Specifically, the `CPO` version of HPCG attains a performance level of approximately 5400 GFLOPS. Our `DBSR` version, on the other hand, gives a 13.3% improvement over `CPO`. The optimized HPCG achieves a peak performance of 6119.2 GFLOPS. The parallel efficiency is consistently above 90% as the number of nodes increases. This remarkable level of performance was achieved using a variety of optimization techniques, with the DBSR strategy playing a critical role in further improving performance.

D. Analysis of SIMD Instructions

The advantage of DBSR over SELL when using SIMD technology is `gather`-free, which brings significant benefits. Fig. 8 compares the HPCG performance using the DBSR format and the SELL format on Intel platforms, and reflects the impact of using the `gather` instruction. They are both improvements to the `CPO` version. The SELL format does not reduce storage overhead compared to the CSR format, so performance gains are minimal. The DBSR format provides an average of 15.8% improvement over the SELL format. When using SIMD technology, the SELL format inevitably uses the `gather` instruction, and we did not get a significant performance gain due to its own overhead offsetting the benefits. This is also true for our DBSR strategy if we do not avoid this instruction. However, if we replace the `gather` instruction, the gain of the SIMD technique is about 12.4% on average. We emphasize that the DBSR strategy in other experiments is `gather`-free as long as the SIMD technique is used.

Note that DBSR is tailored for structured grid problems, and extending it to unstructured grid problems poses challenges. In contrast, SELL is compatible with general sparse matrices.

E. Evaluation of ILU(0) Smoothing

The ILU(0) preconditioner is also used to illustrate the advancements of the DBSR strategy. We conduct a comparative analysis of several parallel strategies commonly used with ILU(0) smoothing, including block Jacobi (noted as **BJ**), multi-color (noted as **MC**), block multi-color (noted as **BMC**) methods, where BMC uses fixed block sizes of 64 [19] (noted as **BMC-FIX**) and optimal block sizes [24] (noted as **BMC-AUTO**), respectively. The **DBSR-FIX/AUTO** strategy is set up the same as the BMC. **SIMD-FIX/AUTO** represents the use of the SIMD instruction for DBSR, the AVX512 instruction for Intel Xeon platforms, and the NEON instruction for KP 920 and Thunder X2 platforms. All methods stop iterating when equal and sufficiently small residuals are reached to calculate their respective solution speeds.

Fig. 9 illustrates the speedups of the above techniques compared to the serial ILU(0) solution rate for different numbers of threads. Each platform contains single- and double-precision results for 27-point and 7-point stencil grids. The BJ method maintains a high speedup ratio due to the absence of synchronization waits and the load is balanced. However, as the number of threads increases, the BJ method may omit too much data, resulting in slower convergence and ultimately lower performance. The BJ method achieves maximum acceleration ratios from 6.90x to 12.86x in double-precision and from 8.89x to 18.13x in single-precision on various platforms. The MC method performs poorly because it requires significantly more iterations. BMC follows the same basic principles as the MC, but BMC has faster convergence and better locality, resulting in higher performance. Empirically, the BMC-AUTO outperforms the BMC-FIX. The maximum acceleration range of the BMC-AUTO is from 9.46x to 20.21x in double-precision and from 10.77x to 24.54x in single-precision, respectively. In all cases, the DBSR strategy consistently demonstrates the highest speedup among all parallel solving strategies. Especially when a large number of threads are used, the actual bandwidth limits the memory access rate of other methods, leading to performance degradation. However, DBSR significantly saves index storage and greatly reduces memory access requirements. The DBSR strategy outperforms the BMC strategy by 11% to 17% in double-precision and by 16% to 40% in single-precision in all cases across all platforms. In addition, SIMD technology enabled the DBSR to achieve the best speedup of 11.53x, 21.47x and 17.82x on three platform, respectively. The final average improvements over the BMC strategy ranged from 17% to 26% in double-precision and 22% to 46% in single-precision, respectively.

