CCF Transactions on High Performance Computing
https://doi.org/10.1007/s42514-023-00151-1

REGULAR PAPER

TileSpTRSV: a tiled algorithm for parallel sparse triangular solve on GPUs

Zhengyang Lu[1] · **Weifeng Liu**[1]

Received: 17 December 2022 / Accepted: 3 May 2023

Abstract
Sparse triangular solve (SpTRSV) is one of the most important level-2 kernels in sparse basic linear algebra subprograms (BLAS). Compared to another level-2 sparse BLAS kernel sparse matrix–vector multiplication (SpMV), SpTRSV is in general more difficult to find high parallelism on many-core processors, such as GPUs. Nowadays, much work focuses on reducing dependencies and synchronizations in the level-set and Sync-free algorithms for SpTRSV. However, there is less work that can make good use of sparse spatial structure for SpTRSV on GPUs. In this paper, we propose a tiled algorithm called TileSpTRSV for optimizing SpTRSV on GPUs through exploiting 2D spatial structure of sparse matrices. We design two algorithm implementations, i.e., `TileSpTRSV_level-set` and `TileSpTRSV_sync-free`, for TileSpTRSV on top of level-set and Sync-free algorithms, respectively. By testing 16 representative matrices on a latest NVIDIA GPU, the experimental results show that `TileSpTRSV_level-set` gives on average 5.29× (up to 38.10×), 5.33× (up to 21.32×) and 2.62× (up to 12.87×) speedups over cuSPARSE, Sync-free and Recblock algorithms on the 16 representative matrices, respectively.

Keywords Sparse matrix · Sparse triangular solve · Tiled algorithm · GPU

1 Introduction

Sparse triangular solve (SpTRSV) is an operation that solves the linear equation $Lx = b$ (or $Ux = b$), where L (or U) is a sparse lower (or upper) triangular matrix, x is a dense vector to be solved and b is a dense right-hand side vector. SpTRSV is an important building block of level-2 sparse basic linear algebra subprograms (BLAS) (Liu 2015). It has a number of applications in numerical computations, such as the solve phase of sparse direct solvers (Davis 2006; Duff et al. 2017; Li 2005; Zhao et al. 2021; Wang et al. 2023) and the preconditioner of sparse iterative solvers (Anzt et al. 2015, 2016, 2018a, b).

Compared with other BLAS kernels such as sparse matrix–vector multiplication (SpMV) (Liu and Vinter 2015b, c; Niu et al. 2021) and sparse matrix-matrix multiplication (SpGEMM) (Liu and Vinter 2015a; Hou et al. 2017; Liu et al. 2019; Xie et al. 2019; Niu et al. 2022), SpTRSV is inherently sequential and generally cannot find satisfactory parallelism on modern many-core processors, due to possible dependencies between the components.

Fortunately, though SpTRSV is difficult to process in parallel, much research has demonstrated that parallel SpTRSV algorithms are possible. Anderson and Saad (1989) and Saltz (1990) propose the level-set method which is a classical parallel algorithm for SpTRSV. They see the input matrix as a graph and divide the components into many sets. The components in each set can be calculated independently, and the calculation between sets should be executed sequentially. The algorithm to some extent resolves the parallel problem of SpTRSV. But with the increase of the number of sets, the level-set algorithm would use much time for global synchronization on parallel processors. To address this problem, Liu et al. (2016, 2017) propose the synchronization-free algorithm (or Sync-free for short). It replaces the barriers with atomic operations on GPUs for further eliminating the expensive cost of global synchronization.

In another direction of SpTRSV optimization, a number of research focuses on the 2D layout and spatial structure of input matrix. Lu et al. (2020) propose a recursive block algorithm (or Recblock for short), which recursively divides the

✉ Weifeng Liu
weifeng.liu@cup.edu.cn

[1] Super Scientific Software Laboratory, China University of Petroleum-Beijing, Beijing, China

Published online: 12 June 2023

input matrix into multiple triangular and square sub-matrices, and uses an adaptive method to automatically select method for each block. In addition to that, for other BLAS kernels, Niu et al. (2021, 2022) and Ji et al. (2022) exploit tiled structure to implement TileSpMV, TileSpGEMM and TileSpMSpV, for SpMV and SpGEMM and sparse matrix-sparse vector multiplication (SpMSpV) respectively, and achieve obviously better performance than existing algorithms. This motivates us to enrich the group of tiled algorithms for SpTRSV in sparse BLAS.

In this paper, we propose an efficient tiled algorithm called TileSpTRSV for SpTRSV and implement two versions of TileSpTRSV: `TileSpTRSV_level-set` and `TileSpTRSV_sync-free`. The two implementations utilize different parallel SpTRSV algorithms and are built on top of the level-set and Sync-free algorithms, respectively. Although they have different calculation modes, the two TileSpTRSV implementations share the same low level storage structure.

TileSpTRSV divides the input matrix into a number of regular sparse tiles of the same size (always 16-by-16) to obtain better data locality and higher bandwidth utilization. It treats each tile as a basic working unit. For each off-diagonal tile, TileSpTRSV stores it in seven different formats according to the internal structure of the tile. For each diagonal tile, TileSpTRSV stores the tile with all nonzeros distributed on the diagonal in the DIA format; otherwise, it would be stored in the CSR format. After that, we use corresponding GPU kernels for processing different formats in the calculation phase.

In our experiments, we use an NVIDIA GeForce RTX 4090 as our experimental platform, and select 16 representative matrices from the SuiteSparse Matrix Collection (Davis and Hu 2011) as our dataset. The experimental results show that our TileSpTRSV gives on average 5.29× (up to 38.10×), 5.33× (up to 21.32×) and 2.62x (up to 12.87×) speedups over cuSPARSE, Sync-free (Liu et al. 2016, 2017) and Recblock algorithms (Lu et al. 2020), respectively.

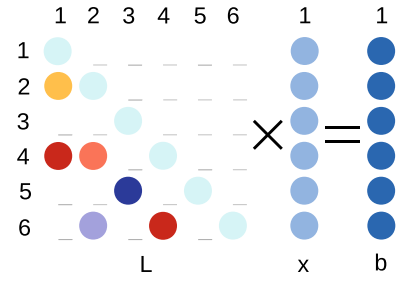

Fig. 1 An example of serial SpTRSV algorithm that solves a resulting vector x with a 6-by-6 sparse matrix and a dense right-hand side vector b

2 Background

2.1 Serial SpTRSV

Serial SpTRSV algorithm executes SpTRSV calculation sequentially because of its inherently dependencies between components. Figure 1 shows an example of serial SpTRSV operation on a 6-by-6 sparse matrix. In Algorithm 1, the off-diagonal value in the i-th row ($val[j]$) is multiplied by the corresponding value of x ($x[colidx[j]]$), and the result is updated to a temporary vector (tmp_sum). Finally, x_i can be updated by the step in line 6. It should be noted that the process to solve the resulting vector x is sequential (from x_0 to x_{m-1}).

Algorithm 1 A serial algorithm for CSR-SpTRSV.

```
1: function SPTRSV-SERIAL()
2:     for i = 0 to m − 1 do
3:         for j = rowptr[i] to rowptr[i + 1]−2 do
4:             tmp_sum[i] ← tmp_sum[i] + val[j] × x[colidx[j]]
5:         end for
6:         x[i] ← (b[i]-left_sum[i])/val[row_ptr[i + 1]−1]
7:     end for
8: end function
```

2.2 Level-set parallel SpTRSV

Level-set algorithm and Saltz (1990).

Algorithm 2 A simplified level-set algorithm for CSR-SpTRSV.

```
1:  function PREPROCESS-LEVELSET()
2:      for li = 0 to n − 1 do
3:          if DEPENDENCIES(li)= 0 then
4:              level_ptr[li]++
5:              nlv++
6:          end if
7:          PREFIX-SUM(level_ptr, n + 1)
8:      end for
9:      for li = 0 to nlv − 1 do
10:         for i =level_ptr[li] to level_ptr[li + 1]−1 do
11:             INSERT(level_item, i)
12:         end for
13:     end for
14: end function
15: function CALCULATE-LEVELSET()
16:     for li = 0 to nlevel − 1 do
17:         for i =level_ptr[li] to level_ptr[li + 1]−1 in parallel do
18:             for j = rowptr[level_item[i]] to rowptr[level_item[i]+1]−1 do
19:                 tmp_sum[i] ← tmp_sum[i] + val[j] × x[colidx[j]]
20:             end for
21:             x[i] ← (b[i]-tmp_sum[i])/val[rowptr[i + 1]−1]
22:         end for
23:     end for
24: end function
```

This algorithm can exploit possible parallelism in the input matrix. It sees the input matrix as a graph and divides it into multiple sets which depend on each other, but the components in each set can be calculated independently. Figure 2 shows the level form of an 8-by-8 matrix after its partition. The calculation of higher-level components is dependent on the computation results obtained from lower-level components. For example, to calculate row_5 in *level*2, we need the results of row_0 in *level*0 as well as row_2 and row_4 in *level*1. Algorithm 2 shows the complete process to solve SpTRSV with level-set algorithm. The preprocessing function (lines 1–14) shows the process to build corresponding data structure, and the calculation function (lines 15–24) solves SpTRSV with the level-set algorithm.

