YuenyeungSpTRSV: A Thread-Level and Warp-Level Fusion Synchronization-Free Sparse Triangular Solve

Feng Zhang, Jiya Su, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang, *Member, IEEE*

Abstract—Sparse triangular solves (SpTRSVs) are widely used in linear algebra domains, and several GPU-based SpTRSV algorithms have been developed. Synchronization-free SpTRSVs, due to their short preprocessing time and high performance, are currently the most popular SpTRSV algorithms. However, we observe that the performance of those SpTRSV algorithms on different matrices can vary greatly by 845 times. Our further studies show that when the average number of components per level is high and the average number of nonzero elements per row is low, those SpTRSVs exhibit extremely low performance. The reason is that, they use a warp on the GPU to process a row in sparse matrices, and such warp-level designs have severe underutilization of the GPU. To solve this problem, we propose YuenyeungSpTRSV, a thread-level and wrap-level fusion synchronization-free SpTRSV algorithm, which handles the rows with a large number of nonzero elements at warp-level while the rows with a low number of nonzero elements at thread-level. Particularly, YuenyeungSpTRSV has three novel features. First, unlike the previous studies, YuenyeungSpTRSV does not need long preprocessing time to calculate levels. Second, YuenyeungSpTRSV exhibits high performance on matrices that previous SpTRSVs cannot handle efficiently. Third, YuenyeungSpTRSV's optimization does not rely on the specific sparse matrix storage format. Instead, it can achieve very good performance on the most popular sparse matrix storage, compressed sparse row (CSR) format, and thus users do not need to conduct format conversion. We evaluate YuenyeungSpTRSV with 245 matrices from the Florida Sparse Matrix Collection on four GPU platforms, and experiments show that our YuenyeungSpTRSV exhibits 7.14 GFLOPS/s, which is 5.98x speedup over the state-of-the-art synchronization-free SpTRSV algorithm, and 4.83x speedup over the SpTRSV in cuSPARSE.

Index Terms—Thread-level, warp-level, synchronization-free, SpTRSV, GPU

1 Introduction

Sparse triangular solves (SpTRSVs) have been extensively used in linear algebra fields, and have been indispensable building blocks in many numerical linear algebra routines, such as least-squares problems [1], direct methods [2], and preconditioners of sparse iterative solvers [3]. For an equation set, $Lx = b$, where L is a lower triangular sparse matrix, x is the target solution vector, and b is a dense vector, SpTRSV computes the target solution vector x based on L and b. Because GPUs demonstrate powerful computing capabilities in the field of linear algebra, researchers have been exploring using GPUs to parallelize the SpTRSV algorithms. However, compared with other linear algebra algorithms for sparse matrices [4], such as sparse matrix-matrix multiplication [5], [6], sparse matrix-vector multiplication [7], [8], [9], [10], [11], [12], and sparse transposition [13], SpTRSV is challenging to be efficiently parallelized because there are more internal dependencies in the solution process.

To parallel the SpTRSV algorithm, we need to understand more details about SpTRSV: the solution in SpTRSV can be divided into subsolutions for each component x_i, which can be parallelized. There exist dependencies in the solutions for each x_i: solving a component x_i may depend on the other components x_j $(j < i)$. Furthermore, the dependency relationships in the component solutions can be described in a directed acyclic graph (DAG), and the components in the dependency DAG can be divided into different levels. Only the components at the same level can be solved in parallel. In the worst case, only one component exists in one level, so there is no parallelism in this case.

To address the dependency problem, a level-set SpTRSV algorithm has been proposed [14], [15], which involves a preprocessing step to group the components in the same level into a set, and the components in the same set can be solved in parallel. However, such a level-set preprocessing often takes too much time [16]; in our experiment, the preprocessing time could be dozens of times to the execution time of solving SpTRSV itself. Moreover, Li *et al.* [17] pointed out that the

- *Feng Zhang, Ruofan Wu, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), School of Information, Renmin University of China, Beijing 100872, China. E-mail: {fengzhang, 2017202106, duyong}@ruc.edu.cn.*
- *Weifeng Liu is with the Department of Computer Science and Technology, China University of Petroleum, Beijing 100088, China. E-mail: weifeng.liu@cup.edu.cn.*
- *Bingsheng He is with the School of Computing, National University of Singapore, Singapore 119077, Singapore. E-mail: hebs@comp.nus.edu.sg.*
- *Jiya Su and Rujia Wang are with Computer Science Department, Illinois Institute of Technology, Chicago, IL 60616 USA. E-mail: Jiya_Su@ruc.edu.cn, rwang67@iit.edu.*

Manuscript received 2 Sept. 2020; revised 2 Feb. 2021; accepted 6 Mar. 2021. Date of publication 17 Mar. 2021; date of current version 30 Mar. 2021. (Corresponding author: Feng Zhang.)
Recommended for acceptance by P. Bangalore.
Digital Object Identifier no. 10.1109/TPDS.2021.3066635

inter-level synchronization incurs large performance overhead in the level-set SpTRSV. Although recent level-set SpTRSV optimizations, such as simplifying synchronization by pruning [18] and replacing synchronization by atomic operations [16], reduce the number of synchronizations, the synchronization overhead is still prohibitively high. Later, Liu and others [16] proposed a synchronization-free SpTRSV algorithm, which solves the synchronization problem and greatly reduces the preprocessing time. Currently, this algorithm is the state-of-the-art SpTRSV algorithm, which outperforms other algorithms on a wide range of workloads. However, this algorithm only considers GPU warp-level parallelism, and we find that such a warp-level synchronization-free SpTRSV algorithm exhibits significant performance degradation when 1) the average number of components per level is large, and 2) the number of related nonzero elements for each row is small.

Solving such synchronization-free SpTRSV performance degradation problems requires handling the following three challenges. First, new SpTRSV algorithms need to be designed to avoid thread idle within warps on GPU. Second, novel intra-warp communication mechanisms need to be carefully designed to avoid deadlocks, since threads within a warp in GPU execute in a lock-step manner. Third, preprocessing time should be as short as possible for the usability and applicability of SpTRSV.

To solve the challenges above, we propose Yuenyeung SpTRSV, a thread-level and warp-level fusion synchronization-free SpTRSV algorithm, which addresses the sparse situations that current synchronization-free SpTRSV algorithm cannot handle efficiently. Those matrices that have *a large number of components per level* and *a small number of nonzero elements per row* are commonly seen in graph applications. Thus, we develop an indicator, *parallel granularity*, detailed in Section 3.2, to comprehensively describe these two characteristics of sparse matrices. A high parallel granularity means that the warp-level synchronization-free SpTRSV algorithms may not be able to fully utilize GPU resources.

The high-level idea of YuenyeungSpTRSV is that, for the rows with a low number of nonzero elements, we use one thread to solve one component, which avoids the resource waste caused by idle threads. At the same time, for the other rows with a large number of nonzero elements, we process these rows at warp level to maintain load balance in GPU warps. Moreover, in order to improve SpTRSV performance in a holistic manner, YuenyeungSpTRSV has three novel features. First, unlike the previous studies [14], [15], YuenyeungSpTRSV avoids the lengthy preprocessing for calculating the levels. Second, YuenyeungSpTRSV exhibits high performance on matrices that have high parallel granularities, which is complementary to current warp-level synchronization-free SpTRSVs. Third, YuenyeungSpTRSV's optimization does not rely on the specific sparse matrix storage format. Instead, it can achieve very good performance on the most popular sparse matrix storage, compressed sparse row (CSR) format, and thus users do not need to conduct format conversion in advance. Our preliminary work, CapelliniSpTRSV [19], provides thread-level optimization and design. In contrast, this work provides both thread-level and warp-level fusion design of synchronization-free SpTRSV, including 1) integration of warp-level and thread-level SpTRSV algorithms, 2) threshold detection to distinguish thread-level and warp-level designs, and 3) computation and segmentation algorithms in YuenyeungSpTRSV. Additionally, we provide both CUDA and OpenCL versions of our implementation, so that our method can run on various platforms, which can help existing applications directly.