F. Analysis of Block Size

The key to DBSR storage is the choice of data block size, which directly affects the storage overhead of the matrix and hence the performance. Firstly, it caters to vector processors of varying lengths. Secondly, it captures the extent of zero-value padding that arises from grid boundary concerns. Therefore, we analyze the impact of $bsize$ in this section. Fig. 10 shows the smoothing time for different sizes of $bsize$ on intel

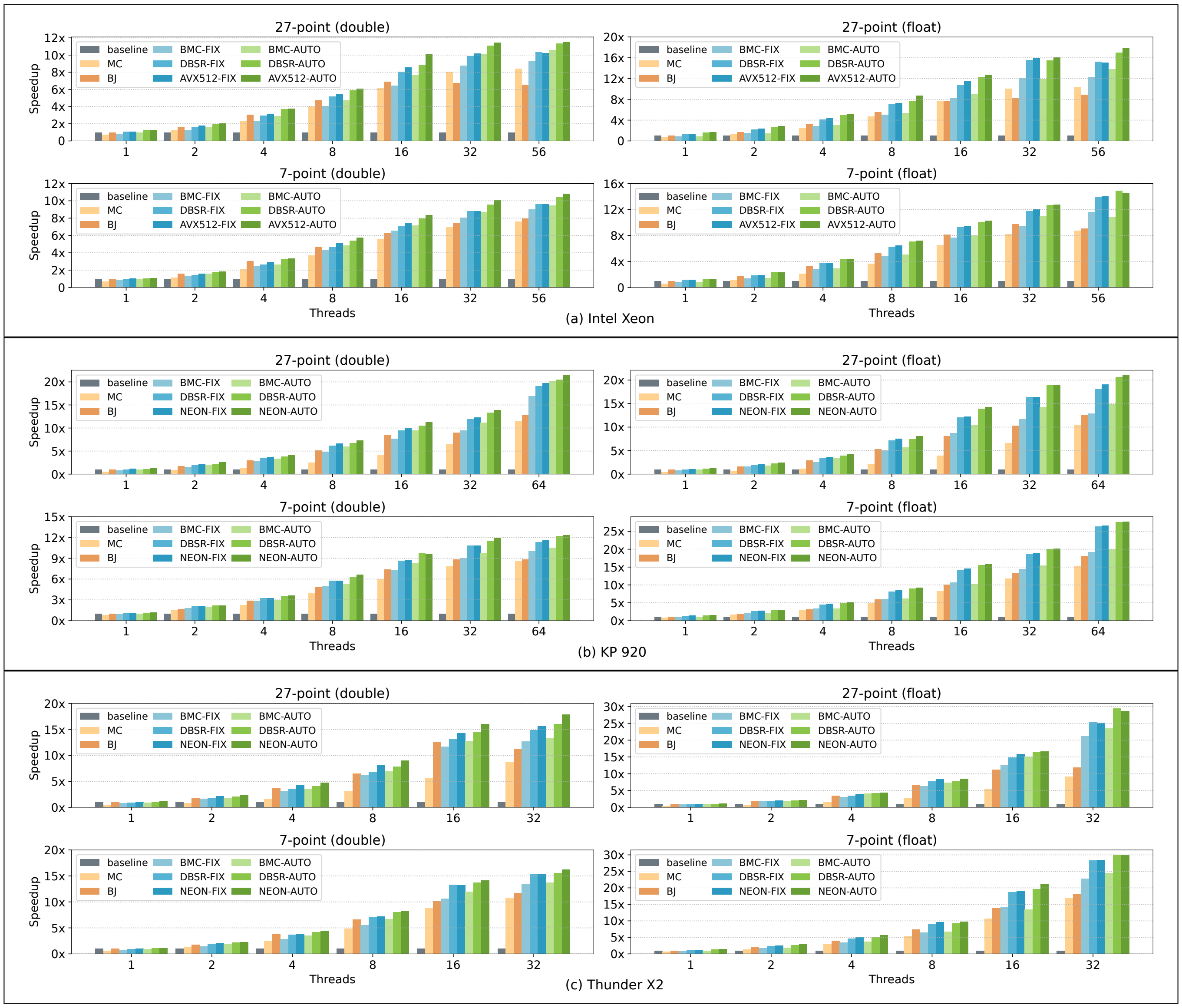


Fig. 9. Parallel test results for various parallel strategies in the smoothing phase of ILU(0), presented in terms of speedup achieved over serial smoothing. Test results for each platform include double-precision (upper left) and single-precision (upper right) for the stencil 27-point grid, and double-precision (lower left) and single-precision (lower right) for the stencil 7-point grid, respectively.

platform. The value range of $bsize$ is $1, 2, 4, 8, 16, 32$ and 64. After $bsize$ reaches 16, the performance remains almost stable. This is because the longer $bsize$ is, the more parallelism is required, which leads to more complex partitioning and lower convergence rate.

To show the advantages of the DBSR format in detail, we analyze the matrix storage overhead for different $bsize$ lengths. We compare this with the basic CSR format. The main matrix storage arrays in CSR format are row_ptr (int), col_ind (int) and $values$ (double). The storage array in DBSR format consists of blk_ptr (int), blk_ind (int), blk_offset (int) and $values$ (double). Fig. 11 shows the result of converting their storage to the number of bytes. The total storage overhead of the DBSR format continues to decrease as $bsize$ increases, and in particular the reduction in row and column indexes is much greater than the amount of zero-value padding. We have observed that the acceleration achieved in single-precision computations is more notable compared to double-precision. This is attributed to the fact that in single-precision, the storage of indexes contributes a larger portion to the matrix storage overhead. Therefore, this approach offers greater advantages. Since the values of blk_offset are all among ($-bsize$, $bsize$), the storage overhead of the DBSR will be further reduced if a smaller storage type is selected.

The DBSR format can be varied according to the SIMD length supported by the hardware platform to efficiently utilize hardware resources. Indeed, in multigrid computations, $bsize$ can be scaled according to the size of each layer of the grid to ensure the need for parallelism.

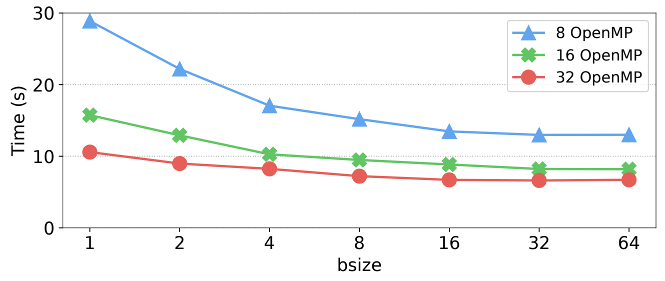


Fig. 10. The smoothing times of DBSR-based ILU(0) with different *bsize*.

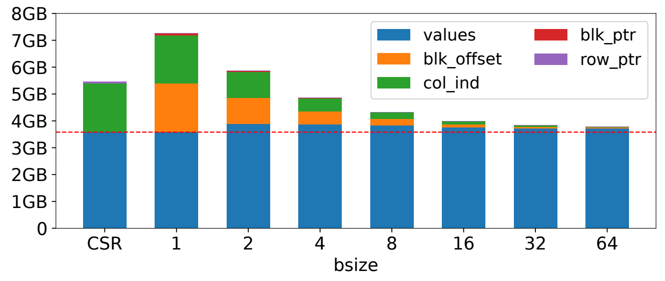


Fig. 11. Comparison of storage overhead between DBSR and CSR format (measured in bytes), varying the *bsize* length selected by DBSR (horizontal axis numbering). For the *values*, original non-zeros are below the dashed line, and zero-value padding is located above the dashed line.