2.3 Synchronization-free parallel SpTRSV

Algorithm 3 A simplified Sync-free algorithm for CSC-SpTRSV.

```
1:  function COMPUTE-IN-DEGREE(*row_ptr, *col_idx, n, nnz, *in_degree)
2:      for i = 0 to nnz − 1 in parallel do
3:          ATOMIC-INCR(&in_degree[col_idx[i]])
4:      end for
5:  end function
6:  function SOLVE-SPTRSV(*row_ptr, *col_idx, *val, *b, n, nnz, *in_degree)
7:      for i = 0 to n − 1 in parallel do
8:          while in_degree[i] ≠ 1 do
9:              //busy wait
10:         end while
11:         x[i] ← (b[i]-left_sum[i])/val[row_ptr[i]]
12:         for j = row_ptr[i]+1 to row_ptr[i + 1]−1 in parallel do
13:             ATOMIC-ADD(&left_sum[col_idx[j]], val[j] × x[i])
14:             ATOMIC-DECR(&in_degree[col_idx[j]])
15:         end for
16:     end for
17: end function
```

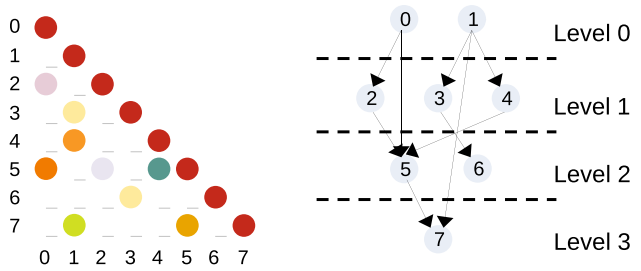

Fig. 2 An example of an 8-by-8 sparse lower triangular matrix L and its corresponding level-set form

Synchronization-Free (Sync-free for short) is a parallel algorithm proposed by Liu et al. (2016, 2017) for SpTRSV on GPUs. The Sync-free algorithm uses fast atomic operations instead of barriers in the level-set algorithm. This can reduce the cost of generating level-sets and eliminate the cost of global synchronization. Algorithm 3 shows a pseudocode of simplified Sync-free process, including preprocessing and calculation stages. The preprocessing function (lines 1–5) generates the array *in_degree* with low overhead. The array records the number of dependent entries of each nonzero element, which means that these entries must be processed in advance; otherwise, the corresponding entry cannot be calculated. In the calculation function (lines 6–17), the algorithm assigns a GPU working unit (a warp with 32 threads in CUDA) to process each column *i*. Note that it first busy-waits until the corresponding value of *in_degree* becomes '1', indicating that the dependencies are removed (lines 8–10). Then, it starts to compute x_i (line 11) and notifies all the later entries that depend on x_i through atomic operations (lines 12–15). As we can see, the Sync-free algorithm only needs to launch one GPU kernel. Compared to the level-set algorithm, this algorithm can eliminate the consumption of global synchronization. Algorithm 3 shows the Sync-free algorithm in compressed sparse column (CSC) format, and a CSR version of the Sync-free algorithm is developed by Dufrechou and Ezzatti (2018b).

3 TileSpTRSV

3.1 Overview

In the preprocessing phase, our TileSpTRSV algorithm first divides the input matrix into multiple 16-by-16 sparse tiles to enhance data locality. We then store the off-diagonal tiles in seven different formats (CSR, COO, ELL, HYB, Dns, DnsRow, and DnsCol) and store the diagonal tiles in either the CSR or DIA format. Section 3.2 provides an introduction to the two-level storage structure utilized by TileSpTRSV.

After partitioning the input matrix, a number of off-zero sparse tiles are generated. In this work, we utilize a format selection strategy to choose an appropriate format from the eight available formats for these sparse tiles. We create a selection tree that can select a suitable format for each tile. Section 3.3 provides an introduction to our format selection strategy for TileSpTRSV.

In the calculation phase, we utilize seven warp-level SpMV kernels for seven distinct formats to process the off-diagonal tiles. For the diagonal tiles, we develop thread-level and warp-level SpTRSV kernels for the CSR format and one thread-level kernel for the DIA format. Section 3.4 provides a detailed introduction of these tile-level algorithms.

We implement two versions of TileSpTRSV, one of which is `TileSpTRSV_level-set`, built on top of the level-set algorithm. This algorithm can divide all off-zero sparse tiles into multiple sets, and process the sparse tiles within the

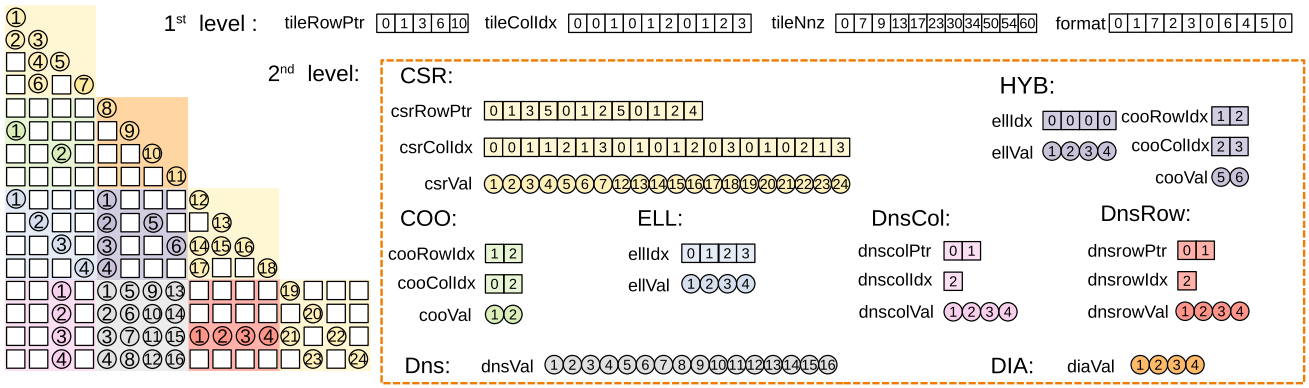

Fig. 3 An example of a lower triangular matrix with dimensions of 16-by-16, stored in 10 sparse tiles of size 4-by-4. The tile structure consists of three arrays, *tileRowPtr*, *tileColIdx*, and *tileNnz*, which represent the offsets for the number of tiles in row-major order, tile column indices, and the offsets for the number of nonzeros in sparse tiles, respectively. The *format* array records the format of each sparse tile. In accordance with the format selection strategy, the diagonal tiles of the 10 tiles are stored in the CSR and DIA formats, while the remaining six tiles are stored in different formats, including COO, ELL, HYB, Dns, DnsRow, and DnsCol in this example. Each format has several corresponding arrays which are used to store the information of nonzeros

same set in parallel. Section 3.5 provides a detailed introduction to this algorithm.

The other implementation of the TileSpTRSV algorithm is called `TileSpTRSV_sync-free`, and is built on top of the Sync-free algorithm. It uses fast atomic operations to reduce global synchronization consumption. Section 3.6 introduces the execution process of this algorithm.

3.2 Two-level storage structure

Our TileSpTRSV uses a 2-level storage structure to store the information of the input sparse triangular matrix. Figure 3 shows an example where an input matrix is divided into 10 sparse tiles and stored in a two-level storage structure that contains multiple arrays. To improve data locality, TileSpTRSV divides the matrix into many sparse tiles of size 16-by-16, so that one 'unsigned char' (8-bit data type) can store the position information of nonzeros in one sparse tile. For clarity, we set the tile size to 4-by-4 instead of 16-by-16 in Fig. 3. After partitioning, the first-level storage structure is generated.