Note that YuenyeungSpTRSV involves novel cross-GPU optimizations, including data structures to represent different processing levels, a lightweight model to predict the configuration, and adaptation to GPU architectures. Moreover, we provide cross-platform YuenyeungSpTRSV implementation. We provide not only CUDA but also OpenCL implementations, so that our method can run on various platforms, which can help existing applications directly.

We evaluate YuenyeungSpTRSV with 245 matrices from the University of Florida Sparse Matrix Collection [20] on four GPU platforms, and compare our method with the state-of-the-art SpTRSV algorithm [16], and the SpTRSV in cuSPARSE [21]. The experimental results show that YuenyeungSpTRSV exhibits high efficiency for the matrices that have high parallel granularity. YuenyeungSpTRSV achieves on average 5.98x performance speedup over the state-of-the-art SpTRSV algorithm [16], and 4.83x speedup over the SpTRSV in cuSPARSE.

To summarize our contributions in this paper:

- We show our insights in current SpTRSV algorithms and propose parallel granularity to describe sparse matrices.
- We develop YuenyeungSpTRSV, a thread-level and warp-level fusion synchronization-free SpTRSV, to process sparse matrices that previous SpTRSV algorithms cannot handle efficiently.
- We evaluate YuenyeungSpTRSV with 245 matrices, and demonstrate its benefits over the state-of-the-art SpTRSV.

2 Preliminaries

In this section, we first discuss the background and preliminaries about SpTRSV, including the basic SpTRSV, level-set SpTRSV, and synchronization-free SpTRSV. Then, we summarize and compare current SpTRSV algorithms, and identify their limitations.

2.1 Concepts and Basic SpTRSV

We first introduce the basic concepts that are essential for understanding SpTRSV. For the equation set, $Lx = b$, we provide the following concepts.

- *Component*: An element in solution vector x.
- *Element*: A nonzero element in matrix L, such as $L_{0,0}$.
- *Dependency*: If the solution of component x_i needs the value of component x_j, x_i has a dependency on x_j.
- *Level*: A solution order according to the dependencies among components. The components at the same level form a level-set.

Sparse Matrix in CSR Format. The compressed sparse row (CSR) format is the most popular sparse matrix compression format, storing a matrix in three arrays without zero values. Fig. 1 illustrates a sparse triangular matrix L in SpTRSV. Fig. 1a shows an 8-by-8 sparse triangular matrix,

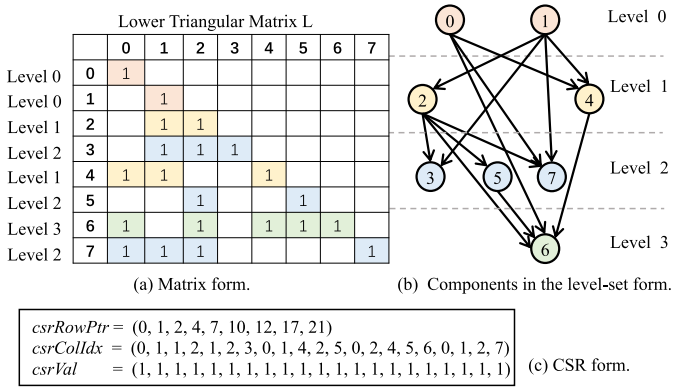


Fig. 1. Lower triangular matrix L in CSR format: (a) the color shows the level of the row; (b) dependency of the components x. Each component relates to one row, and there are four level-sets in L; (c) the CSR format.

which can be divided into four level sets, as shown in Fig. 1b. The matrix in Fig. 1a can be further stored in Fig. 1c. The array *csrRowPtr* stores the beginning position of each row, the array *csrColIdx* stores the column numbers of each element, and the array *csrVal* stores the values.

Basic SpTRSV Algorithm. We show the basic SpTRSV in Algorithm 1. The algorithm traverses all rows (Line 3). In each row, it calculates all elements in the row except the last one, and stores the value in intermediate variable *left_sum* (Lines 5-6). At last, the component of the solution vector x in the same row is solved (Line 7).

Algorithm 1. Basic SpTRSV Algorithm for $Lx = b$

```
1: Input: InputMatrix L, array b
2: Output: array x
3: for i = 0 to L.rows-1 do
4:    left_sum ← 0
5:    for j = L.RowPtr[i] to L.RowPtr[i+1]-2 do
6:       left_sum ← left_sum + L.Val[j] × x[L.ColIdx[j]]
7:    x[i] ← (b[i]- left_sum) / L.Val[L.RowPtr[i+1]-1]
```

2.2 Level-Set SpTRSV

As discussed in Section 2.1, the components x_i at the same level can be solved independently and simultaneously. Therefore, the components can be partitioned into different level-sets, so that the components in the same set can be solved in parallel, while the sets are processed sequentially. Each set relates to one level. However, a preprocessing is required for generating level-sets. In the preprocessing stage of the previous studies [14], [15], the algorithm stores the level-set number in variable *layer*, records the row number in each level in the array *layer_num*, and rearranges the order of rows according to their levels in the array *order*.

Level-Set SpTRSV Algorithm. We show the Level-Set SpTRSV algorithm in Algorithm 2. The algorithm partitions the components into level-sets, and the components in the same level-set can be solved in parallel (Line 4), where *id* is the row number to solve (Line 5). After calculating the whole nonzero elements in the row (Lines 6-8), the component *x[id]* is obtained (Line 9). However, to make sure all the related components have been calculated out, all threads have to wait until the whole components in the set are solved (Line 10). Such synchronizations can be costly in the execution time.

Algorithm 2. Level-Set SpTRSV Algorithm for $Lx = b$

```
1: Input: InputMatrix L, array b
2: Output: array x
3: for i = 0 to layer-1 do
4:    for k = layer_num[i] to layer_num[i+1]-1 in parallel do
5:       id ← order[k]
6:       left_sum ← 0
7:       for j = L.RowPtr[id] to L.RowPtr[id+1]-2 do
8:          left_sum← left_sum+L.Val[j]×x[L.ColIdx[j]]
9:       x[id]←(b[id]-left_sum)/L.Val[L.RowPtr[id+1]-1]
10:   _synchronize
```

2.3 Synchronization-Free SpTRSV

Because Level-Set SpTRSV method involves long preprocessing time and has a bottleneck in synchronization, Liu *et al.* [16] introduced a synchronization-free algorithm for GPUs in CSC format (similar to CSR format except that values are stored in column order). Another previous study [22] proposed a similar synchronization-free algorithm in CSR format. The basic idea is to add a new flag array get_value to show whether the component is solved or not and use a warp to compute a component in parallel according to the original row order of the input matrix, which avoids the synchronization and greatly reduces the processing time. Currently, the synchronization-free SpTRSV algorithm is the state-of-the-art SpTRSV algorithm.