G. The Overhead of Factorization

According to the description of the Algorithm 4, the DBSR strategy also accelerates the LU factorization phase of ILU(0). Fig. 12 compares the efficiency of different parallelization strategies in the LU factorization phase on the Intel platform. For ease of observation, we uniformly use the relative number of iterations in the smoothing phase with the DBSR format to reflect the time spent in the LU factorization phase, i.e., the ratio of factorization time to one smoothing time. It can be seen that the variation of the elapsed time in the factorization phase for both the MC and BMC methods has the same pattern as that of the solution phase. The BMC method is faster due to better localization of the data. The DBSR strategy based on the BMC approach reduces the storage overhead and provides the best results. It spends as much time in the factorization phase as it does smoothing only once. Only the BJ method can catch up with the DBSR strategy at higher parallelism, but the smoothing efficiency of the BJ method is low. In addition, we emphasize that DBSR also supports SIMD techniques during the factorization phase, which can further improve the factorization efficiency.

VI. Related Work

The solution of sparse linear systems is often a bottleneck in engineering applications, and many efforts have been dedicated to develop fast linear solvers. SPTRSV is an important component of both sparse direct and iterative methods. The main optimization strategies are divided into two categories: `level-scheduling` and `color-reordering`. The former has been proposed in [12] and leverages the inherent parallelism of sparse matrices through scheduling [13], [39]. The latter, on the other hand, improves the original dependencies and enhances parallelism through reordering [14], [15].

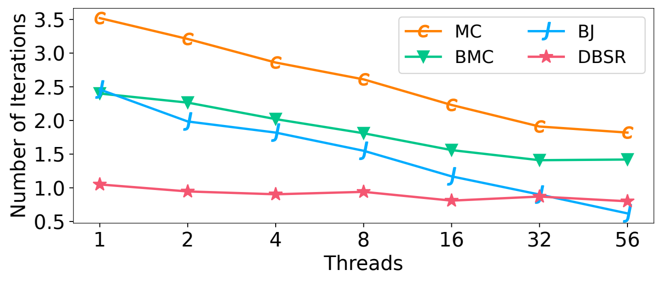


Fig. 12. The factorization comparison of ILU(0) using various parallel schemes. The factorization time has been converted into the number of smoothing iterations using the DBSR format.

Reordering technique is a major parallelization method for sparse matrix computations. It has long been investigated in the analysis of structured grids [40], [41]. Various commonly used reordering methods have been compared in [42]. MC and BMC reordering are the most classical ordering techniques. An algebraic block multicoloring (ABMC) method suitable for solving general sparse linear systems is presented in [43]. For the first time in [44], BMC is being used to accelerate HPCG, using the SELL storage format and leveraging the SIMD parallelism of Intel Xeon Phi. In contrast, we propose a new storage format DBSR in this work, which is more efficient than SELL for structured grids. Much subsequent work has focused on applying BMC techniques to the HPCG benchmark [26], [32], [45]. The key to the BMC technique is to choose the right number of colors and block sizes to pursue a balance between convergence and parallelism. Common block partitioning schemes are fixed block size partitions, such as 64 [19], and dynamic partitions based on the number of compute resources [24]. The experiments in this paper also use these schemes.

ILU is one of the most popular and successful preconditioners for iterative methods. Just as other SPTRSVs, parallel processing of ILU is difficult. Several parallel strategies have been proposed for ILU [6], [28], and parallel reordering is one the best known techniques for constructing and applying ILU [15], [40], [46]. It was shown in [40] that the convergence of ILU can be significant affected by ordering, and different parallel orderings including BMC are compared in [47]. Our vectorized BMC has the same convergence rate as BMC. The recently proposed novel fine-grained ILU factorization [48] uses a fixed-point iteration to approximate the incomplete factors, which can not use the SIMD vectorization either. By introducing a new fill-in control method based on blocks, a block ILU preconditioning is also proposed in [49], which can use the SIMD instructions but may populate a large number of fill-ins. In contrast, our DBSR exploits the sparse structure of these blocks instead of treating them as dense. Our DBSR strategy, which is designed specifically for structured grids, can be used for a variety of preprocessing techniques and is not limited to ILU(0) preconditioner. The DBSR format introduces only a small number of zero elements to the storage matrix, but reduces most of the indexes, thus reducing the total storage overhead. In addition, DBSR is vectorization friendly. These zero elements in the vectorized ILU(0) preconditioner do not affect the result and do not change the number of non-zero elements.