In the first-level, the tile information is stored in the CSR format, and three arrays (*tileRowPtr*, *tileColIdx*, and *tileNnz*) are generated. *tileRowPtr* records the number of off-zero sparse tiles in each row, *tileColIdx* records the tile column index of each tile, and array *tileNnz* records the number of nonzeros in each sparse tile. After that, TileSpTRSV uses a format selection strategy to select a suitable format for each tile. Section 3.3 explains the selection strategy in detail. When the format of each tile has been selected, the second-level of the storage structure and the array *format* can be generated (array *format* records the storage format of each sparse tile).

In the second-level, TileSpTRSV stores the internal information of each sparse tile in different formats. In Fig. 3, we list eight groups of arrays representing the information of the eight formats, respectively. We use the same storage structure for the seven formats (CSR, COO, ELL, HYB, Dns, DnsRow, and DnsCol) as in TileSpMV (Niu et al. 2021). Taking the CSR format as an example, it has three arrays (*csrRowPtr*, *csrColIdx*, *csrVal*) to store the internal information of all CSR sparse tiles. It should be noticed that we combine the information of the standard CSR format of all sparse tiles in the CSR format, but we do not record the last value of the row pointer array. It is because the actual value of it may exceed 255, which is the ceiling of one 'unsigned char'. But we can get the value through searching the array *tileNnz*. For the diagonal tiles where the nonzeros are all concentrated on the diagonal, we store them in the DIA format and only use one array *diaVal* to store the value of nonzeros in the tiles. The orange tile in Fig. 3 is an example.

3.3 Format selection strategy

We design a format selection strategy for TileSpTRSV to select appropriate format for each tile. For off-diagonal tiles, we use the same selection method as proposed by TileSpMV (Niu et al. 2021), and these tiles can be stored in seven different formats: CSR, COO, ELL, HYB, Dns, DnsRow, and DnsCol based on the nonzeros count and sparse structure. For diagonal tiles, our format selection strategy is as follows: (1) DIA format: We store the off-zero sparse tiles where the nonzeros are concentrated on the diagonal line in this format. It can reduce the space cost compared to the CSR format. (2) CSR format: We uniformly store the remaining sparse tiles in the CSR format.

3.4 Tile-level algorithms

After the storage format of each tile has been determined, we use different tile-level algorithms for tiles stored in different formats. Note that we use the same seven tile-level algorithms proposed by TileSpMV (Niu et al. 2021) for off-diagonal tiles stored in seven different formats to perform SpMV operations, so we will not discuss them in this paper. For tiles stored in DIA format, we use a thread-level SpTRSV kernel to perform the division operation directly since all nonzeros are distributed on the diagonal. For diagonal tiles stored in the CSR format, we design thread-level and warp-level kernels to execute different SpTRSV algorithms. The two tile-level SpTRSV kernels will be introduced as follows.

In the thread-level SpTRSV kernel, we use one thread to perform a serial SpTRSV calculation for each diagonal tile. Algorithm 4 shows the computation process of the thread-level SpTRSV kernel.

Algorithm 4 A pseudocode of thread-level SpTRSV kernel.

```
1: ptr_offset ← csrptr_offset[blkid]
2: offset ← csr_offset[blkid]
3: csrRowPtr_tile ← &csrRowPtr[ptr_offset]
4: csrVal_tile ← &csrVal[offset]
5: csrColIdx_tile ← &csrColIdx[offset]
6: for i = 0 to tile_size − 1 do
7:     for j = csrRowPtr_tile[i] to csrRowPtr_tile[i + 1]−2 do
8:         left_sum[i]←left_sum[i]+csrVal_tile[j]×x[csrColIdx_tile[j]]
9:     end for
10:    x[i] ← (b[i]-left_sum[i])/val[csrRowPtr_tile[i + 1]−1]
11: end for
```

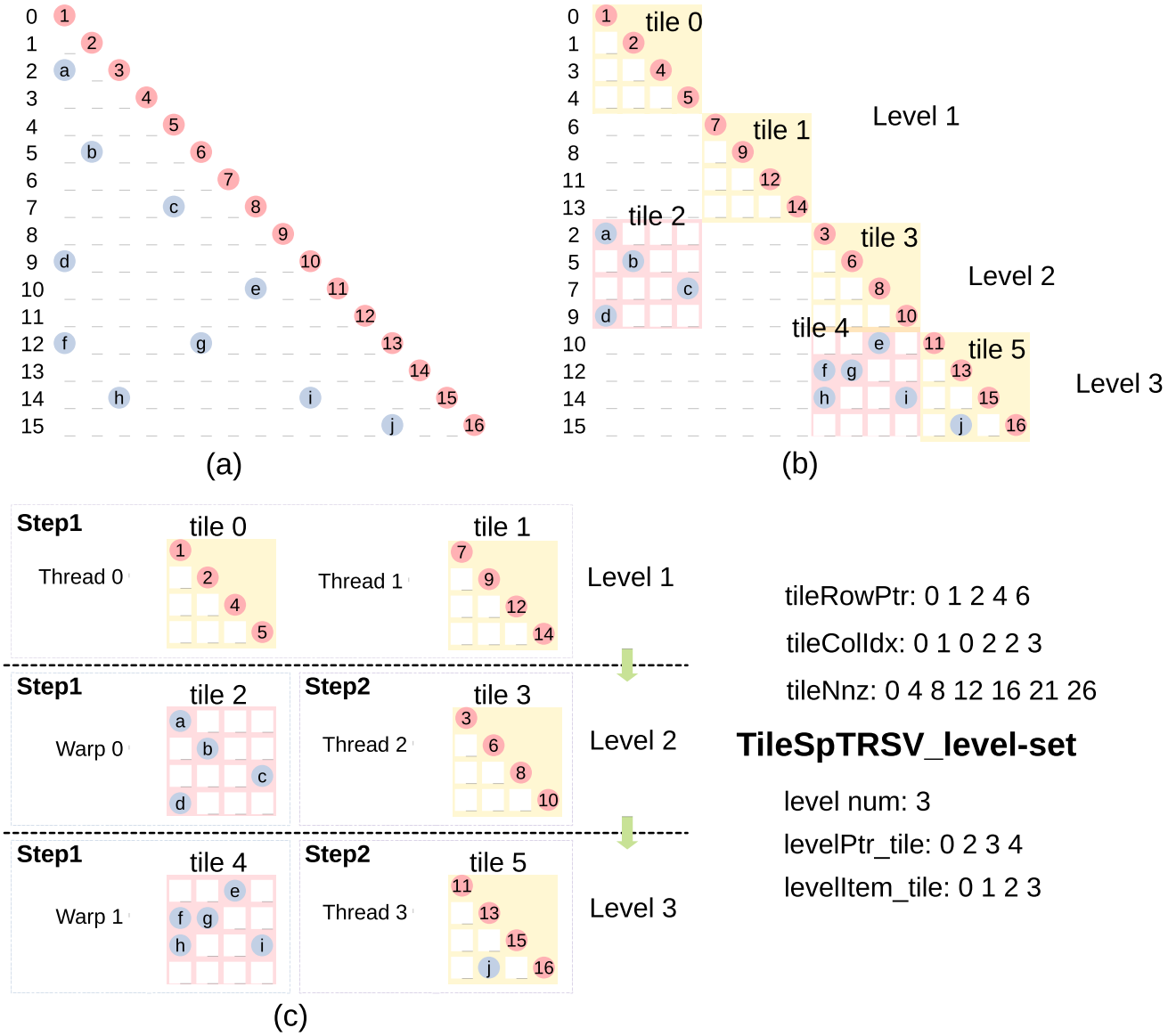

Fig. 4 An example is provided to illustrate the calculation process of `TileSpTRSV_level-set`. In this algorithm, the tile information is stored in the CSR format, which includes the *tileRowPtr*, *tileColIdx*, and *tileNnz* arrays. **a** shows an input matrix *L*, while **b** shows the result of the input matrix after reordering and partitioning. After partitioning, the six tiles are divided into three levels and generate a group of data that includes the *levelnum*, *levelPtr_tile*, and *levelItem_tile*. **c** shows the calculation process of `TileSpTRSV_level-set`. The algorithm processes each level sequentially and processes off-diagonal tiles (step 1) before diagonal tiles (step 2) in the same level

In the warp-level SpTRSV kernel, we use the Sync-free algorithm for SpTRSV calculation. A 32-thread warp is always assigned to process a tile with 16 columns, which means that every two consecutive threads process one column. Before calculation, every diagonal tile would generate an array *graphInDegree* which records the nonzeros of each row in a tile. In the calculation process, every two consecutive threads of a warp correspond to one value of the array *graphInDegree*, and only when the value equals '1', the threads can process the column and use atomic operation to modify the array *graphInDegree*. Otherwise, the threads would be blocked until the value of the array *graphInDegree* becomes '1'. Algorithm 5 shows the execution process of the warp-level SpTRSV kernel.

