Efficient Block Algorithms for Parallel Sparse Triangular Solve

Zhengyang Lu
Super Scientific Software Laboratory, Department of Computer Science and Technology, China University of Petroleum, Beijing, China
2017010055@student.cup.edu.cn

Yuyao Niu
Super Scientific Software Laboratory, Department of Computer Science and Technology, China University of Petroleum, Beijing, China
2019211256@student.cup.edu.cn

Weifeng Liu
Super Scientific Software Laboratory, Department of Computer Science and Technology, China University of Petroleum, Beijing, China
weifeng.liu@cup.edu.cn

ABSTRACT

The sparse triangular solve (SpTRSV) kernel is an important building block for a number of linear algebra routines such as sparse direct and iterative solvers. The major challenge of accelerating SpTRSV lies in the difficulties of finding higher parallelism. Existing work mainly focuses on reducing dependencies and synchronizations in the level-set methods. However, the 2D block layout of the input matrix has been largely ignored in designing more efficient SpTRSV algorithms.

In this paper, we implement three block algorithms, i.e., column block, row block and recursive block algorithms, for parallel SpTRSV on modern GPUs, and propose an adaptive approach that can automatically select the best kernels according to input sparsity structures. By testing 159 sparse matrices on two high-end NVIDIA GPUs, the experimental results demonstrate that the recursive block algorithm has the best performance among the three block algorithms, and it is on average 4.72x (up to 72.03x) and 9.95x (up to 61.08x) faster than cuSPARSE v2 and Sync-free methods, respectively. Besides, our method merely needs moderate cost for preprocessing the input matrix, thus is highly efficient for multiple right-hand sides and iterative scenarios.

CCS CONCEPTS

• **Mathematics of computing** → **Solvers**; **Mathematical software performance**; • **Computing methodologies** → **Shared memory algorithms**; **Vector / streaming algorithms**.

KEYWORDS

sparse matrix, sparse triangular solve, block algorithm, GPU

ACM Reference Format:
Zhengyang Lu, Yuyao Niu, and Weifeng Liu. 2020. Efficient Block Algorithms for Parallel Sparse Triangular Solve. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404413

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404413

1 INTRODUCTION

The sparse triangular solve (SpTRSV) operation solves a linear system of the form $Lx = b$ (or $Ux = b$), where L (or U) is a sparse lower (or upper) triangular matrix, b is a dense right-hand side vector, and x is the dense resulting vector to solve. In a number of algorithms for linear solvers, the SpTRSV kernel is extensively used for completing the solve phase of sparse direct solvers [1, 4, 5, 27, 28, 33, 38, 46] and for accelerating convergence of preconditioned sparse iterative solvers [8, 9, 11, 13, 25, 45, 62, 73].

Compared with other sparse basic linear algebra subprograms (BLAS) [48] such as sparse transposition [77], sparse matrix-vector multiplication (SpMV) [52, 53] and sparse matrix-matrix multiplication (SpGEMM) [41, 47, 51, 83], SpTRSV is in general more difficult to parallelize due to the possible dependencies between the components of the solution vector. Such dependencies lead to that solving a component x_i may have to wait until its previous components $x_0, \ldots, x_{i-1}$ have been solved. Such inherent sequential execution makes SpTRSV one of the most crucial performance bottlenecks of direct solvers with multiple right-hand sides [30, 54, 65] and incomplete factorization preconditioners [13, 26, 45].

Fortunately, much research has demonstrated that it is possible to parallelize SpTRSV. Anderson and Saad [7] and Saltz [64] have seen SpTRSV as a graph problem, and proposed the level-set methods that divide the components into multiple sets in which components can be solved in parallel. But because of the dependencies between the sets, costly global barrier synchronizations have to be added between the process of the level-sets. To reduce the overhead of the synchronizations, Park et al. [60] developed point-to-point synchronizations for CPUs, and Liu et al. [49, 50] proposed synchronization-free algorithms for replacing the synchronizations with atomic operations on GPUs. Besides, several colour-set methods [43, 59, 67] and iterative methods [8, 11, 25] have also been developed for achieving higher parallelism.

Moreover, another direction of research focused on exploiting the spatial structure of the input matrix. The basic idea is that a triangular matrix can be divided into multiple smaller triangular sub-matrices and rectangular/square sub-matrices. The triangular ones still need triangular solves, but the rectangular/square parts could be processed by matrix-vector multiplication with possibly much better parallelism. For better accelerating dense triangular solve on GPUs, Hogg [40] and Charara et al. [20, 21] demonstrated that column and recursive block formulations perform very well due to higher parallelism from dense matrix multiplications.

However, for the sparse cases, the block methods have not been seen as an effective way to improve SpTRSV. Mayer [56] showed that none of the various blocking methods tested for SpTRSV can

give noticeable performance gain. The Sparse Level Tile format proposed by Wang et al. [78] utilized irregular 2D tiles and was optimized for specific register communications on Sunway processors, but is not necessarily generalizable to a wide range of parallel devices such as GPUs. Bradley [14] developed a blocking method, but only worked best for relatively dense triangular matrices factorized from LU decomposition. On distributed memory systems, Liu et al. [54] developed a new method for the 2D cyclic block layout in the SuperLU_DIST direct solver package. But its shared memory part still calls level-set methods.

Because of the aforementioned concerns, we in this paper develop efficient block algorithms for parallel SpTRSV. We first implement three block algorithms for accelerating SpTRSV on modern GPUs. The three algorithms divide the input triangular matrix into a number of column blocks (each including a triangular sub-matrix and a rectangular sub-matrix), row blocks (each including a rectangular sub-matrix and a triangular sub-matrix), and recursive blocks (each including two triangular sub-matrices and a square or near-square sub-matrix), respectively. Through a theoretical analysis and running some experiments, we find that the recursive block algorithm gives the best performance among the three block algorithms. Thus we further improve its performance by using a new data format. In addition, we propose an adaptive method that automatically selects the best SpTRSV and SpMV kernels for the divided triangular sub-matrices and square sub-matrices, respectively, depending on their sparsity structures.

The recursive block algorithm we proposed have obvious advantages over the level-set and Sync-free parallel SpTRSV methods on GPUs. We implement our approaches on NVIDIA GPUs, and compare them with the existing fastest algorithms including NVIDIA cuSPARSE v2 in CUDA 10.2 [58] and the Sync-free method [50]. We use all 159 matrices, of the number of rows/columns no less than 500,000 and of the number of nonzero entries between 5,000,000 and 500,000,000, from the SuiteSparse Matrix Collection [29] as our dataset. The GPU platforms used include an NVIDIA Titan X (Pascal) and an NVIDIA Titan RTX (Turing). The experimental results demonstrate that our method brings on average 4.72x (up to 72.03x) and 9.95x (up to 61.08x) speedups over the latest cuSPARSE v2 and Sync-free algorithms, respectively. Also, the preprocessing cost of our method takes only on average 9.16x execution time of a single SpTRSV operation. When a large amount of SpTRSV is called after the preprocessing, this cost can be easily amortized.

This work makes the following contributions:

- We implement three (i.e., column, row and recursive) block algorithms for parallel SpTRSV on GPUs.
- We propose an adaptive method to automatically select the best SpTRSV and SpMV kernels to maximize performance.
- We evaluate our approach by using 159 matrices and obtain significant speedups over cuSPARSE and Sync-free methods.

2 BACKGROUND

2.1 Preliminaries

2.1.1 Serial SpTRSV. Because of the inherently sequential characteristic, it is natural to solve a triangular system in serial. Algorithm 1 lists a typical serial implementation for solving $Lx = b$. The input matrix L is stored in the compressed sparse row (CSR) format, which consists of three arrays `row_ptr`, `col_idx` and `val`. The array `left_sum` stores the sums of the products of the components of x already calculated with their corresponding nonzero entries in the rows (lines 3–5). After that, the current $\mathtt{x}_i$ is solved by dividing ($\mathtt{b}_i$ - $\mathtt{left\_sum}_i$) with the diagonal entry (line 7). It can be seen that the solution of each single component x_i depends on its previous components $x_0, \ldots, x_{i-1}$. Therefore SpTRSV cannot be simply parallelized based on Algorithm 1.

Algorithm 1 A serial algorithm for CSR-SpTRSV.