Parallelizing SPTRSV on GPUs is challenging, but most of our optimization techniques should be transferable to GPUs. Presently, SPTRSV on GPUs predominantly employs the level-set scheduling method, which introduces substantial synchronization overhead. In contrast, DBSR offers better parallelism. However, when adapting DBSR to GPUs, the high concurrency of GPUs requires using MC or BMC methods with very small task blocks, potentially resulting in slower convergence and suboptimal performance. Therefore, this paper refrains from examining the performance of DBSR on GPUs, with the aim of pursuing more sophisticated and efficient solutions in future work.

VII. Conclusion

In this paper, a new storage format DBSR based on matrix reordering is proposed for the linear solution problem of sparse structure matrices. The DBSR format divides the matrix into blocks, with the matrix elements distributed on only one diagonal of the corresponding block. DBSR only records the location of each data block, thus reducing a lot of index storage overhead. This particular distribution also makes it support continuous memory access and vectorized computation. We apply the DBSR technique to a variety of sparse operations, including sparse matrix-vector multiplication (SPMV), symmetric Gauss-Seidel (SYMGS) method, and incomplete LU(0) preconditioner. When used in multigrid (MG) algorithms, they exhibit faster solution speeds on linear solution problems. Combined with state-of-the-art MG optimization techniques, we apply the DBSR strategy to the HPCG benchmark. Experiments on x86 and ARM architectures show that our approach improves the performance of advanced optimization techniques by 18.8% to 23.9%. Finally, the DBSR-optimized HPCG achieves speedups ranging from 1.47x to 3.40x compared to the vendor's highly optimized HPCG implementation. We also compare the performance of various parallel strategies in ILU(0) preconditioner, including block Jacobi, multi-color, and block multi-color. The DBSR strategy has the best computational efficiency in both the LU factorization phase and the smoothing phase. When compared to the other best methods, our approach demonstrates an improvement of 17%$\sim$46% across these platforms.

In future work, we plan to extend our DBSR approach to unstructured grid problems, particularly for specific applications. Additionally, we aim to apply our DBSR approach to other heterogeneous architectures, including GPUs, SX-Aurora [50], and DSP architectures [18]. Furthermore, we will explore efficient reordering methods for more matrices to take advantage of their structural properties.

Acknowledgments

The authors would like to thank the referees for their valuable comments. This work was supported in part by the National Key R&D Program of China under grant agreement 2023YFB3001702, the National Science Foundation of China (NSFC) under grant agreements 61902411, 62032023, 12002382, 11275269, 42104078, U22B2005, and the Excellent Youth Foundation of Hunan Province under grant agreement 2021JJ10050.

References

[1] N. Nikolova, H. Tam, and M. Bakr, "Sensitivity analysis with the FDTD method on structured grids," *IEEE Transactions on Microwave Theory and Techniques*, vol. 52, no. 4, pp. 1207–1216, 2004.

[2] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, "The Cactus framework and toolkit: Design and applications," in *High Performance Computing for Computational Science — VECPAR 2002*. Berlin, Heidelberg: Springer, 2003, pp. 197–227.

[3] L. D. Montejo, A. D. Klose, and A. H. Hielscher, "Implementation of the equation of radiative transfer on block-structured grids for modeling light propagation in tissue," *Biomedical Optics Express*, vol. 1, p. 861, 2010.

[4] C. Yang, W. Xue, H. Fu, H. You, X. Wang, Y. Ao, F. Liu, L. Gan, P. Xu, L. Wang *et al.*, "10M-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 57–68.