Algorithm 5 A pseudocode of warp-level SpTRSV kernel.

```
1:  for ti = 0 to 31 in parallel do
2:      vi ← ti%2
3:      ri ← ti2
4:      while graphindegree[ri] ≠ 1 do
5:          //busy wait
6:      end while
7:      x[ri] ← (b[ri]-left_sum[ri])/val[cscColPtr[ri]]
8:      for j = cscColPtr[ri]+1+vi to cscColPtr[ri + 1]−1 in parallel do
9:          ATOMIC-ADD(&left_sum[cscRowIdx[j]], cscVal[j] × x[ri])
10:         ATOMIC-DECR(&graphindegree[cscRowIdx[j]])
11:         j ← j+2
12:     end for
13: end for
```

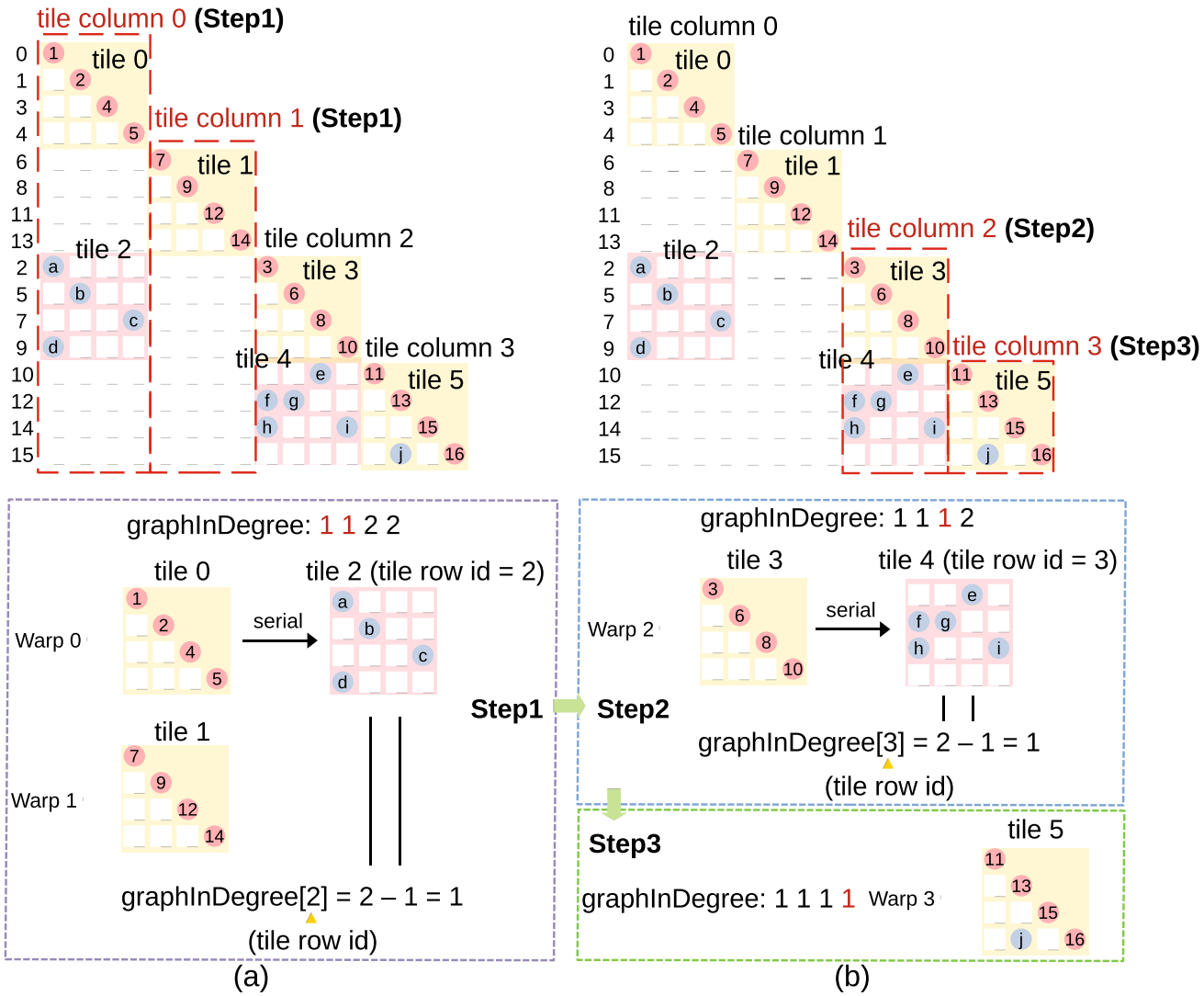

Fig. 5 An example shows that the calculation process of `TileSpTRSV_sync-free`. In this algorithm, the tile-level information is stored in the CSC format, which includes the arrays *tileColPtr*, *tileRowIdx*, and *tileNnz*. **a** explains the first calculation step, while **b** shows the second and third calculation steps. The *graphinDegree* array records the number of sparse tiles in each tile row, and the corresponding value is updated immediately after processing each off-diagonal tile

3.5 TileSpTRSV_level-set

We first implement a tiled SpTRSV algorithm called `TileSpTRSV_level-set` that is built on top of the level-set algorithm. Given an input triangular sparse matrix, the algorithm first sorts its components, i.e., rows and columns, according to their level-set order. This ensures that components in the same level-set are grouped together, leading to better cache utilization. The algorithm then divides the matrix into many sparse tiles of size 16-by-16 and uses the 2-level storage structure proposed in Sect. 3.2 to store tile information.

Then it divides these sparse tiles into multiple sets using the level-set algorithm and treats each tile as a component. This generates three important data structures: *levelPtr_tile*, *levelItem_tile*, and *nlevel*, which are crucial for later computations. We define the average tile row number in one set as tile-level parallelism and design an adaptive SpTRSV kernel selection strategy for `TileSpTRSV_level-set`. This strategy can select either thread-level or warp-level SpTRSV kernel for the CSR format based on tile-level parallelism. We use a threshold *thre* and require the GPU core count to be eight times *thre*. For example, on an RTX 4090 with 16384 CUDA cores, *thre* should be set to 2048. If the tile-level parallelism is less than *thre*, indicating unsatisfactory parallelism, we use the warp-level SpTRSV kernel; otherwise, we use the thread-level SpTRSV kernel. Figure 4 illustrates an example of `TileSpTRSV_level-set` with the thread-level SpTRSV kernel.

For each set, we first assign one warp (32 threads) to perform SpMV operation for the off-diagonal tiles in one tile row. Note that one warp can compute at most 16 tiles in one tile row to ensure load balancing in this work. After completing the calculations of all off-diagonal tiles within the set, we assign one thread to execute the SpTRSV operation for each diagonal tile in parallel. Once all the diagonal tiles have been processed, the calculations for this set are finished. Figure 4 shows an example of the partition and calculation process of the `TileSpTRSV_level-set` algorithm.

Figure 4(a) shows an input matrix. Figure 4(b) shows the matrix after reordering and partitioning according to the `TileSpTRSV_level-set` algorithm, and Fig. 4(c) shows the calculation steps of the algorithm. In Fig. 4(b), all the yellow tiles (diagonal tiles) are stored in CSR or DIA format, and the red tiles (off-diagonal tiles) are stored in a suitable format based on the format selection strategy. As we can see, the six off-diagonal sparse tiles are divided into three levels, and our `TileSpTRSV_level-set` processes them from *level*1 to *level*3 in sequence as shown in Fig. 4(c). Since there are no off-diagonal tiles to do SpMV operation in *level*1, `TileSpTRSV_level-set` can directly use two threads to process *tile*0 and *tile*1 in parallel. Each thread can execute a serial SpTRSV operation for each tile. However, for *level*2, `TileSpTRSV_level-set` will do SpMV for *tile*2 (off-diagonal tile) first and launch one warp to execute the corresponding SpMV kernel. After the calculation of *tile*2, the diagonal tile *tile*3 can be calculated using one thread to execute the SpTRSV kernel. Finally, in *level*3, all off-diagonal tiles (only *tile*4 here) can be processed using the corresponding SpMV kernel. Then, the diagonal tile (*tile*5) can be processed by a thread-level SpTRSV kernel. The calculation process of *level*3 is similar to *level*2. At this point, all calculations in the example are completed.