Algorithm 3. Synchronization-Free SpTRSV Algorithm for $Lx = b$

```
1: Input: InputMatrix L, array b
2: Output: array x
3: MALLOC (*get_value, L.rows)
4: MEMSET (*get_value, 0)
5: for i = 0 to L.rows-1 in parallel do        ▷ One warp for one component.
6:    shared memory: left_sum[warp_size]
7:    for thread_id = 0 to warp_size-1 in parallel do        ▷ One thread for partial nonzeros
8:       sum ← 0
9:       for j = L.RowPtr[i]+thread_id to L.RowPtr[i+1]-2 Step warp_size do        ▷ Step means j+=warp_size.
10:         col ← L.ColIdx[j]
11:         while get_value[col] ≠ true do
12:            // busywait
13:         sum ← sum + L.Val[j] × x[col]
14:      left_sum[thread_id] ← sum
15:      for add_len=warp_size/2 to add_len > 0 Step add_len/=2 do
16:         if thread_id < add_len then
17:            left_sum[thread_id] ← left_sum[thread_id] + left_sum[thread_id+add_len]
18:      if thread_id = 0 then
19:         x[i]←(b[i]-left_sum[thread_id])/L.Val[L.RowPtr[i+1]-1]
20:         __threadfence()
21:         get_value[i] ← true
22: FREE (*get_value)
```

Synchronization-Free SpTRSV Algorithm. The detailed algorithm is shown in Algorithm 3. In Algorithm 3, the algorithm computes components in the original row order of the input

TABLE 1
Case Study for Preprocessing Time and Execution Time of Different SpTRSV Algorithms

Algorithm	Time (ms)	nlpkkt160	wiki-Talk	cant
Level-Set [14], [15]	Preprocessing	310.07	31.09	4.81
	Execution	28.07	12.89	28.79
cuSPARSE [21]	Preprocessing	16.24	1.99	0.28
	Execution	37.98	11.88	7.69
Sync-Free [16]	Preprocessing	8.07	0.42	0.28
	Execution	27.73	10.02	5.02

matrix and uses one warp (*warp_size* threads) to compute one row (Line 5). When calculating the nonzero elements in the row, each thread only computes part of elements in parallel (Lines 7-14). When a thread computes the element $l_{i,col}$, to make sure x_{col} is solved, the thread needs to wait until its flag *get_value[col]* is set to *true* (Lines 10-12), and then calculates the value (Line 13). Next, we add the intermediate results in the *warp_size* threads of a warp together in parallel with the shared array *left_sum* (Lines 15-17). After calculating the whole nonzero elements in row, we obtain the component x_i and set *get_value[i]* to *true* (Lines 19-21).

2.4 cuSPARSE Library

cuSPARSE Library [21] provides functions for SpTRSV directly. Since cuSPARSE is not open-sourced, we do not know the implementation details it adopts, and can only treat it as a black box. Compared to the performance of SpTRSV in cuSPARSE version 7.5 used in [23], the performance in cuSPARSE version 8.0 used in this paper doubles. It shows the significant improvement of SpTRSV in cuSPARSE, which can be viewed as a strong state-of-the-art approach for comparison.

2.5 Summary

We summarize the differences between the three parallel SpTRSV algorithms and test their performance with three random sparse matrices. As shown in Table 1, we can observe that the synchronization-free SpTRSV algorithm exhibits short preprocessing time and high performance. In comparison, the preprocessing time of the Level-Set SpTRSV algorithm is very long, which greatly limits their applicability. Other sparse matrices exhibit similar phenomena.

We also summarize the properties of current SpTRSV algorithms in Table 2, including the preprocessing time, storage format, synchronization, and granularity. Our findings are as follows. First, synchronization-free algorithm has low preprocessing overhead and high performance, which is the current trend for SpTRSV. Second, although the SpTRSV in cuSPARSE is not open source, we speculate that it now uses the synchronization-free SpTRSV algorithm due to the short preprocessing time. Third, to address the limitations of other approaches, our proposed *YuenyeungSpTRSV* is a thread-level and warp-level fusion synchronization-free approach with a very short preprocessing stage.

3 Revisiting Warp-Level Synchronization-Free SpTRSV

In this section, we first show our insights in the synchronization-free SpTRSV algorithm, including the limitations and opportunities, followed by an experimental study to motivate YuenyeungSpTRSV algorithm. Then, we present the technical challenges.

3.1 Motivation

Observation: Warp-level synchronization-free SpTRSV algorithms cannot fully utilize GPU resources when 1) the average number of components x per level is large, and 2) the average number of nonzero elements per row of the input sparse matrix L is small.

Insight: Previous synchronization-free SpTRSV designs are mainly based on 1) warp states (busy or idle) and 2) synchronization between warps, but ignore the thread states in warps. Hence, we call such warp-level SpTRSV coarse-grained. In contrast, we additionally consider both thread and wrap states, and both thread-level and warp-level synchronizations within and between warps, which is a mix, just like *Yuenyeung* (a popular beverage of coffee with tea in Hong Kong).

Although the synchronization-free SpTRSV algorithm [16] solves the performance bottleneck caused by synchronization, the GPU resource still could be underutilized, especially when 1) the average number of components x per level is large, and 2) the average number of nonzero elements per row is small. The reasons are as follows. First, the GPU device consists of a limited number of streaming multiprocessors (SM), and each SM consists of light-weight cores. The number of active warps for each SM is limited. If we use a warp to handle a component, then the number of components that can be processed simultaneously is limited in the SM. When the number of components x in a level is large enough that exceeds the SM threshold, the level has to be processed in several rounds. Second, the instructions for a warp are executed in a lock-step manner, which means that all threads in one warp need to execute the same

TABLE 2
Summary for Different SpTRSV Algorithms

Algorithm	Preprocessing overhead	Storage format	Synchronization required or not	Processing granularity
Level-Set	high	CSR	yes	thread/warp
Sync-Free	low	CSC	no	warp
cuSPARSE	low	CSR	unknown	unknown
YuenyeungSpTRSV	**very low**	**CSR**	**no**	**fusion**