[5] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*. Oxford University Press, 01 2017.

[6] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[7] L. Weifeng, "Parallel and scalable sparse basic linear algebra subprograms," Ph.D. dissertation, Denmark, 2015.

[8] I. S. Duff, M. A. Heroux, and R. Pozo, "An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum," *ACM Transactions on Mathematical Software (TOMS)*, vol. 28, no. 2, pp. 239–267, 2002.

[9] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*. New York: ACM, 2015, pp. 339–350.

[10] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C401–C423, 2014.

[11] W. Liu and B. Vinter, "An efficient GPU general sparse matrix-matrix multiplication for irregular data," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 370–381.

[12] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *International Journal of High Speed Computing*, vol. 1, no. 01, pp. 73–95, 1989.

[13] J. Su, F. Zhang, W. Liu, B. He, R. Wu, X. Du, and R. Wang, "CapelliniSpTRSV: A thread-level synchronization-free sparse triangular solve on GPUs," in *Proceedings of the 49th International Conference on Parallel Processing*, 2020, pp. 1–11.

[14] R. Schreiber and W. Tang, "Vectorizing the conjugate gradient method," *Unpublished manuscript*, 1982.

[15] T. Iwashita and M. Shimasaki, "Block red-black ordering: A new ordering strategy for parallelization of ICCG method," *International Journal of Parallel Programming*, vol. 31, no. 1, pp. 55–75, 2003.

[16] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, "Optimization of geometric multigrid for emerging multi- and manycore processors," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–11.

[17] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu *et al.*, "The ARM scalable vector extension," *IEEE micro*, vol. 37, no. 2, pp. 26–39, 2017.

[18] K. Lu, Y. Wang, Y. Guo, C. Huang, S. Liu, R. Wang, J. Fang, T. Tang, Z. Chen, B. Liu *et al.*, "MT-3000: a heterogeneous multi-zone processor for HPC," *CCF Transactions on High Performance Computing*, vol. 4, no. 2, pp. 150–164, 2022.

[19] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, M. M. A. Patwary, V. Pirogov, P. Dubey, X. Liu, C. Rosales *et al.*, "Optimizations in a high-performance conjugate gradient benchmark for IA-based multi-and many-core processors," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 11–27, 2016.

[20] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *High Performance Embedded Architectures and Compilers*, Y. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, Eds. Berlin Heidelberg: Springer, 2010, pp. 111–125.

[21] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. D. Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods*. Philadelphia, PA: SIAM, 1994.

[22] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," Sandia, Tech. Rep. SAND2013-4744, 2013.

[23] J. Dongarra and P. Luszczek, "HPCG technical specification," Sandia National Laboratories, Tech. Rep. SAND2013-8752, 2013.

[24] X. Yang, S. Li, F. Yuan, D. Dong, C. Huang, and Z. Wang, "Optimizing multi-grid computation and parallelization on multi-cores," in *Proceedings of the 37th International Conference on Supercomputing*. New York, NY, USA: ACM, 2023, p. 227–239.

[25] F. Yuan, X. Yang, S. Li, D. Dong, C. Huang, and Z. Wang, "Optimizing multi-grid preconditioned conjugate gradient method on multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 5, pp. 768–779, 2024.

[26] D. Ruiz, F. Mantovani, M. Casas, J. J. Labarta Mancho, and F. Spiga, "The HPCG benchmark: analysis, shared memory preliminary improvements and evaluation on an arm-based platform," https://upcommons.upc.edu/bitstream/handle/2117/116642/1HPCG_shared_mem_implementation_tech_report.pdf?sequence=8&isAllowed=y, 2018.

[27] I. Product, "Developer Reference for Intel oneAPI Math Kernel Library," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html

[28] J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. ver der Vorst, *Numerical Linear Algebra on High-Performance Computers*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1998.