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

2.1.2 Level-set Parallel SpTRSV. The level-set algorithm proposed by Anderson and Saad [7] and Saltz [64] is a typical method for exploiting possible parallelism in the input matrix. This method sees a matrix as a graph and divides its components into multiple sets. The components in each set do not depend on each other, so they can be solved in parallel. Algorithm 2 shows the procedure including a preprocessing stage for finding the level-sets (lines 1–11) and an execution stage for consuming the parallelizable level-sets one by one (lines 12–22). Taking the problem in Figure 1 as an example, the input matrix L of 15 nonzeros can generate a graph of four level-sets. Due to the dependencies, the components in a level must wait for the calculation of other components in its upper levels. For example, x_2, x_3 and x_4 can be solved in parallel after x_0, x_1 and x_6 are completed. In the whole process, levels 0 and 1 can utilize parallelism, and levels 2 and 3 still have to run in serial.

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

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

2.1.3 Synchronization-Free Parallel SpTRSV. The main objective of the Synchronization-free approach, or Sync-free for short, proposed by Liu et al. [49, 50] is to reduce the cost of generating level-sets (lines 1–11 in Algorithm 2) and to eliminate the costly barrier synchronizations between the sets at runtime (line 20 in Algorithm 2) by using fast atomic operations on GPUs. Algorithm 3 explains a

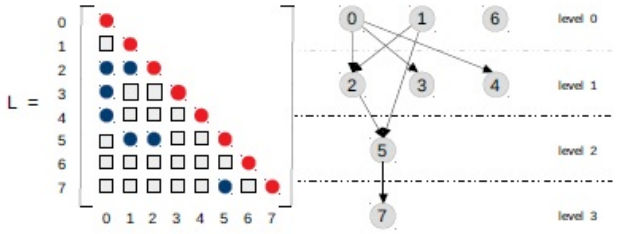


Figure 1: An example of an 8-by-8 sparse lower triangular matrix L and its corresponding level-set form.

simplified Sync-free process. Through a light-weight preprocessing stage (lines 1–5 in Algorithm 3), each component knows how many entries have to be computed in advance (i.e., the number of dependent entries, or the in-degree of the component, or the number of nonzero entries of the row excluding the diagonal entry). Then in the solve phase, each component x_i is processed by a GPU working unit (e.g., a 32-thread warp in CUDA). It firstly busy-waits until its dependencies are removed (lines 8–10), then starts to compute x_i (line 11) and notifies all the later entries that depend on x_i by atomic updates (lines 12–15). As can be seen, compared to the level-set methods calling multiple GPU kernels (lines 13–21 in Algorithm 2), Sync-free method only issues one GPU kernel (lines 6–17 in Algorithm 3), and no explicit synchronizations are needed. Note that Algorithm 3 processes the input matrix in the compressed sparse column (CSC) format, and a CSR version of the Sync-free method is given by Dufrechou and Ezzatti [36].

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

```
1:  function PREPROCESS-SYNCFREE()
2:      for i = 0 to nnz − 1 in parallel do
3:          ATOMIC-INCR(&in_degree[row_idx[i]])
4:      end for
5:  end function
6:  function SPTRSV-SYNCFREE()
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[col_ptr[i]]
12:         for j = col_ptr[i]+1 to col_ptr[i + 1]−1 in parallel do
13:             ATOMIC-ADD(&left_sum[row_idx[j]], val[j] × x[i])
14:             ATOMIC-DECR(&in_degree[row_idx[j]])
15:         end for
16:     end for
17: end function
```

2.2 Motivation

In spite of the success in achieving parallelism and reducing synchronizations, the level-set and Sync-free algorithms overly focused on the graph form of the input matrix, but largely ignored its 2D spatial structure. As a result, some critical performance issues, such as low data locality, load imbalance and inadequate parallelism, are hard to be addressed. Hence in this paper, we see the matrix as a matrix and exploit its 2D spatial structure for improving SpTRSV performance.

Firstly, the locations of x and b can be accessed very randomly, thus level-set and Sync-free methods are likely to encounter a high cache miss rate. In particular when the input matrix becomes larger and more irregular, the cache miss rate tends to worsen [22, 44, 85, 86]. According to the research by Wolf et al. [82] and Wittmann et al. [81], bad cache locality can be the most crucial performance bottleneck of SpTRSV. Thus we aim to make the data locality better, through dividing the triangular matrix into a series of smaller triangular and rectangular/square sub-matrices. Thus the nonzero entries in each of the small parts are stored more tightly. This helps a cache of limited size to store more usable entries of x and b for better data locality.

Secondly, neither level-set nor Sync-free method considered relieving the load balancing problem. Some sparse matrices, in particular the ones from circuit simulation and network analysis problems, obey the power-law distribution, meaning that some very long rows or columns may dominate the execution time of SpTRSV. This makes the performance even worse on modern massively parallel processors such as GPUs. So we in this work will exploit 2D block layouts to naturally cut those long rows and columns into shorter segments. This can better saturate GPUs and avoid the performance degradation brought by the small number of very long rows and columns.

Thirdly, it is well known that SpMV in general has better parallelism than SpTRSV. However, if we see the input triangular matrix as a graph, it is hard to exploit the potential parallelism inside its matrix form. In studies on dense triangular solve, Hogg [40] and Charara et al. [20, 21] already utilized various 2D blocking methods and converted part of the triangular solve computations to matrix-vector multiplication. Also, it is easy to see that the finer the triangular matrix is divided, the more nonzeros will be partitioned into the rectangular/square parts. However, existing research using this spatial characteristic for sparse matrices either brings little performance gain [14, 56], or is specialized to specific architectures [78]. Therefore, we in this paper will find the best formulation from various blocking methods and their parameters, and accelerate SpTRSV on modern GPUs.

3 BLOCK ALGORITHMS

In this section, we first implement three block algorithms (i.e., column block, row block and recursive block algorithms) for parallel SpTRSV (Section 3.1). Then according to a comparison of the execution features, we find that the recursive block algorithm brings the best performance among the three approaches (Section 3.2). Therefore, we further optimize the recursive block algorithm by rearranging nonzeros (Section 3.3). Finally we introduce an adaptive method that automatically selects the best SpTRSV and SpMV kernels for executing the sub-matrices (Section 3.4).

3.1 Implementation of Three Block Algorithms

3.1.1 Column Block Algorithm. A matrix can be divided into a number of column blocks, and each one consists of a triangular submatrix on top and a rectangular sub-matrix on bottom. Figure 2(a) shows an example matrix of four column blocks, and Algorithm 4 lists a solve pseudocode of the solve kernel. The resulting vector x is solved segment by segment. Each segment x_{si} is solved by executing an SpTRSV function on the sith triangular part (line 3). Once x_{si} is solved, it is used to multiply the rectangular part with an SpMV kernel, and a new right-hand side b_{si} is generated for the

rest of the computation (lines 4–6). In the example, four SpTRSV and three SpMV kernels are called.

Algorithm 4 A column block algorithm for SpTRSV.

```
1: function SPTRSV-COLUMN-BLOCK(nseg)
2:     for si = 0 → nseg − 1 do
3:         x_si ← SPTRSV( blk-tri_si, b_si)
4:         if si ≠ nseg − 1 then
5:             b_si+1 ← SPMV(blk-rec_si, x_si, b_si)
6:         end if
7:     end for
8: end function
```

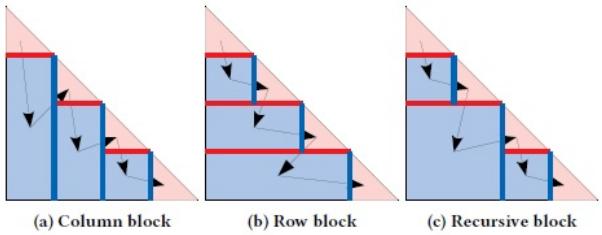


Figure 2: Three block algorithms implemented. The arrows indicate the processing order of the blocks. The bold blue and red lines means reading x and updating b, respectively.

3.1.2 Row Block Algorithm. Similar to the column block algorithm, the row block algorithm cuts a matrix horizontally. Now each row block includes a rectangular sub-matrix on the left and a triangular sub-matrix on the right. For each row block, the rectangular part is first consumed by an SpMV, then an SpTRSV is issued for the triangular part. Figure 2(b) gives an example of four row blocks, and Algorithm 5 lists a pseudocode.