Algorithm 6 A pseudocode of TileSpTRSV_level-set algorithm.

```
1: function PREPROCESS-TILESPTRSV_LEVEL-SET()
2:     for li = 0 to tilen − 1 do
3:         for tilerow = 0 to tilen − 1 do
4:             if DEPENDENCIES_TILE(tilerow)= 0 then
5:                 levelptr_tile[li]++
6:                 INSERT(levelitem_tile, tilerow)
7:             end if
8:         end for
9:     end for
10:    PREFIX-SUM(levelptr_tile, tilen + 1)
11: end function
12: function TILESPTRSV_LEVEL-SET()
13:    for li = 0 to nlevel − 1 do
14:        SPMV-TILE-WARPLEVEL(levelptr_tile, levelitem_tile, tilePtr,
    tileColIdx)
15:        //global synchronization
16:        SPTRSV-TILE-THREADLEVEL(levelptr_tile, levelitem_tile, tilePtr,
    tileColIdx)
17:    end for
18: end function
```

Table 1 The GPU and four algorithms evaluated

NVIDIA GPU	Four algorithms
GeForce RTX 4090 (Ada Lovelace)	(1) cuSPARSE
16384 CUDA cores @ 2595 MHz	(2) Sync-free (Liu et al. 2017)
24GB, B/W 984 GB/s	(3) Recblock (Lu et al. 2020)
	(4) TileSpTRSV (this work)

Algorithm 6 shows a pseudocode for the `TileSpTRSV_level-set` algorithm, which includes preprocessing and calculation phases. The preprocessing function (lines 1–11) generates the auxiliary arrays *levelPtr_tile* and *levelItem_tile*. The calculation function (lines 12–20) describes the solving process. For each level, the algorithm calls a tile-level SpMV kernel (line 14) to perform SpMV operations for the off-diagonal tiles first. After global synchronization, it then calls a tile-level SpTRSV kernel (line 18) to perform SpTRSV operations for the diagonal tiles.

3.6 TileSpTRSV_sync-free

We also implement another tiled SpTRSV algorithm called `TileSpTRSV_sync-free` on top of the Sync-free algorithm. This algorithm uses the same partition method (dividing the input matrix into a number of sparse tiles) and format selection strategy, but has a different calculation mode compared to `TileSpTRSV_level-set`. Additionally, the tile-level information needs to be converted to the CSC format, which contains the *tileColPtr*, *tileRowIdx*, and *tileNnz* arrays.

Figure 5 shows an example of the calculation process using this algorithm. As shown in Fig. 5, there are six off-diagonal sparse tiles after partitioning, and four diagonal tiles (in yellow) are stored in the CSR or DIA format, while the other two tiles (in red) are stored in another formats. In this algorithm, there is an auxiliary array called *graphinDegree*, which records the number of off-diagonal sparse tiles in each tile row. For example, the matrix in Fig. 5(a) has only one tile in both *tilerow*0 and *tilerow*1, and it has two tiles in both *tilerow*2 and *tilerow*3. Therefore, the initial values of *graphinDegree* in the example are '1', '1', '2', '2'.

After we obtain the array *graphinDegree*, we assign one warp for each tile column, and each warp constantly accesses the corresponding value of *graphinDegree* until the value becomes '1'. Then the warp can process the tiles in the corresponding tile column. For example, in Fig. 5(a), we assign four warps for the four tile columns and use three steps to process the calculations. In step 1, because the 0-th value and 1-st value in *graphinDegree* are '1', *warp*0 and *warp*1 can process the tiles in *tile column* 0 (*tile*0 and *tile*2) and *tile column* 1 (*tile*1), respectively. It should be noted that diagonal tiles, such as *tile*0 and *tile*1, will perform SpTRSV operation using one thread of the corresponding warp, whereas off-diagonal tile, such as *tile*2, will be processed for SpMV operation using an entire warp (32 threads). When the warp finishes the execution of the off-diagonal tile, the *i*-th value of *graphinDegree* will decrease by 1 (assuming the tile row index of the tile is *i*). For example, in Fig. 5(a), when the execution of *tile*2 is complete, the third value of *graphinDegree* ('2') will become '1'. After the execution of *tile column* 0 and *tile column* 1, step 2 can begin, and *warp*2 will start to process the tiles in *tile column* 2 because the corresponding value of *graphinDegree* has become '1', as shown in Fig. 5(b). Then the 3-rd value of *graphinDegree* becomes '1' due to the completion of *tile*4. Therefore, *warp*3 will repeat the process like the other warps have done and process the last diagonal tile (*tile*5), in the step 3. At this point, all calculations in the example of Fig. 5 are completed.

Algorithm 7 A pseudocode of TileSpTRSV_sync-free algorithm.

```
1:  function PREPROCESS-TILESPTRSV_SYNC-FREE()
2:      for i = 0 to tilennz − 1 in parallel do
3:          ATOMIC-INCR(&graphindegree[tileRowIdx[i]])
4:      end for
5:  end function
6:  function TILESPTRSV_SYNC-FREE()
7:      for i = 0 to tilen − 1 in parallel do
8:          while graphindegree[i] ≠ 1 do
9:              //busy wait
10:         end while
11:         SPTRSV-TILE-THREADLEVEL(i, tilePtr, tileRowIdx)
12:         SPMV-TILE-WARPLEVEL(i, tilePtr, tileRowIdx)
13:     end for
14: end function
```

Table 2 The matrix information of the 16 representative matrices

Matrix	Plot	n	nnz	Tile-level parallelism
ins2		309K	1.5 M	2.4K
boyd2		466K	1 M	15K
blockqp1		60K	360K	0.1K
hangGlider_4		15K	86k	0.2K
a2nnsnsl		80K	231K	1.2K
road_usa		24 M	53 M	21K
bloweybl		30K	80K	0.6K
analytics		304K	1.2 M	1.5K
t3dl_e		20K	20K	1.3K
c-big		375K	1.3 M	11K
circuit5M		5.5 M	32 M	21K
hugebubbles-00000		18 M	45 M	21K
bcsstm39		47K	47K	2.9K
c-66b		50k	274k	1.6K
m3plates		11K	11K	0.7K
ASIC_680K		683K	1.5 M	1.1K

Algorithm 7 shows a pseudocode of `TileSpTRSV_sync-free`. The preprocessing function (lines 1–5) generates the auxiliary array *grapginDegree* for solving phase. After that, in the solving function (lines 6–14), the programming will call tile-level SpTRSV kernel (line 11) to process diagonal tile in each tile column first, then call tile-level SpMV kernel (line 12) to process remaining off-diagonal tiles.

4 Experimental results

4.1 Experimental setup

In our experiment, we use a modern NVIDIA GPU: GeForce RTX 4090 as our experimental platform. The GPU driver version is 520.56.06, and the CUDA version is 11.8. The specification of the GPU is listed in Table 1.

We compare our TileSpTRSV with three existing SpTRSV algorithms, i.e., kernel cusparseSpSV_solve in cuSPARSE v2 of CUDA v11.8, Sync-free algorithm (Liu et al. 2017) and recursive block algorithm (Lu et al. 2020) on the 16 representative matrices (refer to Table 2).

4.2 Comparison of two implements for TileSpTRSV

Firstly, we compare the two implementations of TileSpTRSV on 16 representative matrices. Table 2 lists the information of the tested matrices, where tile-level parallelism is the average tile row number of one set in `TileSpTRSV_level-set` algorithm. Figure 6 shows the comparison result. As we can see, `TileSpTRSV_level-set` (red bars) achieves higher performance compare to `TileSpTRSV_sync-free` (blue bars) on most matrices of the 16 matrices. The performance of `TileSpTRSV_level-set` can be up to 4.31x faster than `TileSpTRSV_sync-free` on matrix 'boyd2'. Though on the matrices with lower tile-level parallelism (like matrix 'bloweybl' and 'blockqp1'), `TileSpTRSV_level-set` can achieve better performance than `TileSpTRSV_sync-free`, it indicates our adaptive SpTRSV kernel selection strategy is effective to improve performance while the tile-level parallelism is unsatisfactory.

4.3 Performance comparison over existing SpTRSV works

Figure 7 presents the performance comparison of our `TileSpTRSV_level-set` algorithm with three other SpTRSV algorithms on the 16 representative matrices. Our algorithm achieves the best performance on 10 matrices and the second-best performance on the remaining 6 matrices. On average, it provides speedups of 5.29×, 5.33×, and 2.62× over cuSPARSE, Sync-free, and Recblock algorithms, respectively, for these tested matrices. The best speedups are observed on matrix 'ins2' (38.10×), 'boyd2' (21.32×), and 'blockqp1' (12.87×), respectively.