[29] S. Fujino and T. Takeuchi, "ILU factorization well suited to the vector processor using a variant of the 5-point difference scheme," *Computer Physics Communications*, vol. 85, no. 3, pp. 371–381, 1995.

[30] K. Suzuki, T. Fukaya, and T. Iwashita, "A novel ILU preconditioning method with a block structure suitable for SIMD vectorization," *Journal of Computational and Applied Mathematics*, vol. 419, p. 114687, 2023.

[31] Y. Saad, "Sparskit: A basic tool-kit for sparse matrix computation, version 2," 1994, software available at http://www.cs.umn.edu/~saad.

[32] Y. Liu, C. Yang, F. Liu, X. Zhang, Y. Lu, Y. Du, C. Yang, M. Xie, and X. Liao, "623 Tflop/s HPCG run on Tianhe-2: Leveraging millions of hybrid cores," *International Journal of High Performace Computing Applications*, vol. 30, no. 1, pp. 39–54, 2016.

[33] E. Phillips and M. Fatica, "Performance analysis of the high-performance conjugate gradient benchmark on gpus," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 28–38, 2016.

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

[35] Huawei, "Kunpeng 920," 2019. [Online]. Available: https://www.hisilicon.com/cn/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920

[36] F. Mantovani, M. Garcia-Gasulla, J. Gracia, E. Stafford, F. Banchelli, M. Josep-Fabrego, J. Criado-Ledesma, and M. Nachtmann, "Performance and energy consumption of hpc workloads on a cluster based on arm thunderx2 cpu," *Future generation computer systems*, vol. 112, pp. 800–818, 2020.

[37] Phytium, "FT-2000+/64," 2019. [Online]. Available: https://en.wikichip.org/wiki/phytium/feiteng/ft-2000%2B-64

[38] ARM Software, "HPCG for ARM," 2019. [Online]. Available: https://github.com/ARM-software/HPCG_for_Arm

[39] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A synchronization-free algorithm for parallel sparse triangular solves," in *Euro-Par 2016: Parallel Processing*. Cham: Springer International Publishing, 2016, pp. 617–630.

[40] I. S. Duff and G. A. Meurant, "The effect of ordering on preconditioned conjugate gradients," *BIT Numerical Mathematics*, vol. 29, pp. 635–657, 1989.

[41] M. Benzi, W. Joubert, and G. Mateescu, "Numerical experiments with parallel orderings for ILU preconditioners," *Electronic Transactions on Numerical Analysis*, vol. 8, pp. 88–114, 1999.

[42] J. D. Trotter, S. Ekmekçibaşı, J. Langguth, T. Torun, E. Düzakın, A. Ilic, and D. Unat, "Bringing order to sparsity: A sparse matrix reordering study on multicore cpus," *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023.

[43] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in iccg method," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 474–483.

[44] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 945–955.

[45] Y. Ao, C. Yang, F. Liu, W. Yin, L. Jiang, and Q. Sun, "Performance Optimization of the HPCG Benchmark on the Sunway TaihuLight Supercomputer," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, mar 2018.

[46] M. Monga-Made, "Ordering strategies for modified block incomplete factorizations," *SIAM Journal on Scientific Computing*, vol. 16, no. 2, pp. 378–399, 1995.

[47] T. Iwashita, Y. Nakanishi, and M. Shimasaki, "Comparison criteria for parallel orderings in ILU preconditioning," *SIAM Journal on Scientific Computing*, vol. 26, no. 4, pp. 1234–1260, 2005.

[48] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM J. Sci. Comput.*, vol. 37, 2015.

[49] K. Suzuki, T. Fukaya, and T. Iwashita, "A novel ILU preconditioning method with a block structure suitable for simd vectorization," *J. Comput. Appl. Math.*, vol. 419, p. 114687, 2022.

[50] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "HPCG on long-vector architectures: Evaluation and optimization on NEC SX-Aurora and RISC-V," *Future Generation Computer Systems*, 2023.