Algorithm 5 A row block algorithm for SpTRSV.

```
1: function SPTRSV-ROW-BLOCK(nseg)
2:     for si = 0 → nseg − 1 do
3:         if si ≠ 0 then
4:             b_si ← SPMV(blk-rec_si, x_si−1, b_si)
5:         end if
6:         x_si ← SPTRSV( blk-tri_si, b_si)
7:     end for
8: end function
```

3.1.3 Recursive block Algorithm. Unlike the former column and row block methods, recursive block approach divides a triangular matrix into two smaller triangular and one square or near square blocks, and the two triangular sub-matrices could be further recursively divided into the three parts. For each triangular sub-matrix divided, two SpTRSV and one SpMV kernels are executed. The example matrix in Figure 2(c) is divided with a recursion depth 2, and gets four triangular and three square blocks to compute. Algorithm 6 shows a process of recursive block algorithm.

3.2 Comparison of the Three Block Algorithms

Even though the three block algorithms execute the same number of SpTRSV and SpMV kernels and process the same number of nonzero entries, the performance of them can be different. The major difference is from the SpMV part. As can be seen from the three examples in Figure 2, the total length of the bold blue lines indicates that the right-hand side b can be updated at different times, and the total length of the bold red lines shows that the solved part of the solution vector x will be loaded/cached at different times.

Algorithm 6 A recursive block algorithm for SpTRSV.

```
1: function SPTRSV-RECURSIVE-BLOCK(depth)
2:     if depth = 0 then
3:         x ← SPTRSV(tri, b)
4:     else
5:         x_depth ← SPTRSV-RECURSIVE-BLOCK(tri-top_depth, depth − 1)   ▷ Recursion
6:         b_depth ← SPMV(rec_depth, x_depth, b_depth)
7:         x_depth ← SPTRSV-RECURSIVE-BLOCK(tri-bottom_depth, depth − 1)   ▷ Recursion
8:     end if
9: end function
```

Tables 1 and 2 quantify the difference on updating b and reading x, respectively (x in Tables 1 and 2 means iterations). It is clear to see that the column block method has obvious disadvantage on updating b, and the row block method spends significant longer time on loading x. But fortunately, the recursive block algorithm achieves a good tradeoff between the column and row block methods.

Table 1: The number of items updated to right-hand side b.

Method	Formula (n is #rows)	#triangular parts divided			
		4	16	256	65536
col. block	$2^{x-1}n + 0.5n$	$2.5n$	$8.5n$	$128.5n$	$32768.5n$
row block	$2n - 2^{-x}n$	$1.75n$	$1.94n$	$1.99n$	$1.99n$
rec. block	$0.5nx + n$	$2n$	$3n$	$5n$	$9n$

Table 2: The number of items loaded from solution vector x.

Method	Formula (n is #rows)	#triangular parts divided			
		4	16	256	65536
col. block	$n - 2^{-x}n$	$0.75n$	$0.94n$	$0.99n$	$0.99n$
row block	$2^{x-1}n - 0.5n$	$1.5n$	$7.5n$	$127.5n$	$32767.5n$
rec. block	$0.5nx$	n	$2n$	$4n$	$8n$

Although the above theoretical analysis is for dense cases, the same trend is confirmed through benchmarking sparse matrices. We evaluate the three block algorithms by running two representative sparse matrices (see the third and fourth matrices in Table 4) on an NVIDIA Titan RTX GPU (see Table 3) and plot their execution time on SpMV in Figure 4. As can be seen, the SpMV kernels in recursive block algorithm almost always take less overhead than the column and row block methods. As a result, we in this paper will focus on further accelerating the recursive block data structure and algorithm.

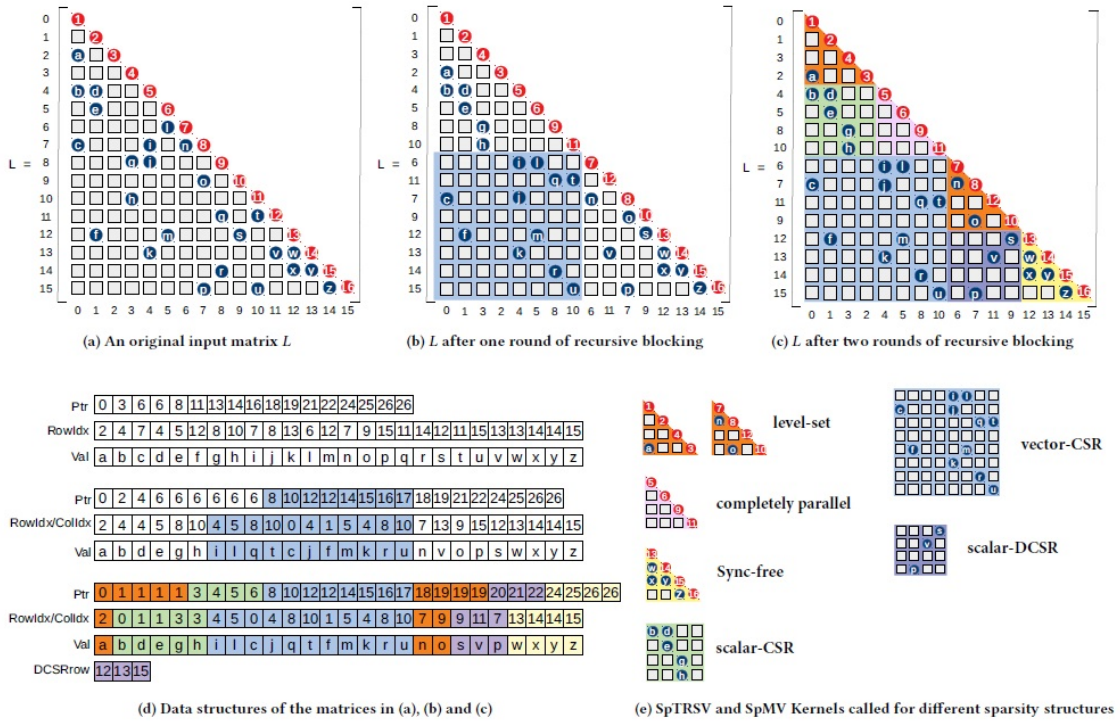


Figure 3: An example sparse matrix L is reordered, blocked, stored and executed in a fast recursive block algorithm.

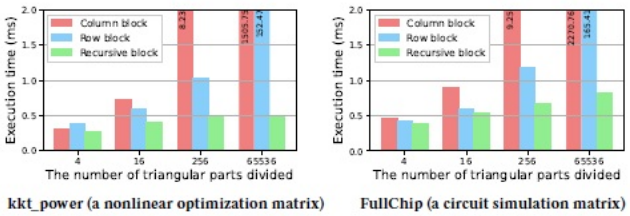


Figure 4: The execution time (in milliseconds) of the SpMV part of the three block algorithms on two sparse matrices.

3.3 Improved Recursive Block Data Structure

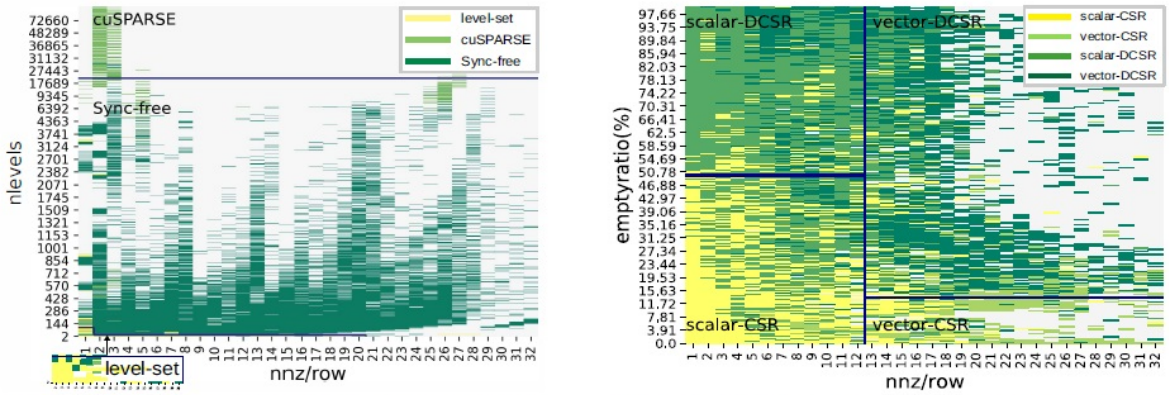
Although the standard recursive implementation listed in Algorithm 6 can work well, it often takes more memory than a loop implementation of the same function. Therefore we convert the input matrix into a new representation optimized for a loop version of the recursive block algorithm. The conversion is conducted in a preprocessing stage for SpTRSV. Its main function is to alternately store the triangular and square sub-matrices in the execution order shown in Figure 2(c).