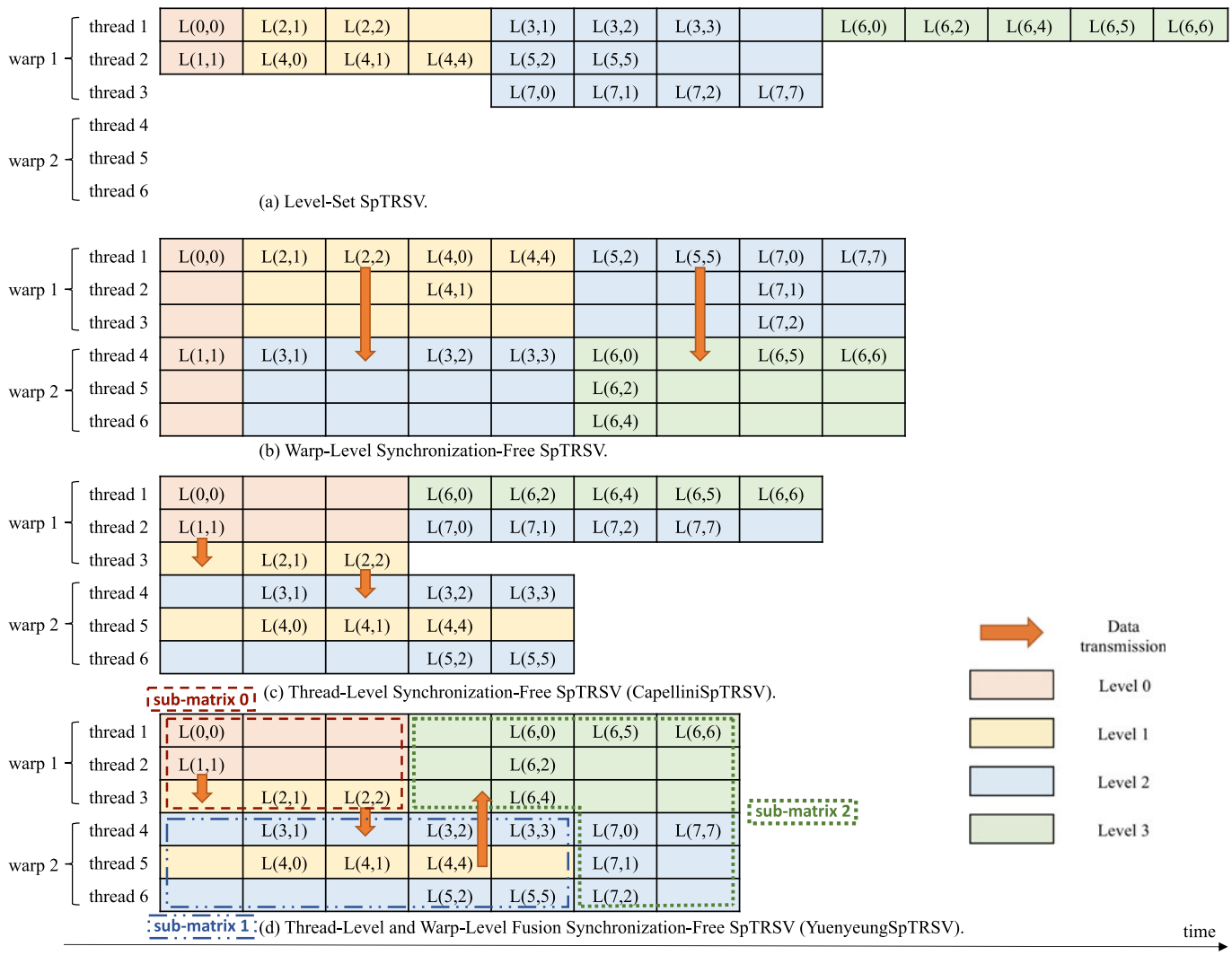


Fig. 2. An example to show the benefits from our YuenyeungSpTRSV. The partitions for sub-matrices in (d) are used to integrate thread-level and warp-level algorithms, detailed in Section 4.

instruction. Assume the warp size is *warp_size* (32 in Nvidia GPUs). When the related row of a component has fewer nonzero elements than *warp_size*, some threads will be idle and have to wait until the end of the warp execution.

Opportunities. A fine-grained thread-level and warp-level fusion synchronization-free SpTRSV could solve the limitations of current warp-level synchronization-free SpTRSV algorithms. First, when we handle the matrix parts with low parallel granularity (detailed in Section 3.2), a thread-level design could be applied without the limitations of warp-level synchronization-free SpTRSV algorithms. Second, when we handle other matrix parts with large parallel granularity, we remain to use the warp-level design to fully utilize the GPU capacities. Third, with such fusion design, we do not need to worry about whether a thread will be idle waiting or have imbalanced load in different situations. Before we show our experimental analysis, we use a case study for illustration.

Case Study. We show the SpTRSV workflow for different algorithms in Fig. 2. We use the matrix L of Fig. 1 as input. For simplicity, we assume the GPU device can launch two warps at the same time, and each warp can support three threads. First, in Fig. 2a, for Level-Set SpTRSV, although it can execute at thread level, the synchronization in the level-set design limits its parallelism. Second, in Fig. 2b, although the warp-level synchronization-free algorithm achieves performance improvement by removing synchronizations compared to Fig. 2a, there are still many idle threads. Note that for $L(4,4)$, *thread3* cannot handle it along with $L(4,0)$ and $L(4,1)$ because $L(4,4)$ needs to be integrated with the intermediate results after $L(4,0)$ and $L(4,1)$ are processed. Third, in Fig. 2c, which is our preliminary design [19], the overall efficiency is improved but there is still thread idle waiting; *thread3* of *warp1* is idle when *warp1* is solving components x_6 and x_7, and *warp2* is not fully used. Fourth, in Fig. 2d, the thread-level and warp-level fusion SpTRSV design utilizes the GPU better, but there exist several challenges, which shall be discussed in Section 3.3.

3.2 Experimental Study

We use real sparse matrices from the University of Florida Sparse Matrix Collection [20] to analyze the performance of warp-level synchronization-free SpTRSV algorithm. Before we show our experimental findings, we need to design an indicator for describing the parallelism in sparse matrices.

Parallel Granularity. We define a new indicator, *parallel granularity*, as shown in Equation (1) to describe the

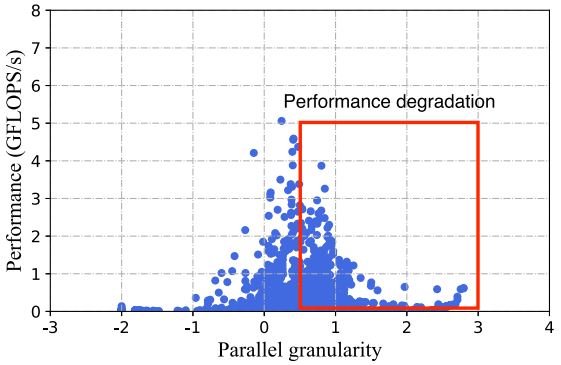


Fig. 3. Performance trend of warp-level synchronization-free SpTRSV. The performance declines after reaching the peak state.

influence from the two factors: 1) the average number of components per level n_{level}, and 2) the average number of nonzero elements per row nnz_{row}. The larger n_{level}, the worse the performance. The reason is that for warp-level SpTRSV, if the number of components x in a level exceeds the supported number of warps on GPUs, the level has to be processed in several rounds. The larger nnz_{row}, the better the performance, since a large nnz_{row} can reduce idle states. We mainly use the logarithm function to normalize n_{level} and nnz_{row} in our analysis, because these two factors show a different range of values. We add bias of b_1 and b_2 in Equation (1) to avoid numerical errors. The parameters of bases and bias in Equation (1) can be adjusted by users; by default, we use common logarithm where the all the bases are 10, and b_1 and b_2 are 0.01 in Equation (1). For other values of these parameters, the performance trend is similar.

$$parallel\_granularity = log_{c_1}\left(\frac{log_{c_2}(n_{level})}{log_{c_3}(nnz_{row} + b_1)} + b_2\right). \tag{1}$$

Performance Trend. The performance trend of the current warp-level synchronization-free SpTRSV is shown in Fig. 3. As the increase of parallel granularity, the SpTRSV performance increases at first, and then declines. The reason is that as the parallel granularity increases, the GPU resources are underutilized: more idle states appear in threads. A thread-level and warp-level fusion synchronization-free SpTRSV could help when the performance declines.

3.3 Challenges

We present the technical challenges for developing YuenyeungSpTRSV.

Challenge 1: Fusion of Thread-Level and Warp-Level Algorithms. To further improve the performance of SpTRSV, we propose a warp-level design and thread-level fusion design in Fig. 2d, which means that we integrate the warp-level design of Fig. 2b and the thread-level design of Fig. 2c together. However, we encounter two major difficulties. First, we need to develop a segmentation method to allocate the rows with fewer nonzero elements to be processed at thread level, and allocate the other rows to be processed at warp level. Additionally, the segmentation should not disrupt the row order. Second, both warp-level and thread-level algorithms coexist at the same kernel execution, which could cause new partitioning issues. For example, assume we plan to process *row0* and *row1* at thread level, and process *row2* at warp level. If we use *thread1* and *thread2* to process *row0* and *row1* separately, and use *threads 3 to 5* to process *row2*, then both thread-level and warp-level algorithms are executed in the same warp (*theads 1 to 3*), which causes deadlock; however, if use another warp such as *warp2* to process *row2*, *thread3* is wasted.