Our algorithm achieves satisfactory performance and speedups on the 'boyd2' matrix, reaching 35.19 GFlops and providing speedups of 3.75×, 21.3×, and 1.61× over cuSPARSE, Sync-free, and Recblock algorithms, respectively. The high performance is due to the high tile-level parallelism of 15K in this matrix, which allows our algorithm to obtain high parallelism to improve performance. Additionally, there are multiple off-diagonal sparse tiles in Dns, DnsRow, or DnsCol format in 'boyd2', allowing our algorithm to exploit more efficient SpMV kernels to process them. For example, there are 11,654 off-diagonal tiles stored in DnsCol format in it. Our algorithm generally has a performance advantage in handling matrices with a large proportion of Dns or DnsRow/DnsCol tiles. It also can achieve comparable performance to the three algorithms on matrices with a moderate number of

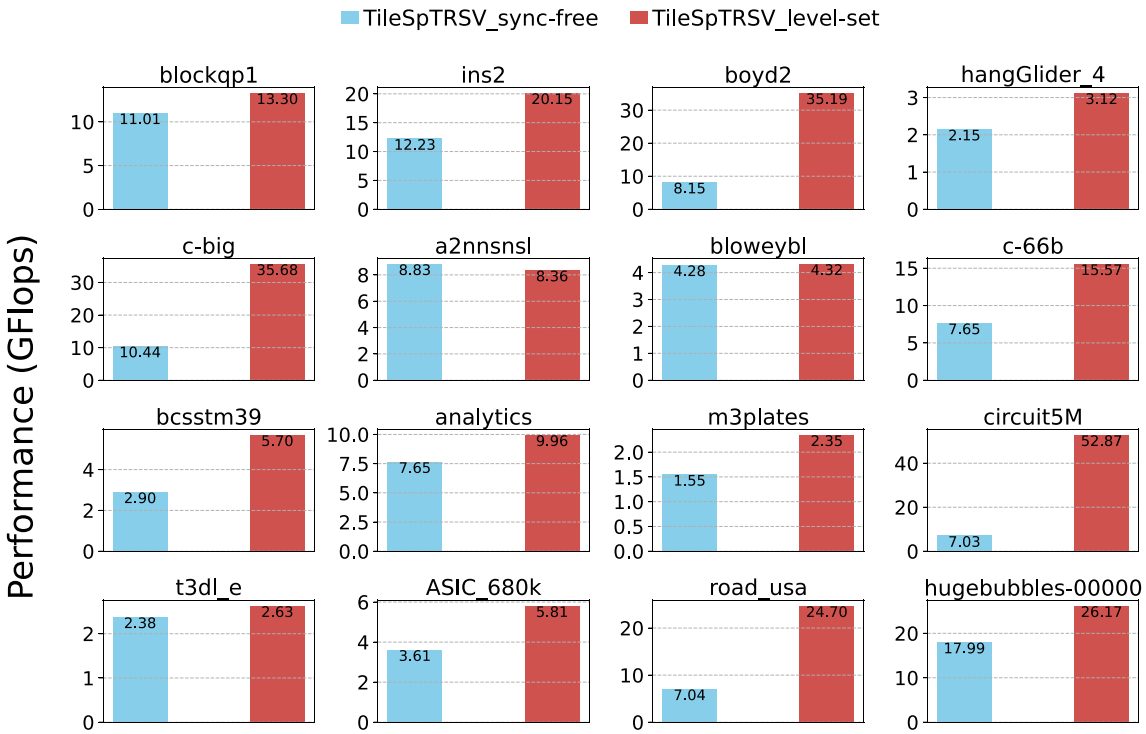

Fig. 6 The performance comparison of the two implementations of TileSpTRSV on the 16 representative matrices

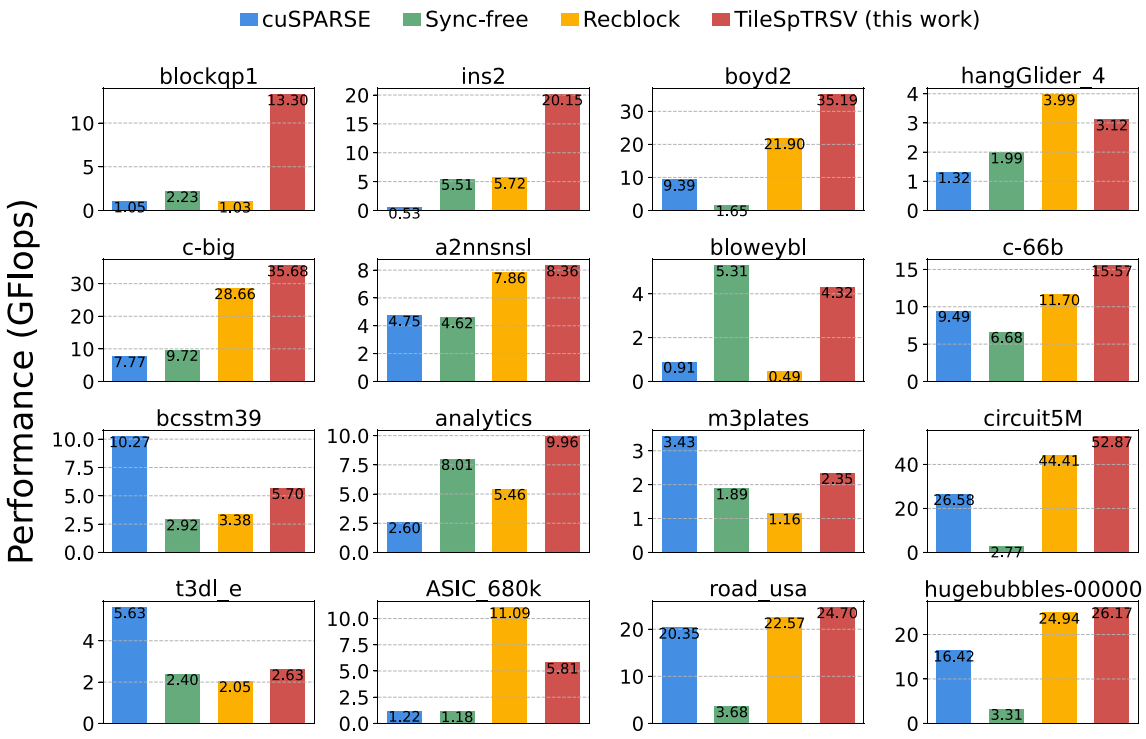

Fig. 7 The performance comparison of TileSpTRSV and the other three SpTRSV algorithms on the 16 representative matrices

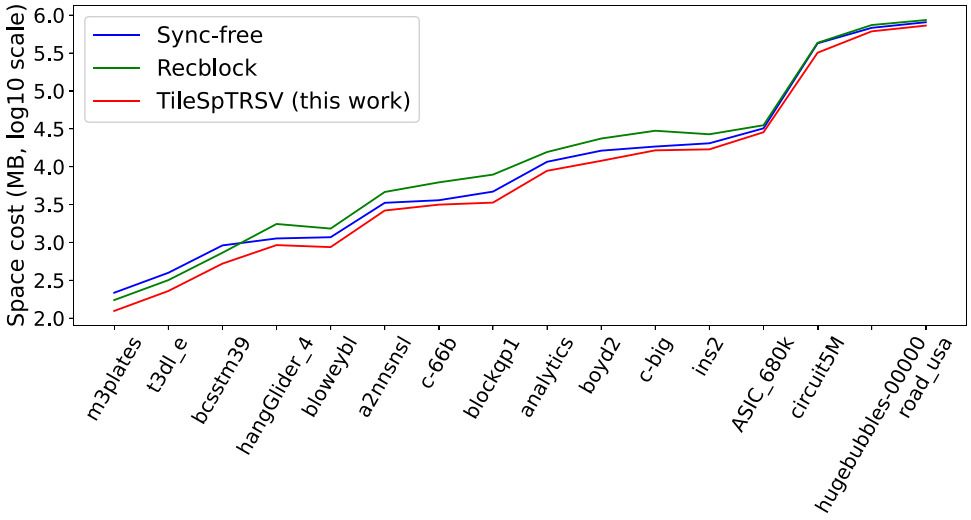

Fig. 8 The memory consumption comparison of TileSpTRSV and the other two existing SpTRSV algorithms