Besides, we sort the components, i.e., both rows and columns, of any triangular matrix according to its level-set order. This means that the components in the same level-set are "physically" (if not consider virtual memory addressing) moved together. This will in general bring better cache utilization, and may bring more nonzeros into the square parts, since components with more nonzeros are more likely to move backwards but not to the very end.

Figure 3(a) plots an original input matrix L. It is reordered according to its level-set order and turned into Figure 3(b). Then its two triangular parts are further reordered according to their level-set orders, and the resulting matrix is the one shown in Figure 3(c). It can be seen that the number of nonzeros in the square part of Figure 3(b) is 11, which is higher than 8, i.e., the number of nonzeros in the same area of Figure 3(a). Obviously, through the reordering, more nonzeros are concentrated in square parts to achieve better parallel performance.

After reordering the sparse triangular matrix, the next step is to store it into a series of arrays. We assume the input matrix is in the CSC format including `col_ptr`, `row_idx` and `val` arrays as shown in the top three arrays in Figure 3(d). Once it is reordered and saved as three sub-matrices, the triangular parts can be still saved in CSC, but the square parts should be transposed into CSR for using faster SpMV kernel. Although now the CSC and CSR are mixed together, their arrays can be still stored continuously. As can be seen, the blue items in the three arrays in the middle of Figure 3(d) store the CSR data of the square part between the two CSC sub-matrices. When the matrix is further blocked into seven sub-matrices, the two CSC sub-matrices are further stored with the same layout. In the bottom of Figure 3(d), two new squares are in light green and purple, and the triangular parts are in orange and light yellow. Note that for brevity, we assume the diagonal is saved separately.

In addition, the square blocks may be very sparse, meaning that a large portion of rows are probably empty. In such case, we use a method similar to the DCSC format proposed by Buluç and Gilbert [15] and store the CSR data with a simplified row pointer



(a) Best SpTRSV kernels matching parameters `nnz/row` and `nlevels`

(b) Best SpMV kernels matching parameters `nnz/row` and `emptyratio`

Figure 5: Two heatmaps demonstrating the best matches of the areas and the fastest SpTRSV and SpMV kernels.

with an extra array saving the actual indices. We call this format DCSR in the paper. The last array in Figure 3(d) shows an example for saving the third square block of three nonzeros in DCSR.

3.4 Adaptive Kernel Selection for SpTRSV and SpMV in Recursive Block Algorithm

After the input matrix is divided and stored, the next step is to compute the small triangular and square blocks with SpTRSV and SpMV kernels, respectively. There have been a number of data structures and algorithms accelerating SpTRSV and SpMV on GPUs. Although most of them demonstrated good performance, they often need to convert the input matrix into a new and more complex format and add several auxiliary arrays. To avoid introducing extra complexity, we in this work use the most basic parallel implementations for SpTRSV and SpMV in our recursive block algorithm.

However, even selecting a basic SpTRSV or SpMV implementation working best for a specific sparsity structure is non-trivial. There are basically four kinds of sparsity structures in the smaller triangular matrices: (1) *diagonal structure* that contains only a diagonal and gives perfect parallelism for SpTRSV, (2) *highly parallel structure* that includes just a small number of level-sets and may be good for calling basic `level-set` algorithm using a thread or a 32-thread warp for solving one component, (3) *less highly parallel structure* of tens of or more than hundreds of levels, and the level-set methods normally will not give good performance because of inadequate parallelism. So the `Sync-free` algorithm using a warp for one component can be utilized, and (4) *near serial structure* that could not be well performed by `level-set` or `Sync-free`, but is found to be working well with cuSPARSE. In this case, we call `cuSPARSE` kernel for them.

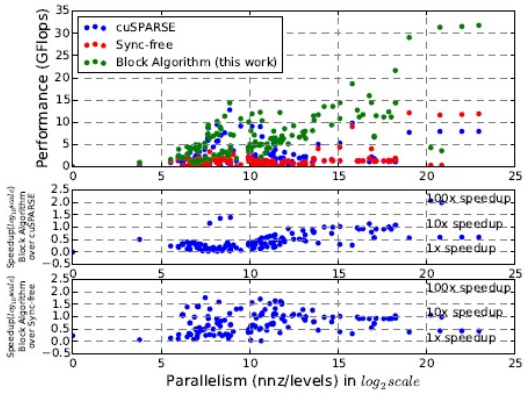
As for the square matrices, we categorize them into two kinds of sparsity structures: (1) *structure with short rows*, and (2) *structure with long rows*. They are computed by `scalar-CSR` and `vector-CSR` SpMV kernels, respectively. The difference is that the former uses one thread for computing one row, and the latter uses one warp for one row. Moreover, some very sparse square sub-matrices can be saved in the DCSR format mentioned in Section 3.3. So when the number of empty rows is high enough, two more SpMV kernels named `scalar-DCSR` and `vector-DCSR` can be called.

We now have in total four kernels for SpTRSV and four kernels for SpMV, and need to find critical parameters and thresholds to decide which kernel should be called for a given sub-matrix. For SpTRSV kernel selection, we introduce two critical parameters: (1) `nnz/row` indicating the average row length of a sub-matrix, and (2) `nlevels` giving the number of level-sets in the triangular part. For SpMV kernel selection, we also use two parameters: (1) the `nnz/row` mentioned above, and (2) `emptyratio` reflecting the ratio of the number of empty rows and the total number of rows.

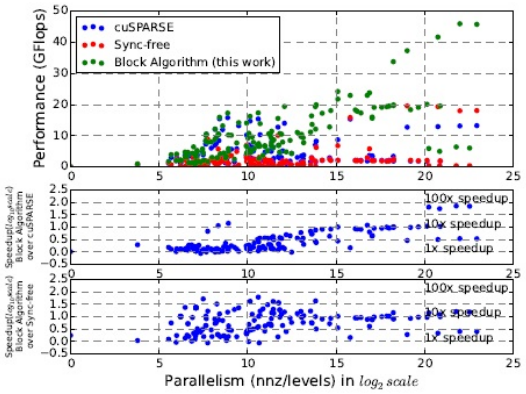
On the basis of the available kernels, we propose an adaptive approach to select the best thresholds for the parameters. Specifically, we divide the 159 sparse matrices (see Section 4.1 for details) into sub-matrices of various sizes, and run all kinds of kernels on the Titan RTX GPU (see Table 3) to collect a large amount of performance data. Specifically, 203,251 sets of SpTRSV performance data and 170,563 sets of SpMV data are collected. Each set of the data includes the two parameters (i.e., `nnz/row` and `nlevels` for SpTRSV, and `nnz/row` and `emptyratio` for SpMV) and the performance in GFlops of all kernels. Then it is easy to find the fastest kernel for the corresponding parameter pair. For different sub-matrices with the same parameter values, we select the kernel providing overall fastest performance as the best kernel for matching the feature. Figures 5(a) and (b) plot the best SpTRSV and SpMV kernels selected, respectively, according to the 373,814 sets of performance data.

Figure 5(a) shows that `level-set` works best in two scenarios: (1) `nnz/row` ≤ 15 and `nlevels` ≤ 20, or (2) `nnz/row` $= 1$ and `nlevels` ≤ 100. When `nlevels`> 20000, `cuSPARSE` is the fastest. In the other area, `Sync-free` is almost always the best choice. Note that the completely parallel case (i.e., only containing a diagonal) is easily selected and not plotted in the figure.

In Figure 5(b), it can be seen that when `nnz/row` ≤ 12, the kernels `scalar-CSR` and `scalar-DCSR` work best when `emptyratio` $\leq 50\%$ and `emptyratio` $> 50\%$, respectively. When `nnz/row` > 12, `vector-CSR` and `vector-DCSR` should be used for `emptyratio` $\leq 15\%$ and `emptyratio` $> 15\%$, respectively.



(a) Double precision SpTRSV performance and speedups on Titan X.



(b) Double precision SpTRSV performance and speedups on Titan RTX.