Challenge 2: Avoiding Deadlocks. Previous deadlock solution designs of warp-level synchronization-free SpTRSV do not work at thread level. Previous methods [16], [22] usually use a *while*-loop to constantly check whether the related value has been updated. Because the threads in a warp of the warp-level algorithms are designed to update the same value, they do not have deadlocks. In thread-level design, the threads in one warp may have dependencies. For example, if our program simply requires processing all the elements before updating the component, then *thread2* and *thread3* in Fig. 2d shall incur deadlocks. Because *thread2* and *thread3* are in the same warp, when *thread3* constantly checks x_1 for $L(2,1)$, according to the GPU execution manner [22], *thread2* also executes the same instructions, but does not update the status of x_1.

Challenge 3: Last Element Checking. In SpTRSV, when processing a nonzero element in a row, we need to verify whether the processed element is on the diagonal since the element on the diagonal is the last element and processing the last element means that the related component x_i is ready to be calculated. A common solution is to add an *if* statement for checking the last element before processing each nonzero element. However, such last element checking causes runtime overhead. For example, in the process of *thread5* in Fig. 2d, last element checking happens before *thread5* processing $L(4,0)$ and $L(4,1)$, which should be removed. In our experiments, such as matrix *nlpkkt160*, this overhead can cause 27.3 percent performance slowdown.

Challenge 4: Thread Execution Model. Although we can use a thread to handle one component, the GPUs are still executed in the warp execution mode. In detail, the threads in the same warp have to transmit the required components simultaneously. For example, in Fig. 2d, *thread6* requires x_2 for processing $L(5,2)$, which can only be obtained after the third cycle. However, if we simply use a conditional *while*-loop to check the condition to move on, *thread6* starts this checking from the beginning and the *thread4* and *thread5* within the same warp also need to wait for *thread6* in the constant condition check, which means that the processing of $L(3,1)$ and $L(4,0)$ also needs to be postponed to the fourth cycle though their required x_1 and x_0 are ready at the second cycle.

4 Overview of YuenyeungSpTRSV

We show YuenyeungSpTRSV in Fig. 4, which integrates both the thread-level and warp-level synchronization-free SpTRSVs. In detail, it identifies the components that cause GPU underutilization at warp level, and processes these components and their related rows in sparse matrices at thread level. For the rest of components, YuenyeungSpTRSV remains to use the warp-level algorithm.

We next show our four novel designs in Yuenyeung SpTRSV, and then discuss how these designs solve the challenges mentioned in Section 3.3.

Design to Integrate Thread-Level and Warp-Level Algorithms. We develop a light-weight fusion solution, which avoids

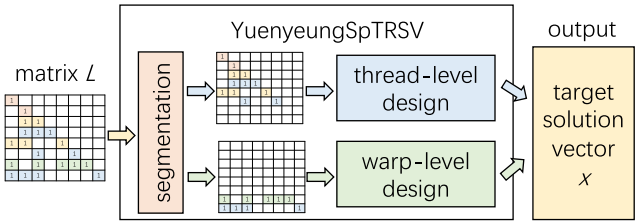


Fig. 4. YuenyeungSpTRSV illustration.

warp-level and thread-level SpTRSVs being executed in the same warp and at the same time, ensures that no threads are idle. In detail, we divide the input matrix into multiple sub-matrices with *warp_size* rows. In Fig. 2d, we divide *matrix L* into three sub-matrices, *sub-matrix0* from *row0* to *row2*, *sub-matrix1* from *row3* to *row5*, and *sub-matrix2* from *row6* to *row7*. If we process a sub-matrix at warp level, the number of required warps is the number of rows in each sub-matrix, which is *warp_size* except the last sub-matrix. For example, in Fig. 2d, we process *sub-matrix2* at warp level, where *warp1* handles *row6* and *warp2* handles *row7*. If the sub-matrix is solved at thread level, then we need only one warp (*warp_size* threads) to handle it, where each thread solves one component. For example, in Fig. 2d, *sub-matrix0* is solved by *warp1*, and *sub-matrix1* is solved by *warp2*. With this design, the warp-level and thread-level algorithms coexist together with no idle threads.

Design to Avoid Deadlocks. We propose a two-phase mechanism to avoid the deadlocks in YuenyeungSpTRSV. We divide the computation process of a warp into two phases. The first phase is for the elements in the related row of matrix L that has no inter-dependency within a warp. These elements can be processed directly and do not cause the deadlock problem. The busy-waiting strategy can be applied here to obtain the uncalculated data. For example, in Fig. 2d, *thread4* in *warp2* waits x_2 from *thread3* in *warp1*. The second phase relates to the rest of the elements in the row that have inter-dependency within the warp. Instead of using an endless loop, we use a *for-loop* and the number of loops is the *warp* size: we guarantee the data that need to be transmitted shall be put into the target place within a period of *warp-size* loops. For example, in Fig. 2d, *thread3* waits one loop for x_1 from *thread2* in the same warp to process $L(2,1)$.

Efficient Last Element Checking. As discussed in Challenge 3 of Section 3.3, last elements refer to the elements on the diagonal of matrix L. Since the time-consuming part is the constant *if* checking for the last elements, a possible optimization is to reduce the number of such last element checkings. We further analyze the SpTRSV process, and find that to process element $L(i,j)$, the component x_j needs to be ready. Consequently, the last element checking can be integrated into the element processing: if x_j is ready, then the related $L(i,j)$ must not be on the diagonal (x_j is the target to be calculated for row *j*) and thus is not the last element of row *i*. Therefore, we only need to check the element whose relevant component x_j is not ready. For example, in the process of *thread5* in Fig. 2d, *thread5* obtains x_0 for $L(4,0)$ and x_1 for $L(4,1)$, and do not need to make further last element checking.

Adaptation to GPU Thread Execution. Because GPUs execute in warps, we do not distribute components during warp execution. Instead, we distribute tasks at the beginning of the warp execution. For example, in Fig. 2d, we do not distribute the task for *row3* of the component x_3 to *thread4* during the warp execution; we distribute *row3* to *thread4* along with *row4* to *thread5* and *row5* to *thread6*, but *thread4* is in a waiting state. After the component x_1 has been processed, $L(3,1)$ can be processed. Similar process also happens for *thread5* and *thread6*, which wait until the components x_0 and x_2 are ready. With this strategy, our thread-level execution can adapt to the current warp-based GPU architectures. Furthermore, we propose a *Writing-First* optimization in Section 5.3 that threads can compute the elements and write the partial results first without waiting for the other threads. For example, in Fig. 2d, *thread4* and *thread5* can compute elements $L(3,1)$ and $L(4,0)$ without waiting $L(5,2)$, and *thread5* can compute the component x_4 in the fourth cycle without waiting *thread4* and *thread6*.

Features. In addition to addressing the challenges above, YuenyeungSpTRSV has the following desirable features.

- *Strong effectiveness*. By addressing the limitations of existing approaches, YuenyeungSpTRSV supports sparse matrices that have high parallel granularity, which enables the synchronization-free SpTRSV design to be efficient for various sparse matrices.
- *CSR format*. YuenyeungSpTRSV adopts the most popular CSR format, so that users do not need to conduct format transformation.
- *Very low preprocessing time*. YuenyeungSpTRSV does not need to calculate levels or convert formats, so the preprocessing time is very low and it can be easily applied to various situations.

In the rest parts of the paper, we start with our design of thread-level synchronization-free SpTRSV (Section 5), followed by the fusion of thread-level and warp-level designs which shows how to integrate the warp-level optimization to our thread-level design (Section 6), and then our detailed implementation (Section 7).

5 Thread-Level Design

Following the general design in Section 4, we show our thread-level synchronization-free SpTRSV in this section, which mainly derived from CapelliniSpTRSV [19].