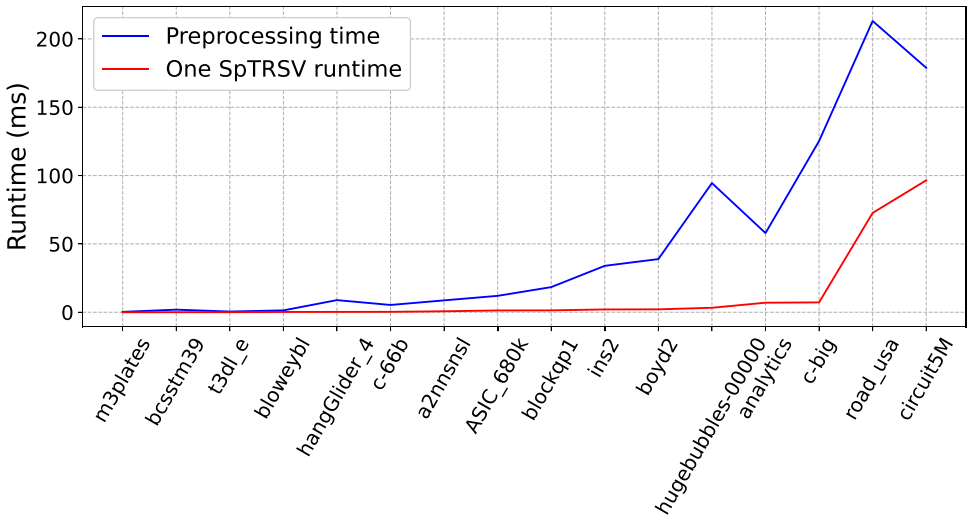

Fig. 9 Comparison of preprocessing time and a single SpTRSV time of the 16 representative matrices

CSR and COO tiles, such as the 'bloweybl' matrix with 3127 CSR tiles and 1872 COO tiles. It should be noticed that we only use the COO or CSR format when the internal structure of the tile is sparse enough and irregular which can help our algorithm save more space compare to other works.

4.4 Space cost comparison

We evaluate the memory consumption of Sync-free, Recblock and our TileSpTRSV, and Fig. 8 shows the space cost of the data structure of input matrix on the 16 representative matrices. Note that the two implementations of TileSpTRSV have the same storage structure and same memory consumption. As we can see, the memory consumption of our TileSpTRSV is in general less than other two algorithms on the 16 matrices. It is because that TileSpTRSV uses a 8-bit 'unsigned char' instead of a 32-bit 'int' to store the information of one tile of 16-by-16 size. When the tiles are stored in Dns, DnsRow, DnsCol and DIA formats, we only use one array to store the value of nonzeros in the sparse tiles.

4.5 Preprocessing overhead

Figure 9 shows an execution time comparison of the preprocessing and the single SpTRSV time on CPU. It should be noticed that our two implementations of TileSpTRSV have the same preprocessing phase, so that their execution time of this phase is same. We can see that the preprocessing time is less than 10× SpTRSV execution time on most matrices of our dataset. Since we can do many times SpTRSV iterations after one preprocessing operation, the cost of this preprocessing is acceptable.

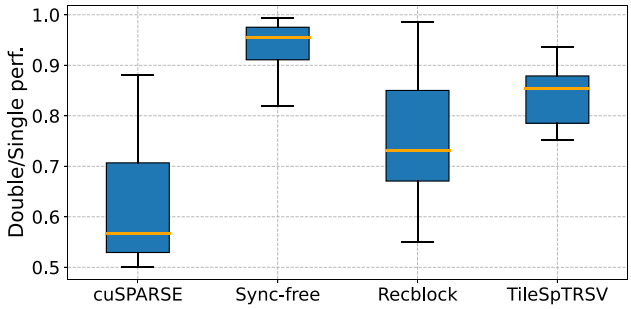

Fig. 10 Performance ratio of double precision to single precision in box plots of running the four SpTRSV algorithms on the 16 matrices on the RTX 4090

4.6 Performance of different precision

In our experiments plotted above (Figs. 6, 7, 8, and 9), the experimental data are in double precision. We also test the performance of `TileSpTRSV_level-set` and the other three existing algorithms in float precision and compare with that in double precision. The boxes ploted in Fig. 10 show the comparison result. As can be seen, the ratio of double and single precision performance of Sync-free is between 0.9 and 1.0, the ratio of RecBlock is between 0.7 and 0.8, and that of our algorithm is between 0.8 and 0.9, while the ratio of cuSPARSE is between 0.5 and 0.6. This demonstrates that compared to the Sync-free algorithm, our algorithm is a bit more sensitive to the floating-point value used, but compared to the cuSPARSE and Recblock algorithms, our algorithm is relatively insensitive to the operation of floating-point values.

5 Related work

SpTRSV is a crucial building block of the sparse BLAS (Liu 2015). Most studies on parallel SpTRSV algorithms focus on level-set, synchronization-free, color-set, and block methods.

To make SpTRSV parallelizable, Anderson and Saad (1989) and Saltz (1990) propose a classical parallel SpTRSV algorithm called the level-set algorithm. This algorithm divides all components into a number of level-sets. The components within each set can be solved in parallel, but the sets should be processed sequentially due to their dependencies on each other. Many studies focus on optimizing this algorithm through obtaining better data locality and reducing the cost of synchronization. Naumov (2011) implement the GPU version of level-set and decrease the number of sets. Park et al. (2014) reduce the dependencies in level-set algorithm. With the development of GPU general computing, Li and Saad (2013) reduce the number of level-sets and achieve better parallelism by exploiting topological sorting. Xie et al. (2021) implement level-set algorithm for SpTRSV on modern multi-GPU and obtain the performance gain.

Although level-set algorithm on GPU can achieve great parallelism and obtain performance gain, the cost of synchronization between kernel calls is expensive. To address the problem, Liu et al. (2016, 2017) propose synchronization-free algorithm which uses fast atomic operation instead of expensive cost of global synchronization and apply it to parallel multiply right-hand sides. In order to make the algorithm available for the CSR format which is the most popular sparse storage format, Dufrechou and Ezzatti (2018b, 2018a) implement synchronization-free algorithm in the CSR format. Su et al. (2020) exploit large-scale thread-level parallelism for faster synchronization-free algorithm on modern GPUs. Zhang et al. (2021) improve the performance of Sync-free algorithm by fusing thread-level and warp-level techniques.

Schreiber and Tang (1982) first propose colour-set algorithm which uses graph colouring to implement parallel SpTRSV algorithm. In this algorithm, after the input matrix has been coloured, the components in each colour-set can be regarded as a diagonal block which means they can be processed in parallel. Suchoski et al. (2012) implement the GPU version of the algorithm. Naumov et al. (2015) demonstrate the effectiveness of colour-set algorithm to accelerate parallel SpTRSV on GPU. Besides, Kabir et al. (2015) use graph colouring to optimize the performance of SpTRSV on NUMA architectures. But it is well known that graph colouring is an NP-completed problem. Its preprocessing overhead in real-world applications is often unacceptable.

There are also some studies that use block algorithm to accelerate parallel SpTRSV. Mayer (2009) first point out that 2D blocking should be able to accelerate SpTRSV. Wang et al. (2018a, 2018b) propose a novel data layout called Sparse Level Tile and design a Producer-Consumer pairing method for structured problems on Sunway processors. Vuduc et al. (2002) and Bradley (2016) develop blocking schemes for SpTRSV. Lu et al. (2020) design a recursive blocking algorithm and utilize a new data structure to store the input matrix. Ahmad et al. (2021) accelerate parallel SpTRSV calculations by dividing them into two SpTRSV and one SpMV systems and employing different algorithms for each SpTRSV system.

There are a number of studies on block/tile optimization for other BLAS kernels. Buttari et al. (2007) design the BCSR format for SpMV. On GPUs, Choi et al. (2010) use blocked formats to model SpMV, and Yan et al. (2014) develop the BCCOO format, which stores dense 2D blocks. Additionally, Niu et al. (2021, 2022) propose a tiled SpMV algorithm and a tiled SpGEMM algorithm called TileSpMV and TileSpGEMM, respectively. Ji et al. (2022) further developed TileSpMSpV for multiplying sparse matrix and sparse vector. Such work motivates us to design the TileSpTRSV algorithm using

the tiled structure in this work for SpTRSV to obtain performance gain and enrich tiled algorithms for sparse BLAS.