Figure 6: The two sub-figures on top show performance (in GFlops) of the three SpTRSV methods on the two GPUs. The four sub-figures on bottom show the speedups (in log_{10} scale) of our block algorirthm over the cuSPARSE v2 and Sync-free methods.

It is worth noting that the thresholds selected above are in general not the *optimal* choice (i.e., not all cells in the selected areas in Figures 5(a) and (b) have exactly the same color). But nevertheless we still use the adaptive selection depending on the large amount of test data for making our algorithm both simple and fast. According to the thresholds, Algorithm 7 gives a decision tree for selecting the best kernels in the improved recursive block algorithm.

Algorithm 7 An improved recursive block algorithm for SpTRSV.

```
1: function IMPROVED-RECURSIVE-BLOCK-ALGORITHM(nseg)
2:     for si = 0 → nseg − 1 do
3:         if info[si].shape == triangle then
4:             if nlevels[si]== 1 then
5:                 x_si ← SPTRSV-COMPLETELYPARALLEL(b_si,info[si])
6:             else if nlevels[si]> 20000 then
7:                 x_si ← SPTRSV-CUSPARSE(b_si,info[si])
8:             else if (nnz/row[si]== 1 and nlevels[si]≤ 100) or (nnz/row[si]≤ 15 and
   nlevels[si]≤ 20) then
9:                 x_si ← SPTRSV-LEVEL-SET(b_si,info[si])
10:            else
11:                x_si ← SPTRSV-SYNC-FREE(b_si,info[si])
12:            end if
13:        else if info[si].shape == square then
14:            if nnz/row[si]≤ 12 and emptyratio[si]≤ 50% then
15:                b_si ← SPMV-SCALAR-CSR(x_si,b_si,info[si])
16:            else if nnz/row[si]≤ 12 and emptyratio[si]> 50% then
17:                b_si ← SPMV-SCALAR-DCSR(x_si,b_si,info[si])
18:            else if nnz/row[si]> 12 and emptyratio[si]≤ 15% then
19:                b_si ← SPMV-VECTOR-CSR(x_si,b_si,info[si])
20:            else if nnz/row[si]> 12 and emptyratio[si]> 15% then
21:                b_si ← SPMV-VECTOR-DCSR(x_si,b_si,info[si])
22:            end if
23:        end if
24:    end for
25: end function
```

The last parameter should be decided is the depth of recursion. We select to constantly divide the matrix until the number of rows of the next smallest block is less than 20 times of the GPU core counts (e.g., on Titan RTX of 4608 CUDA cores, the block size should not be smaller than 92160), since we find that this is in general the smallest problem size can well saturate the high-end GPUs tested.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We on two NVIDIA GPUs, i.e., a Pascal architecture Titan X and a Turing architecture Titan RTX, evaluate three SpTRSV algorithms, i.e., a level-set method in cuSPARSE v2 of CUDA v10.2 [58], a Sync-free method [50] and the recursive block algorithm proposed in this paper. The platforms and the methods are listed in Table 3.

Table 3: The two GPUs and three algorithms evaluated.

Two NVIDIA GPUs	Three algorithms
(1) Titan X (Pascal), 3072 CUDA cores	(1) cuSPARSE v2 [58]
@ 1075 MHz, 12 GB, B/W 336.5 GB/s	(2) Sync-free [50]
(2) Titan RTX (Turing), 4608 CUDA cores	(3) Recursive block
@ 1770 MHz, 24 GB, B/W 672 GB/s	algorithm (this work)

From the SuiteSparse Matrix Collection [29] we select 159 sparse matrices. They are selected through three filter conditions: (1) square matrices, (2) the number of rows should be no less than 500,000, and (3) the number of nonezeros should be no less than 5,000,000 and no greater than 500,000,000. Their lower triangular parts (plus a diagonal to avoid singular) are tested in $Lx = b$. All tests run 200 times, and the average performance is reported.

We also selected six representative matrices of various sparsity structures for more precise comparison. Detailed features and performance data are listed in Table 4.

4.2 SpTRSV Performance Comparison

The performance data of the three methods on the two GPUs are shown in Figure 6. For brevity, only the results in double precision are plotted. As can be seen, our block algorithm performs much better than cuSPARSE and Sync-free, and is almost never slower than them. Specifically, on Titan X, compared to cuSPARSE, our method obtains an average of speedup of 5.00x, and the best speedup is 113.84x, occurs in matrix 'mawi_201512012345'. Compared to Sync-free, our method achieved an average speedup of 10.34x, and

Table 4: Six representative matrices (from electromagnetic problems, optimization problems and circuit simulation in the SuiteSparse Collection), the number of level-sets and parallelism (i.e., the number of components in level-sets), SpTRSV performance of the three algorithms, and speedups of our recursive block algorithm over cuSPARSE and Sync-free on Titan RTX.

Matrix	Plot	n	nnz	#level-sets	Parallelism			Performance (GFlops)			Speedups of blk alg.	
					min	ave.	max	cuSP.	Sync.	blk alg.	vs. cuSP.	vs. Sync.
nlpkkt200		16240000	232232816	2	8000000	8120000	8240000	13.26	18.09	**45.75**	**3.45x**	**2.53x**
mawi_201512020030		68863315	140570795	19	11	3624385	34544376	0.09	0.40	**6.41**	**72.03x**	**16.02x**
kkt_power		2063494	8545814	17	1090	121382	626597	3.67	5.81	**23.77**	**6.48x**	**4.09x**
FullChip		2987012	14804570	324	1	9219	468405	3.83	0.70	**7.78**	**2.03x**	**11.05x**
vas_stokes_4M		4382246	96836943	2815	1	1556	37519	15.39	0.28	**17.35**	**1.13x**	**61.08x**
tmt_sym		726713	2903837	726235	1	1	135	0.014	0.008	**0.015**	**1.03x**	**1.77x**

the best speedup is 57.97x, occurs in matrix 'af_shell10'. On Titan RTX, our method obtained the best speedups over cuSPARSE and Sync-free on matrices 'mawi_201512020030' and 'vas_stokes_4M', respectively. Overall, the best speedups over cuSPARSE and Sync-free are 72.03x and 61.08x, and the average speedups are 4.72x and 9.95x, respectively.

Moreover, we can see that the performance on Titan RTX of Turing architecture is in general around 40% faster than on Titan X of Pascal architecture. Considering the GPU architecture improvement and higher specifications of core counts and bandwidth (see Table 3), our proposed recursive block algorithm scales quite well.

According to the above results, we can say that the recursive block algorithm proposed in this paper makes SpTRSV significantly faster on GPUs. To conduct a more detailed analysis, we further list performance and speedups of six representative matrices in Table 4.

As can be seen, for matrices 'nlpkkt200' and 'mawi_201512020030' with very high parallelism, our method is 3.45x and 72.03x, and 2.53x and 16.02x faster than cuSPARSE and Sync-free, respectively. The first reason is that the blocks are now much smaller than the whole matrix, and only a small portion of vector x is accessed in each kernel call. Thus better cache utilization can be achieved. In addition, some small triangular sub-matrices now only have a diagonal, thus can be perfectly parallelized for much higher performance.

For matrix 'kkt_power' with good but not very high parallelism, our block algorithm gives 6.48x and 4.09x speedups over cuSPARSE and Sync-free, respectively. That is because now most nonzeros are moved to the square parts computed by the SpMV operations, so better parallelism could be naturally obtained.

As for matrices 'FullChip' and 'vas_stokes_4M' with limited parallelism, our method is not only faster than cuSPARSE, but also brings significant speedups (61.08x and 11.05x, respectively) over Sync-free. The main reason is that the two matrices both have a power-law distribution, and their long rows/columns are now cut into smaller segments, thus the load imbalance problem is relieved. Considering Sync-free uses atomic addition for accumulating intermediate products (but our method uses parallel sum in SpMV), the significant speedups are achieved as expected.

Even for matrix 'tmt_sym' with almost no parallelism (note that its average parallelism is 1), our method still gives comparable performance over cuSPARSE and is bit faster than Sync-free. This implies that our method in general would not degrade performance for such 'serial' problems.

Furthermore, the adaptive method for automatically selecting the best SpTRSV and SpMV kernels (recall Figure 5) also brings better overall performance for the above cases.