5.1 Algorithm Design

In this part, we show our first version of thread-level synchronization-free SpTRSV in a two-phase manner.

Overview. The thread-level design does not need preprocessing. Our thread-level SpTRSV computes the components in the original row order of the input sparse matrix L. As discussed in Section 4, the first phase is used to handle the elements in the row of matrix L that have no inter-dependency in a warp, and the second phase is for the rest elements that have dependencies.

Detailed Algorithm. We show our Two-Phase Yuenyeung SpTRSV in Algorithm 4. In the algorithm, each thread computes a row or a component in the original row order of the matrix. According to the prior paragraph, we divide the elements of the row into two groups according to the dependencies within a warp. Because the threads compute the components in order, there is only a border *warp_begin* we need to compute to divide the elements (Line 6). We first

compute the elements without the inter-warp dependency (Lines 8-15) in the first phase, since these elements do not cause the deadlock issue. In this group, we use the traditional busy-waiting method (Lines 11-12).

Algorithm 4. Two-Phase YuenyeungSpTRSV

```
 1: Input: InputMatrix L, array b
 2: Output: array x
 3: MALLOC (*get_value, L.rows)
 4: MEMSET (*get_value, 0)
 5: for i = 0 to L.rows-1 in parallel do          ▷ One thread for one component
 6:   warp_begin ← (i/warp_size) × warp_size
 7:   left_sum ← 0
 8:   for j = L.RowPtr[i] to L.RowPtr[i+1]-2 do          ▷ Phase 1
 9:     col ← L.ColIdx[j]
10:     if col < warp_begin then
11:       while get_value[col] ≠ true do
12:         // busywait
13:       left_sum ← left_sum + L.Val[j] × x[col]
14:     else
15:       break
16:   col ← L.ColIdx[j]
17:   for k = 0 to warp_size-1 do                   ▷ Phase 2
18:     while get_vlaue[col] = true do
19:       left_sum ← left_sum + L.Val[j] × x[col]
20:       j ← j + 1
21:       col ← L.ColIdx[j]
22:     if col = i then
23:       x[i]←(b[i]-left_sum)/L.Val[L.RowPtr[i+1]-1]
24:       __threadfence()
25:       get_value[i] ← true
26:       j ← j+1
27:       break
28: FREE (*get_value)
```

After calculating the elements without inter-warp dependency, we compute the interdependent elements in the second phase. Because components only depend on previous ones, after computing all the components outside the warp, the warp can solve at least one component in each *for-loop*. Hence, the maximum number of loops for computing the components in a warp is equal to the warp size *warp_size*, and we set the number of iterations for the *for-loop* to the warp size (Line 17). Since threads in the same warp execute synchronously, the traditional busy-waiting method cannot be used. Instead, the threads have to check the finishing conditions. The first condition is whether the current element has been computed or not. If the element is computed (Line 18), then the algorithm accumulates its value (Line 19) and moves to the next element in the same row (Lines 20-21). The second condition is whether the current element is the last one in the row (Line 22). The variable *col* is the column number of the element. If *col* is equal to the last one of the row (Line 22), then the algorithm will calculate and save the component's related value (Line 23), and set the array *get_value* to *true* (Line 25) to tell the other threads that the component is solved.

5.2 Limitation of Two-Phase Design

Before we introduce our final thread-level synchronization-free SpTRSV, we revisit Algorithm 4 shown in Section 5.1.

For the first phase, the *while-loop* (Line 11) has a runtime issue due to the busy waiting for the threads in the warp: before the computation in Line 13, the thread needs to wait for *get_value[col]* to be set to *true*; even worse, the other threads in the same warp also need to wait due to the internal warp execution mechanism in GPUs. For example, in Fig. 2c, *thread6* waits until the fourth cycle to process $L(5,2)$; however, due to the *while-loop* (Line 11), the computations of $L(3,1)$ for *thread4* and $L(4,0)$ for *thread5* also need to be postponed to the fourth cycle. For the second phase (Line 17), the premise of starting the second phase is that all threads in the same warp have finished the calculation of all nonzero elements whose relevant components have been computed in the other warps. Due to the warp-level synchronous execution in GPUs, for the threads that have finished their first-phase computation, they still have to wait for the other threads in the same warp to enter the second-phase in Line 17. For example, in Fig. 2c, *thread5* cannot process $L(4,4)$ directly after the computation for $L(4,1)$, but needs to wait for the processing of $L(3,2)$ and $L(5,2)$ in Line 16.

5.3 Control Flow Optimization

To solve the above performance limitation, we design a Writing-First YuenyeungSpTRSV, which removes the computing part for the elements without inter-warp dependency (the first phase), and expands the scope of the computation from the inter-warp dependent elements (the second phase) to the whole elements in the row.

Algorithm 5. Writing-First YuenyeungSpTRSV

```
 1: Input: InputMatrix L, array b
 2: Output: array x
 3: MALLOC (*get_value, L.rows)
 4: MEMSET (*get_value, 0)
 5: for i = 0 to L.rows-1 in parallel do          ▷ One thread for one component
 6:   left_sum ← 0
 7:   j ← L.RowPtr[i]
 8:   while j < L.RowPtr[i+1] do
 9:     col ← L.ColIdx[j]
10:     while get_value[col] = true do
11:       left_sum ← left_sum + L.Val[j] × x[col]
12:       j ← j + 1
13:       col ← L.ColIdx[j]
14:     if i = col then
15:       x[i]←(b[i]-left_sum)/L.Val[L.RowPtr[i+1]-1]
16:       __threadfence()
17:       get_value[i] ← true
18:       j ← j + 1
19:       break
20: FREE (*get_value)
```

Detailed Algorithm. We show our Writing-First Yuenyeung SpTRSV in Algorithm 5. In this algorithm, each thread computes a component, which relates to a row, in the original row order of the matrix (Line 5). The variable *j* is equal to the location of the current computing element in the CSR-format matrix (Line 7), and the variable *col* is equal to the column number of the current element (Line 9). There are two conditions to check. The first one is about whether the current computing element is solved. If it is *true* (Line 10), then the

algorithm accumulates its value (Line 11) and moves to the next element in the same row (Lines 12-13). The second condition is whether the current element is the last one or not. If *col* is equal to the last one in the row (Line 14), then, the algorithm shall calculate and save the related values of the component (Line 15), and set the related value in the array *get_value* to *true* (Line 17) to tell the other threads that the component is ready.

6 Fusion Design of YuenyeungSpTRSV

After introducing the thread-level design in Section 5, in this section, we show how to integrate it with the warp-level synchronization-free SpTRSV. We first show our general design of YuenyeungSpTRSV, and then show our segmentation method in preprocessing, followed by the detailed algorithm design. YuenyeungSpTRSV involves novel cross-GPU optimizations, including data structures to represent different processing levels, a lightweight model to predict the configuration, and adaptation to GPU architectures.

6.1 Fusion of Thread-Level and Warp-Level SpTRSV

We show our design in combining our thread-level SpTRSV (Algorithm 5 in Section 5) with previous warp-level synchronization-free SpTRSV (Algorithm 3 in Section 2.3) in this part.

Analysis. As discussed in Section 3.3, a warp of threads needs to be regarded as a whole to process components at warp level or thread level. To handle such limitations, we propose the following design rules. First, if we use one thread of a warp to compute one component with one row in the input sparse matrix, which represents the thread-level SpTRSV, then the other threads within the same warp also have to process components at thread level; otherwise, we use the whole warp of threads to compute one component with one row, which represents the warp-level SpTRSV. Second, due to the warp-specific limitation on GPUs, the sparse matrix needs to be segmented at warp granularity to avoid assigning both thread-level and warp-level SpTRSVs to one warp. In detail, for a group of rows of continuous warp size, it can only select either warp-level or thread-level SpTRSV to be processed. Third, an efficient mapping mechanism needs to be developed to map threads to components (rows in the sparse matrix) at thread level or warp level.