6 Conclusion

In this paper, we implement two versions of TileSpTRSV algorithm, namely `TileSpTRSV_level-set` on top of level-set algorithm, and `TileSpTRSV_sync-free` on top of Sync-free algorithm, respectively, on NVIDIA GPUs. We also design an adaptive SpTRSV kernel selection strategy for `TileSpTRSV_level-set` to use different SpTRSV algorithms according to the tile-level parallelism of the input matrix. In the experiment, we select 16 representative matrices from the SuiteSparse Matrix Collection and evaluate TileSpTRSV, cuSPARSE, Sync-free and Recblock algorithms on a modern NVIDIA GPU: GeForce RTX 4090. The experimental results show that our TileSpTRSV achieves on average of 5.29× (up to 38.10×), 5.33× (up to 21.32×), and 2.62× (up to 12.87×) speedups over cuSPARSE, Sync-free and Recblock algorithms on the 16 representative matrices, respectively.

Acknowledgements We deeply appreciate the invaluable comments from all the reviewers. We are also so grateful to Hemeng Wang for the help in the experimental test. Weifeng Liu is the corresponding author of this paper. This research was supported by the National Natural Science Foundation of China under Grant No. 61972415.

References

Ahmad, N., Yilmaz, B., Unat, D.: A split execution model for sptrsv. IEEE Trans. Parallel Distrib. Syst. **32**(11), 2809–2822 (2021)

Anderson, E., Saad, Y.: Solving sparse triangular linear systems on parallel computers. Int. J. High Speed Comput. **1**(1), 73–95 (1989)

Anzt, H., Chow, E., Dongarra, J.: Iterative sparse triangular solves for preconditioning. In: Euro-Par '15. p 650–661 (2015)

Anzt, H., Chow, E., Szyld, D.B., et al.: Domain overlap for iterative sparse triangular solves on GPUs. Softw. Exascale Comput. SPPEXA **2013–2015**, 527–545 (2016)

Anzt, H., Chow, E., Dongarra, J.: ParILUT—a new parallel threshold ILU factorization. SIAM J. Sci. Comput. **40**(4), C503–C519 (2018a)

Anzt, H., Huckle, T., Brackle, J., et al.: Incomplete sparse approximate inverses for parallel preconditioning. Parallel Comput. **71**, 1–22 (2018b)

Bradley, A.M.: A hybrid multithreaded direct sparse triangular solver. In: SIAM CSC workshop '16, pp 13–22 (2016)

Buttari, A., Eijkhout, V., Langou, J., et al.: Performance optimization and modeling of blocked sparse kernels. Int. J. High Perform. Comput. Appl. **21**(4), 467–484 (2007)

Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on gpus. In: PPoPP '10, pp 115–126 (2010)

Davis, T.: Direct methods for sparse linear systems. Society for Industrial and Applied Mathematics (2006)

Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 11–125 (2011)

Duff, I.S., Erisman, A.M., Reid, J.K.: Direct methods for sparse matrices, 2nd edn. Oxford University Press, Inc, Oxford (2017)

Dufrechou, E., Ezzatti, P.: A new GPU algorithm to compute a level set-based analysis for the parallel solution of sparse triangular systems. In: IPDPS '18, pp 920–929 (2018a)

Dufrechou, E., Ezzatti, P.: Solving sparse triangular linear systems in modern GPUs: a synchronization-free algorithm. In: PDP '18, pp 196–203 (2018b)

Hou, K., Liu, W., Wang, H., et al. Fast segmented sort on GPUs. In: ICS '17, pp 12:1–12:10 (2017)

Ji, H., Song, H., Lu, S., et al. Tilespmspv: a tiled algorithm for sparse matrix-sparse vector multiplication on gpus. In: ICPP '22 (2022)

Kabir, H., Booth, J.D., Aupy, G., et al.: STS-k: A multilevel sparse triangular solution scheme for NUMA multicores. In: SC '15, pp 55:1–55:11 (2015)

Li, X.S.: An overview of SuperLU: algorithms, implementation, and user interface. ACM Trans. Math. Softw. **31**(3), 302–325 (2005)

Li, R., Saad, Y.: GPU-accelerated preconditioned iterative linear solvers. J. Supercomput. **63**(2), 443–466 (2013)

Liu, W.: Parallel and scalable sparse basic linear algebra subprograms. PhD thesis, University of Copenhagen (2015)

Liu, W., Li, A., Hogg, J., et al.: A synchronization-free algorithm for parallel sparse triangular solves. In: Euro-Par '16, pp 617–630 (2016)

Liu, W., Vinter, B.: A framework for general sparse matrix-matrix multiplication on GPUs and heterogeneous processors. J. Parallel Distrib. Comput. **85**(C), 47–61 (2015a)

Liu, W., Vinter, B.: CSR5: an efficient storage format for cross-platform sparse matrix-vector multiplication. In: ICS '15, pp 339–350 (2015b)

Liu, W., Vinter, B.: Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. Parallel Comput. **49**(C), 179–193 (2015c)

Liu, W., Li, A., Hogg, J.D., et al.: Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. Concurr. Comput. Pract. Exp. **29**(21), e4244 (2017)

Liu, J., He, X., Liu, W., et al.: Register-aware optimizations for parallel sparse matrix-matrix multiplication. Int. J. Parallel Program. **47**, 403–417 (2019)

Lu, Z., Niu, Y., Liu, W.: Efficient block algorithms for parallel sparse triangular solve. In: ICPP '20, pp 1–11 (2020)

Mayer, J.: Parallel algorithms for solving linear systems with sparse triangular matrices. Computing **86**(4), 291–312 (2009)

Naumov, M.: Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. Tech. rep, NVIDIA (2011)

Naumov, M., Castonguay, P., Cohen, J.: Parallel graph coloring with applications to the incomplete-LU factorization on the GPU. Nvidia White Paper (2015)

Niu, Y., Lu, Z., Dong, M., et al.: Tilespmv: a tiled algorithm for sparse matrix-vector multiplication on gpus. In: IPDPS '21, IEEE, pp 68–78 (2021)

Niu, Y., Lu, Z., Ji, H., et al.: Tilespgemm: a tiled algorithm for parallel sparse general matrix-matrix multiplication on gpus. In: PPoPP '22, pp 90–106 (2022)

Park, J., Smelyanskiy, M., Sundaram, N., et al.: Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In: ISC '14, pp 124–140 (2014)

Saltz, J.H.: Aggregation methods for solving sparse triangular systems on multiprocessors. SIAM J. Sci. Stat. Comput. **11**(1), 123–144 (1990)

Schreiber, R., Tang, W.P.: Vectorizing the conjugate gradient method. In: Proceedings of the Symposium on CYBER 205 Applications (1982)

Su, J., Zhang, F., Liu, W., et al.: CapelliniSpTRSV: a thread-level synchronization-free sparse triangular solve on GPUs. In: ICPP '20 (2020)

Suchoski, B., Severn, C., Shantharam, M., et al.: Adapting sparse triangular solution to GPUs. In: ICPPW '12, pp 140–148 (2012)

Vuduc, R., Kamil, S., Hsu, J., et al.: Automatic performance tuning and analysis of sparse triangular solve. In: ICS '02 Workshop (2002)

Wang, X., Liu, W., Xue, W., et al.: SwSpTRSV: a fast sparse triangular solve with sparse level tile layout on sunway architectures. In: PPoPP '18, p 338-353 (2018a)

Wang, X., Xu, P., Xue, W., et al.: A fast sparse triangular solver for structured-grid problems on sunway many-core processor SW26010. In: ICPP '18 (2018b)

Wang, T., Li, W., Pei, H., et al.: Accelerating sparse lu factorization with density-aware adaptive matrix multiplication for circuit simulation. In: DAC '23 (2023)

Xie, Z., Tan, G., Liu, W., et al.: IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In: ICS '19, pp 94–105 (2019)

Xie, C., Chen, J., Firoz, J., et al.: Fast and scalable sparse triangular solver for multi-gpu based hpc architectures. In: ICPP '21, pp 1–11 (2021)

Yan, S., Li, C., Zhang, Y., et al. (2014) yaspmv: yet another spmv framework on gpus. In: PPoPP '14, pp 107–118 (2021)

Zhang, F., Su, J., Liu, W., et al.: Yuenyeungsptrsv: a thread-level and warp-level fusion synchronization-free sparse triangular solve. IEEE Trans. Parallel Distrib. Syst. **32**(9), 2321–2337 (2021)

Zhao, J., Wen, Y., Luo, Y., et al.: Sflu: Synchronization-free sparse lu factorization for fast circuit simulation on gpus. In: DAC '21, pp 37–42 (2021)