4.3 Performance of Different Precision

Both double and single precision computations are important for SpTRSV. For the sake of brevity, we only report performance of double precision in the other benchmarks (e.g., Figures 4 and 6, and Tables 4 and 5). To make the experiments more complete, we conduct a performance comparison of single and double precision of the three methods shown in Figure 7. As the major overhead of sparse matrix computations is often from the sparsity structure of the matrix, the performance ratio should in general not be similar to dense problems with a ratio of about 0.5. As can be seen, on the two GPUs, the ratio of double and single precision performance of the Sync-free is around 0.9, and the ratio of our block algorithm is between 0.8 and 0.9, while that of cuSPARSE is between 0.7 and 0.8. This indicates that compared to Sync-free, our algorithm is bit more sensitive to the costs for reading/writing floating point values. But compared to cuSPARSE, our approach is relatively more insensitive to the floating point precision used.

4.4 Preprocessing Cost Comparison

As SpTRSV is often used for solving triangular systems with multiple right-hand sides or preconditioning a system with ILU factoriation, it is important to consider the costs both for single SpTRSV and for preprocessing the input matrix. Besides, the number of iterations is crucial to evaluate overall improvement of a new SpTRSV method. Table 5 shows the preprocessing costs, a single SpTRSV cost and the overall costs for 100, 500 and 1000 iterations of a complete procedure by using cuSPARSE, Sync-free and our recursive block algorithm in double precision on the NVIDIA Titan RTX GPU.

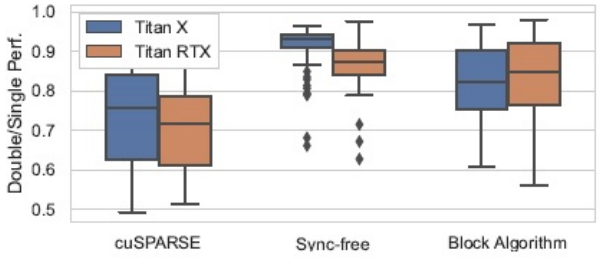


Figure 7: Performance ratio of double precision to single precision in box plots of running the three SpTRSV algorithms on the 159 matrices on the Titan X and Titan RTX GPUs.

As can be seen, the preprocessing costs of cuSPARSE and Sync-free are relatively low, since they only generate auxiliary arrays in the preprocessing stage. But although our method uses a bit longer preprocessing time (on average 9.16x over a single SpTRSV) for generating a new sparse matrix composed of successive sub-matrices (recall Figure 3), the overall cost for a complete computation with a number of iterations is still obviously lower than cuSPARSE and Sync-free, because of the much faster SpTRSV computation.

Table 5: The average time (in milliseconds) for preprocessing an input matrix, completing a single SpTRSV and solving problems of a preprocessing and 100, 500 and 1000 SpTRSV.

Methods	Preprocessing time	Single SpTRSV time	Overall time of 100 iters	Overall time of 500 iters	Overall time of 1000 iters
cuSPARSE v2	91.32	103.09	10400.71	51638.30	103185.29
Sync-free	2.34	94.79	9481.10	47396.15	94789.96
block algorithm	104.44	11.40	1244.05	5802.48	11500.52

5 RELATED WORK

To exploit parallelism from sparsity structures, the **level-set algorithms** have been proposed by Anderson and Saad [7] and Saltz [64]. The methods divide all components into a number of level-sets. The components within the same level-set can be solved in parallel, but the different level-sets have to run in serial. To use the highly parallel architectures of GPU, Maumov [58] implemented a level-set method and merged small level-sets into a single GPU kernel to save the cost for synchronizations between kernel calls. Park et al. [60] and Yımaz et al. [84] further analyzed the sparsity structure of the input matrix, and removed fine-grained dependencies and redundant waiting to achieve higher performance. Li and Saad [45] demonstrated that multiple minimum degree (MMD) reordering can reduce the number of level-sets and improve parallelism. Liu et al. [49, 50] proposed a CSC format synchronization-free SpTRSV algorithm that fully removes the barrier synchronizations between level-sets through utilizing atomic operations on GPUs. Dufrechou and Ezzatti [35, 36] designed a CSR version of the synchronization-free algorithm, and simplified the procedure finding level-sets. Su et al. [70] recently further exploit large-scale thread-level parallelism for faster synchronization-free algorithm on modern GPUs. In this work, the level-set information is used for reordering the triangular parts, and the level-set method and its Sync-free variant are called to solve the triangular sub-systems.

Another group of algorithms relies on graph colouring or partitioning. Schreiber and Tang [67] for the first time designed **colour-set algorithms** using graph colouring for accelerating SpTRSV. Each colour-set shows up as a diagonal block, which can be independent to the others thus can be processed in parallel. Naumov et al. [59] and Suchoski et al. [71] demonstrated the effectiveness of the colour-set methods for parallel SpTRSV on GPUs. Kabir et al. [43] further improved the performance of SpTRSV on NUMA architectures through graph colouring. Picciau et al. [61] developed a graph partitioning method for achieving better data locality and higher concurrency. However, in spite of the performance gain, it is well known that graph colouring, reordering or partitioning is NP-complete and in general very costly. In contrast, our method proposed does not colour or partition the graph form of the input matrix, thus could keep the preprocessing cost relatively low.

Tile/Block algorithms for dense matrix problems have been particularly successful in the recent years. The tile algorithms designed by Buttari et al. [18, 19] have been widely used in factorization routines. Haidar et al. [39] studied scheduling problems in tile algorithms of PLASMA [2]. Amestoy et al. [6] and Akbudak et al. [3] developed block low-rank methods for matrix factorization. Haidar et al. [39] developed tile algorithms for eigensolver problems. Dongarra et al. [31, 32] proposed recursive tile methods for LU factorization. Charara et al. [20, 21] exploited recursive blocking for accelerating the dense triangular solve on multi-GPUs.

Tile/Block algorithms for sparse matrix problems also received attention but have not shown wide effectiveness. Mayer [56] pointed out that 2D blocking should be able to accelerate SpTRSV but did not provide positive experimental results. Wang et al. [78, 79] developed a new format called Sparse Level Tile and a new method for structured problem on Sunway processors. But it may not be equally efficient for other architectures such as x86 and GPU. Duff and Uçar [34] studied block triangular form of symmetric sparse matrices. Vuduc et al. [76] and Bradley [14] developed blocking schemes for SpTRSV, but they mainly work well for relatively dense triangular matrices from sparse LU decomposition.

It is also possible to solve triangular systems through **iterative methods** that expose higher parallelism by calling a number of SpMV operations. Anzt et al. [8, 11] and Chow et al. [25] proposed several blocking and Jacobi iterations methods for SpTRSV in incomplete LU decomposition [12, 26]. Anzt et al. [10, 13] and Uçar and Aykanat [73] also proposed fast methods for sparse approximate inverse for solving triangular systems.

There has been much research about **SpMV optimization** on modern architectures heavily used in this work. Liu and Vinter proposed the CSR5 format [52] for better SIMD utilization, load balancing and cross-platform execution on various multi-core and many-core processors. Saule et al. [66] evaluated several sparse kernels on Xeon Phi. Matam and Kothapalli [55] developed new methods for selecting formats for SpMV computation. Sadi et al. [63] accelerated SpMV by using high bandwidth memory. Buono et al. [16] proposed new partitioning methods for SpMV on POWER processors. Buttari et al. [17] designed aligned block sparse matrix format. Vooturi et al. [72, 75] recently exploited block sparsity

patterns in weight matrices for accelerating sparse neural network computations.

Most of the aforementioned work can be seen as an analysis phase (or inspector) used before repeatedly calling a sparse kernel for many times. The stage can be generalized to be part of a **compiler**. The Sparse Polyhedral Framework proposed by Strout et al. [68, 69], the Sympiler code generator and the ParSy framework developed by Cheshmi et al. [23, 24] are all representative work in this area. Besides, Mohammadi et al. [57] and Venkat et al. [74] proposed effective compilation strategy for parallel SpTRSV.

Distributed algorithms for solving both dense and sparse triangular systems also received much attention. Irony and Toledo [42] designed 3D algorithms for distributed dense triangular solve, Wicky et al. [80] developed a new communication-avoid implementation for the algorithm, and González-Domínguez et al. [37] implemented the algorithm by using the UPC language. Recently the SuperLU_DIST package updated its solve phase with new distributed SpTRSV methods. Liu et al. [54] used asynchronous communication in the 2D block cyclic layout of SuperLU_DIST, Sao et al. [65] designed a well-scaled communication-avoid SpTRSV, and Ding et al. [30] further improved distributed SpTRSV through one-sided communication. We believe our block algorithms proposed here can help distributed SpTRSV to be more efficient.