Our Approach. We add an additional data structure to represent different processing levels. To map the threads to rows of the sparse matrix at both thread and warp levels efficiently, we add a buffer *execute_row_id* to store the start position for each warp. For *warp i*, if its related number of rows "$execute\_row\_id[i+1]$-$execute\_row\_id[i]$" is the warp size, then this warp processes rows of warp size at thread level; if "$execute\_row\_id[i+1] - execute\_row\_id[i]$" is one, the warp processes one row at warp level. We show an example in Fig. 5. Assume a warp contains *n* threads and there are *m* warps in the system. The threads in *warp1* process *rows 1 to n* at thread level, because the *address2* minus *address1* in *execute_row_id* equals the warp size *n*. The threads in *warp2* process *row n+1* at warp level, because the *address3* minus *address2* in *execute_row_id* equals one. With this adaptation to GPU architectures, thread-level and warp-level kernels co-run in YuenyeungSpTRSV.

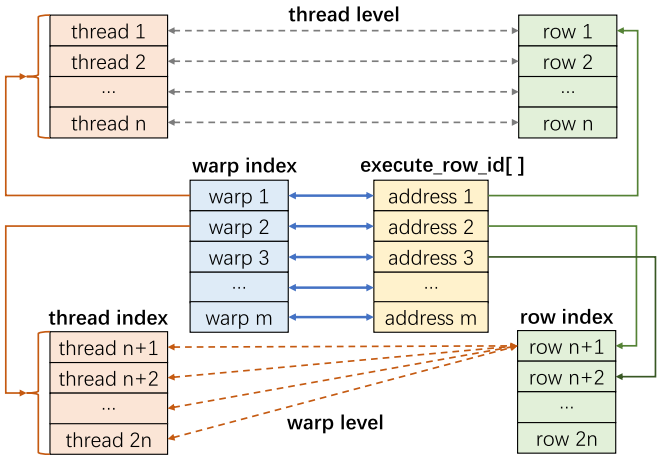


Fig. 5. Warp-level and thread-level fusion SpTRSV design.

6.2 Detecting Threshold

As discussed in Section 3, the thread-level SpTRSV is good at processing sub-matrices with high parallel granularity, while the warp-level SpTRSV is suitable for sub-matrices with low parallel granularity. Hence, we need to define a threshold to distinguish whether to use thread-level design or warp-level design.

Analysis. We first analyze the selection criteria. Parallel granularity has two influencing factors: the first is the average number of components per level n_{level}, and the second is the average number of nonzero elements per row nnz_{row}. However, calculating the number of components in each level needs to identify the level where the row is located, and then requires counting the number of components in each level, which incurs large time overhead.

Our Approach. We build a lightweight model to predict the configuration. First, we can have a preprocessing phase to determine the threshold. During the procedure to find a suitable threshold, the preprocessing time should be very short, so that our YuenyeungSpTRSV can have a wide range of application scenarios. Second, to minimize the preprocessing time, we only use the average number of nonzero elements per row nnz_{row} to select the processing level. After we identify a threshold, if the nnz_{row} of a sub-matrix exceeds the threshold, the warp-level SpTRSV is used, and each row is computed by one warp; otherwise, the thread-level SpTRSV is used, and the entire sub-matrix is calculated by one warp. Third, the threshold could be platform dependent, which means that the thresholds on different platforms could be different.

Detailed Design. We prepare a training set of 1,000 matrices generated from Graph 500 [24] with various parallel granularities. We keep the lower triangular part of the sparse matrices and calculate the number of nonzero elements per row for training. For each matrix, we compare the number of nonzero elements per row with its performance on both warp-level synchronization-free SpTRSV and thread-level synchronization-free SpTRSV. Note that the training set needs to be executed only once when a platform is available, and the threshold is determined after the training process. Then, when a sparse matrix comes, we only need to calculate the average number of nonzero elements per row for row segmentation and assign an appropriate processing method to each row by setting *execute_row_id*.

6.3 YuenyeungSpTRSV Algorithm Design

In this part, we illustrate our thread-level and warp-level fusion synchronization-free SpTRSV in Algorithm 6. The *ALGCHOOSE* function is used to assign different processing methods to sub-matrices.

Algorithm 6. YuenyeungSpTRSV

```
1: Input: InputMatrix L, array b
2: Output: array x
3: execute_row_id, array_len = function ALGCHOOSE
4: for warp = 0 to array_len-2 in parallel warp do
5:   if execute_row_id[warp+1] - execute_row_id[warp] > 1 then
6:     use thread-level SpTRSV to compute rows in a warp
7:   else
8:     use warp-level SpTRSV to compute a row in a warp
9:
10: Function ALGCHOOOSE(InputMatrix L)
11:   warp_id ← 0
12:   for row_start = 0 to L.rows-1 step warp_size do
13:     row_end ← minimum(row_start+warp_size, L.rows)
14:     avg_element_row ← (L.RowPtr[row_end]-L.RowPtr
        [row _start]) / (row_end-row_start)
15:     if avg_element_row ≥ threshold then        ▷ warp-level
16:       for i = row_start to row_end-1 do
17:         execute_row_id[warp_id] ← i
18:         warp_id ← warp_id + 1
19:     else                                       ▷ thread-level
20:       execute_row_id[warp_id] ← row_start
21:       warp_id ← warp_id + 1
22:   execute_row_id[warp_id] ← L.rows
23:   warp_id ← warp_id + 1
24:   array_len ← warp_id
25:   Return array execute_row_id, array_len
```

YuenyeungSpTRSV. In Algorithm 6, *execute_row_id* stores the start location for each warp and *array_len* stores the length of *execute_row_id*; "*array_len-1*" is the number of warps need to be executed. For each warp, if it needs to handle multiple rows, which is *warp_size* rows in default (Line 5), YuenyeungSpTRSV calls the thread-level design for the sub-matrix (Algorithm 5); otherwise, it calls the warp-level design (Algorithm 3).

Segmentation. The *ALGCHOOSE* function is our segmentation algorithm. In the ALGCHOOSE function, *row_start* (Line 12) is the first row number of the sub-matrix, and *row_end* (Line 13) is its last row number. The sub-matrices have *warp_size* rows except the last sub-matrix. The average number of the nonzero elements per row in the sub-matrix is stored in variable *avg_element_row* (Line 14). We use the threshold described in Section 6.2; if *avg_element_row* is greater than the threshold, which means that the number of nonzero elements in each row of this sub-matrix is large, YuenyeungSpTRSV selects warp-level synchronization-free SpTRSV to handle this sub-matrix (Line 15). We use one warp to process one row, so we set the row location for each warp of the sub-matrix in *array execute_row_id* (Lines 16-18). If *avg_element_row* is less than the threshold, indicating that the average number of nonzero elements in each row of this sub-matrix is small, YuenyeungSpTRSV selects the thread-level design for this sub-matrix (Line 19). In thread-level design, YuenyeungSpTRSV needs one warp to solve a sub-matrix, so we record only the first-row location of the sub-matrix for this warp in *array execute_row_id* (Lines 20-21). For the last warp, we set the last element of *array execute_row_id* to the total number of rows of the input sparse matrix (Line 22).