6 CONCLUSIONS

In this paper we have implemented three block algorithms for parallel SpTRSV on modern GPUs, and proposed an adaptive approach that automatically selects the best kernels and parameters for computing sub-matrices divided. Our experiments conducted on 159 matrices and on two high-end NVIDIA GPUs have shown on average 4.72x (up to 72.03x) and 9.95x (up to 61.08x) speedups over cuSPARSE and Sync-free methods, respectively. Also, the preprocessing cost of our method is moderate.

ACKNOWLEDGMENTS

We deeply appreciate the invaluable comments from all the reviewers. Weifeng Liu is the corresponding author of this paper. This research was supported by the Science Challenge Project under Grant No. TZZT2016002, the National Natural Science Foundation of China under Grant No. 61972415, and the Science Foundation of China University of Petroleum, Beijing under Grant No. 2462019YJRC004, 2462020XKJS03.

REFERENCES

[1] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Implementing Multifrontal Sparse Solvers for Multicore Architectures with Sequential Task Flow Runtime Systems. *ACM Trans. Math. Softw.*, 43(2), 2016.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. *Journal of Physics: Conference Series*, 180:012037, 2009.

[3] K. Akbudak, H. Ltaief, A. Mikhalev, A. Charara, A. Esposito, and D. Keyes. Exploiting Data Sparsity for Large-Scale Matrix Computations. In *Euro-Par '18*, pages 721–734, 2018.

[4] P. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. On the Complexity of the Block Low-Rank Multifrontal Factorization. *SIAM Journal on Scientific Computing*, 39(4):A1710–A1740, 2017.

[5] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures. *ACM Trans. Math. Softw.*, 45(1), 2019.

[6] P. R. Amestoy, A. Buttari, J.-Y. L'Excellent, and T. Mary. Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures. *ACM Trans. Math. Softw.*, 45(1), 2019.

[7] E. Anderson and Y. Saad. Solving Sparse Triangular Linear Systems on Parallel Computers. *International Journal of High Speed Computing*, 1(1):73–95, 1989.

[8] H. Anzt, E. Chow, and J. Dongarra. Iterative Sparse Triangular Solves for Preconditioning. In *Euro-Par '15*, pages 650–661. 2015.

[9] H. Anzt, E. Chow, and J. Dongarra. ParILUT–A New Parallel Threshold ILU Factorization. *SIAM Journal on Scientific Computing*, 40(4):C503–C519, 2018.

[10] H. Anzt, E. Chow, T. Huckle, and J. Dongarra. Batched Generation of Incomplete Sparse Approximate Inverses on GPUs. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 49–56, 2016.

[11] H. Anzt, E. Chow, D. B. Szyld, and J. Dongarra. Domain Overlap for Iterative Sparse Triangular Solves on GPUs. In *Software for Exascale Computing - SPPEXA 2013-2015*, pages 527–545, 2016.

[12] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, and M. Köhler. Preconditioned Krylov solvers on GPUs. *Parallel Computing*, 68:32 – 44, 2017.

[13] H. Anzt, T. Huckle, J. Brackle, and J. Dongarra. Incomplete Sparse Approximate Inverses for Parallel Preconditioning. *Parallel Computing*, 71:1–22, 2018.

[14] A. M. Bradley. A Hybrid Multithreaded Direct Sparse Triangular Solver. In *SIAM CSC workshop '16*, pages 13–22, 2016.

[15] A. Buluç and J. R. Gilbert. On the Representation and Multiplication of Hypersparse Matrices. In *IPDPS '08*, pages 1–11, 2008.

[16] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan. Optimizing Sparse Matrix-Vector Multiplication for Large-Scale Data Analytics. In *ICS '16*, 2016.

[17] A. Buttari, V. Eijkhout, J. Langou, and S. Filippone. Performance Optimization and Modeling of Blocked Sparse Kernels. *The International Journal of High Performance Computing Applications*, 21(4):467–484, 2007.

[18] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel Tiled QR Factorization for Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[19] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing*, 35(1):38 – 53, 2009.

[20] A. Charara, D. Keyes, and H. Ltaief. A Framework for Dense Triangular Matrix Kernels on Various Manycore Architectures. *Concurrency and Computation: Practice and Experience*, 29(15):e4187, 2017.

[21] A. Charara, H. Ltaief, and D. Keyes. Redesigning Triangular Dense Matrix Computations on GPUs. In *Euro-Par '16*, pages 477–489, 2016.

[22] J. Chen, J. Fang, W. Liu, T. Tang, and C. Yang. clMF: A Fine-Grained and Portable Alternating Least Squares Algorithm for Parallel Matrix Factorization. *Future Generation Computer Systems*, 108:1192–1205, 2020.

[23] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi. Sympiler: Transforming Sparse Matrix Codes by Decoupling Symbolic Analysis. In *SC '17*, page 1–13, 2017.

[24] K. Cheshmi, S. Kamil, M. M. Strout, and M. M. Dehnavi. ParSy: Inspection and Transformation of Sparse Matrix Computations for Parallelism. In *SC '18*, pages 779–793, 2018.

[25] E. Chow, H. Anzt, J. Scott, and J. Dongarra. Using Jacobi Iterations and Blocking for Solving Sparse Triangular Systems in Incomplete Factorization Preconditioning. *Journal of Parallel and Distributed Computing*, 119:219 – 230, 2018.

[26] E. Chow and A. Patel. Fine-Grained Parallel Incomplete LU Factorization. *SIAM Journal on Scientific Computing*, 37(2):C169–C193, 2015.

[27] T. Cojean, A. Guermouche, A. Hugo, R. Namyst, and P. Wacrenier. Resource Aggregation for Task-Based Cholesky Factorization on Top of Modern Architectures. *Parallel Computing*, 83:73 – 92, 2019.

[28] T. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006.

[29] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011.

[30] N. Ding, S. Williams, Y. Liu, and X. S. Li. Leveraging One-Sided Communication for Sparse Triangular Solvers. In *SIAM PP '20*, pages 93–105, 2020.

[31] J. Dongarra, V. Eijkhout, and P. Łuszczek. Recursive Approach in Sparse Matrix LU Factorization. *Scientific Programming*, 9(1):51–60, 2001.

[32] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Achieving Numerical Accuracy and High Performance Using Recursive Tile LU Factorization with Partial Pivoting. *Concurrency and Computation: Practice and Experience*, 26(7):1408–1431, 2014.

[33] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., 2nd edition, 2017.

[34] I. S. Duff and B. Uçar. On the Block Triangular Form of Symmetric Matrices. *SIAM Review*, 52(3):455–470, 2010.

[35] E. Dufrechou and P. Ezzatti. A New GPU Algorithm to Compute a Level Set-Based Analysis for the Parallel Solution of Sparse Triangular Systems. In *IPDPS '18*, pages 920–929, 2018.

[36] E. Dufrechou and P. Ezzatti. Solving Sparse Triangular Linear Systems in Modern GPUs: A Synchronization-Free Algorithm. In *PDP '18*, pages 196–203, 2018.

[37] J. González-Domínguez, M. J. Martín, G. L. Taboada, and J. Touriño. Dense Triangular Solvers on Multicore Clusters using UPC. *Procedia Computer Science*, 4:231 – 240, 2011.

[38] L. Grigori, J. W. Demmel, and X. S. Li. Parallel Symbolic Factorization for Sparse LU with Static Pivoting. *SIAM Journal on Scientific Computing*, 29(3):1289–1314, 2007.

[39] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. *Concurrency and Computation: Practice and Experience*, 24(3):305–321, 2012.

[40] J. D. Hogg. A Fast Dense Triangular Solve in CUDA. *SIAM Journal on Scientific Computing*, 35(3):C303–C322, 2013.

[41] K. Hou, W. Liu, H. Wang, and W.-c. Feng. Fast Segmented Sort on GPUs. In *ICS '17*, pages 12:1–12:10, 2017.

[42] D. Irony and S. Toledo. Trading Replication for Communication in Parallel Distributed-Memory Dense Solvers. *Parallel Processing Letters*, 12(01):79–94, 2002.

[43] H. Kabir, J. D. Booth, G. Aupy, A. Benoit, Y. Robert, and P. Raghavan. STS-k: A Multilevel Sparse Triangular Solution Scheme for NUMA Multicores. In *SC '15*, pages 55:1–55:11, 2015.

[44] A. Li, W. Liu, M. R. B. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez, and S. L. Song. Exploring and Analyzing the Real Impact of Modern On-package Memory on HPC Scientific Kernels. In *SC '17*, pages 26:1–26:14, 2017.

[45] R. Li and Y. Saad. GPU-Accelerated Preconditioned Iterative Linear Solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[46] X. S. Li. An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Trans. Math. Softw.*, 31(3):302–325, 2005.

[47] J. Liu, X. He, W. Liu, and G. Tan. Register-Aware Optimizations for Parallel Sparse Matrix-Matrix Multiplication. *International Journal of Parallel Programming*, page 403–417, 2019.

[48] W. Liu. *Parallel and Scalable Sparse Basic Linear Algebra Subprograms.* PhD thesis, University of Copenhagen, 2015.

[49] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter. A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves. In *Euro-Par '16*, pages 617–630, 2016.

[50] W. Liu, A. Li, J. D. Hogg, I. S. Duff, and B. Vinter. Fast Synchronization-Free Algorithms for Parallel Sparse Triangular Solves with Multiple Right-Hand Sides. *Concurrency and Computation: Practice and Experience*, 29(21):e4244–n/a, 2017.

[51] W. Liu and B. Vinter. A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. *Journal of Parallel and Distributed Computing*, 85(C):47–61, 2015.

[52] W. Liu and B. Vinter. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *ICS '15*, pages 339–350, 2015.

[53] W. Liu and B. Vinter. Speculative Segmented Sum for Sparse Matrix-vector Multiplication on Heterogeneous Processors. *Parallel Computing*, 49(C):179–193, 2015.

[54] Y. Liu, M. Jacquelin, P. Ghysels, and X. S. Li. Highly Scalable Distributed-Memory Sparse Triangular Solution Algorithms. In *SIAM CSC workshop '18*, pages 87–96.

[55] K. K. Matam and K. Kothapalli. Accelerating Sparse Matrix Vector Multiplication in Iterative Methods Using GPU. In *ICPP '11*, pages 612–621, 2011.

[56] J. Mayer. Parallel Algorithms for Solving Linear Systems with Sparse Triangular Matrices. *Computing*, 86(4):291–312, 2009.

[57] M. S. Mohammadi, T. Yuki, K. Cheshmi, E. C. Davis, M. Hall, M. M. Dehnavi, P. Nandy, C. Olschanowsky, A. Venkat, and M. M. Strout. Sparse Computation Data Dependence Simplification for Efficient Compiler-Generated Inspectors. In *PLDI '19*, page 594–609, 2019.

[58] M. Naumov. Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU. Technical report, NVIDIA, 2011.

[59] M. Naumov, P. Castonguay, and J. Cohen. Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU. *Nvidia White Paper*, 2015.

[60] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey. Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver. In *ISC '14*, pages 124–140, 2014.

[61] A. Picciau, G. E. Inggs, J. Wickerson, E. C. Kerrigan, and G. A. Constantinides. Balancing Locality and Concurrency: Solving Sparse Triangular Systems on GPUs. In *HiPC '16*, 2016.

[62] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[63] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti. Efficient SpMV Operation for Large and Highly Sparse Matrices Using Scalable Multi-Way Merge Parallelization. In *MICRO '19*, page 347–358, 2019.

[64] J. H. Saltz. Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors. *SIAM Journal on Scientific and Statistical Computing*, 11(1):123–144, 1990.

[65] P. Sao, R. Kannan, X. S. Li, and R. Vuduc. A Communication-Avoiding 3D Sparse Triangular Solver. In *ICS '19*, page 127–137, 2019.

[66] E. Saule, K. Kaya, and Ü. V. Çatalyürek. Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi. In *PPAM '14*, pages 559–570, 2014.

[67] R. Schreiber and W.-P. Tang. Vectorizing the Conjugate Gradient Method. In *Proceedings of the Symposium on CYBER 205 Applications*, 1982.

[68] M. M. Strout, M. Hall, and C. Olschanowsky. The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018.

[69] M. M. Strout, A. LaMielle, L. Carter, J. Ferrante, B. Kreaseck, and C. Olschanowsky. An Approach for Code Generation in the Sparse Polyhedral Framework. *Parallel Computing*, 53:32 – 57, 2016.

[70] 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 *ICPP '20*, 2020.

[71] B. Suchoski, C. Severn, M. Shantharam, and P. Raghavan. Adapting Sparse Triangular Solution to GPUs. In *ICPPW '12*, pages 140–148, 2012.

[72] D. T. Vooturi, G. Varma, and K. Kothapalli. Dynamic Block Sparse Reparameterization of Convolutional Neural Networks. In *ICCV '19 Workshops*, Oct 2019.

[73] B. Uçar and C. Aykanat. Partitioning Sparse Matrices for Parallel Preconditioned Iterative Methods. *SIAM Journal on Scientific Computing*, 29(4):1683–1709, 2007.

[74] A. Venkat, M. S. Mohammadi, J. Park, H. Rong, R. Barik, M. M. Strout, and M. Hall. Automating Wavefront Parallelization for Sparse Matrix Computations. In *SC '16*, pages 480–491, 2016.

[75] D. T. Vooturi and K. Kothapalli. Efficient Sparse Neural Networks Using Regularized Multi Block Sparsity Pattern on a GPU. In *HiPC '19*, pages 215–224, 2019.

[76] R. Vuduc, S. Kamil, J. Hsu, R. Nishtala, J. W. Demmel, and K. A. Yelick. Automatic Performance Tuning and Analysis of Sparse Triangular Solve. In *ICS '02 Workshop*, 2002.

[77] H. Wang, W. Liu, K. Hou, and W.-c. Feng. Parallel Transposition of Sparse Data Structures. In *ICS '16*, pages 33:1–33:13, 2016.

[78] X. Wang, W. Liu, W. Xue, and L. Wu. SwSpTRSV: A Fast Sparse Triangular Solve with Sparse Level Tile Layout on Sunway Architectures. In *PPoPP '18*, page 338–353, 2018.

[79] X. Wang, P. Xu, W. Xue, Y. Ao, C. Yang, H. Fu, L. Gan, G. Yang, and W. Zheng. A Fast Sparse Triangular Solver for Structured-Grid Problems on Sunway Many-Core Processor SW26010. In *ICPP '18*, 2018.

[80] T. Wicky, E. Solomonik, and T. Hoefler. Communication-Avoiding Parallel Algorithms for Solving Triangular Systems of Linear Equations. In *IPDPS '17*, pages 678–687, 2017.

[81] M. Wittmann, G. Hager, R. Janalik, M. Lanser, A. Klawonn, O. Rheinbach, O. Schenk, and G. Wellein. Multicore Performance Engineering of Sparse Triangular Solves Using a Modified Roofline Model. In *SBAC-PAD '18*, pages 233–241, 2018.

[82] M. M. Wolf, M. A. Heroux, and E. G. Boman. Factors Impacting Performance of Multithreaded Sparse Triangular Solve. In *VECPAR '10*, pages 32–44. 2011.

[83] Z. Xie, G. Tan, W. Liu, and N. Sun. IA-SpGEMM: An Input-Aware Auto-Tuning Framework for Parallel Sparse Matrix-Matrix Multiplication. In *ICS '19*, pages 94–105, 2019.

[84] B. Yılmaz, B. Sipahioğrlu, N. Ahmad, and D. Unat. Adaptive Level Binning: A New Algorithm for Solving Sparse Triangular Systems. In *HPC Asia '20*, page 188–198, 2020.

[85] F. Zhang, W. Liu, N. Feng, J. Zhai, and X. Du. Performance Evaluation and Analysis of Sparse Matrix and Graph Kernels on Heterogeneous Processors. *CCF Transactions on High Performance Computing*, pages 131–143, 2019.

[86] F. Zhang, J. Zhai, B. Wu, B. He, W. Chen, and X. Du. Automatic Irregularity-Aware Fine-Grained Workload Partitioning on Integrated Architectures. *IEEE Transactions on Knowledge and Data Engineering*, 2019.