Applicability. The idea behind YuenyeungSpTRSV is not limited to SpTRSV and can be used for other sparse problems, especially in DAG-based situations that involve massive dependencies. In this work, the SpTRSV process can be represented as a DAG traversal, as shown in Fig. 1, where each node in the DAG tries to solve the related component x_i. In addition, our idea can be applied to the other irregular task scheduling situations. For example, a large application can be divided into several modules with dependencies, and the modules without dependencies can be executed in parallel. Another example is the query plan optimization in database domain. A SQL query can be represented as a DAG of operators. In these cases, the amount of computation of each operator node is different, and we can choose different methods to handle each node based on the amount of computation. In this way, our idea can be applied to efficiently execute such irregular computations on GPU with synchronization-free thread-level and warp-level adaptation.

7 Cross-Platform Implementation

We provide cross-platform YuenyeungSpTRSV implementation for two purposes. The first purpose is to ease programmers' burden in porting YuenyeungSpTRSV to various platforms. To this end, we provide not only CUDA implementation but also OpenCL implementation, which is similar to [23]. The CUDA module is used for the SpTRSV on Nvidia GPUs, while the OpenCL module is used for the other platforms, such as AMD GPUs. The second purpose is to provide a light-weight high-performance SpTRSV implementation, which can help existing applications directly.

8 Evaluation

In this section, we evaluate YuenyeungSpTRSV in comparison with the state-of-the-art synchronization-free and cuSPARSE SpTRSV algorithms.

8.1 Experimental Setup

Methods. Our SpTRSV algorithm is denoted as "Yuenyeung". We compare our YuenyeungSpTRSV with the state-of-the-art synchronization-free SpTRSV algorithm [23], which is denoted as "SyncFree". Because cuSPARSE [21] is very popular and has been widely used in various areas, we also compare our algorithm with the SpTRSV in cuSPARSE. Moreover, we compare our work to CapelliniSpTRSV, denoted as "Capellini", which is our preliminary work presented in [19] with only thread-level designs. We do not further analyze level-set based methods due to their excessive preprocessing time, as discussed in Section 2.5. Because for SpTRSV, precision is very important [16], [23], [25], we mainly focus on the double precision.

Platforms. We measure the performance of the SpTRSV algorithms on four experimental platforms, as shown in Table 3, including three generations of Nvidia GPUs (Pascal, Volta, and Turing micro architectures) and an AMD APU.

Datasets. We randomly download 873 sparse matrices, whose numbers of nonzero elements are larger than 100,000, from the University of Florida Sparse Matrix Collection [20], which have been widely used in previous research [16], [23].

TABLE 3
Platform Configuration

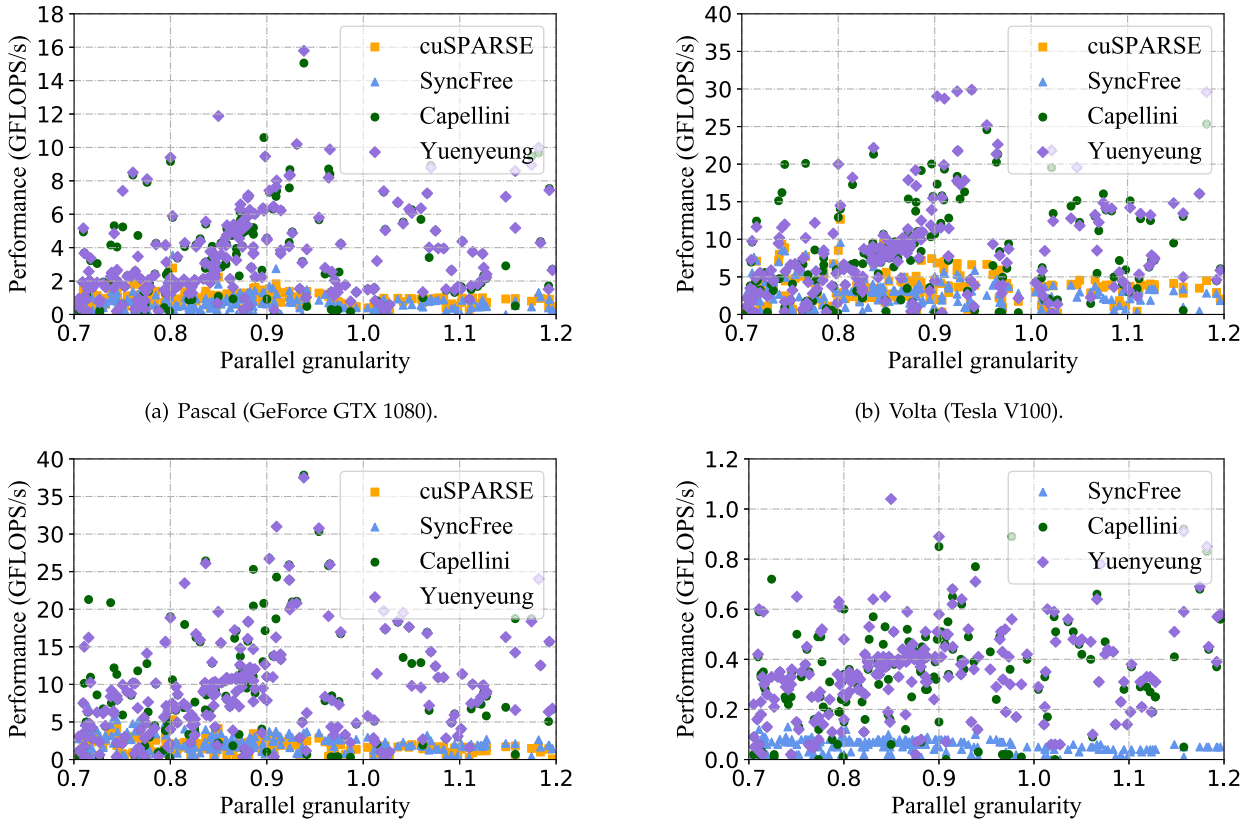
Platform	Pascal	Volta	Turing	APU
GPU	GTX 1080	V100	RTX 2080 Ti	Radeon Vega 11
Memory	GDDR5X	HBM2	GDDR6	DDR4
CPU	i7-7700K	E5-2640	i9-9900K	Ryzen 5 2400G
OS	Ubuntu 16.04.4	Ubuntu 16.04.1	Ubuntu 18.04.4	Ubuntu 18.04.3
Compiler	CUDA 8	CUDA 9	CUDA 10.2	ROCm

To ensure the matrices are lower triangular (we use unit-lower triangular here), we keep only the lower-left elements and assign values to the diagonal elements. The average number of nonzero elements per row is 19.6, and the average number of components per level is 12484.9. As Fig. 3 in Section 3.2, the performance of SyncFree SpTRSV decreases after the parallel granularity is larger than 0.7. Therefore, we mainly focus on the sparse matrices with parallel granularity larger than 0.7, which include 245 matrices. We use the same matrices as in [19]. These matrices come from various domains: 42.0 from graph applications, 13.9 percent from circuit simulations, 11.0 percent from combinatorial problems, 9.4 percent from linear programming problems, and 8.6 percent from optimization problems.

8.2 Performance

YuenyeungSpTRSV targets sparse matrices with high parallel granularity. We show the performance of different algorithms in this part, which proves the effectiveness of our SpTRSV algorithm.

GFLOPS. Experiments show that on all platforms, YuenyeungSpTRSV exhibits the highest performance in the matrices with parallel granularity larger than 0.7. We show the performance results for different algorithms on various GPU platforms when the parallel granularity ranges from 0.7 to 1.2 in Fig. 6, which shows that YuenyeungSpTRSV brings significant performance benefits. We show the average performance for different algorithms on the four platforms in Table 4. On average, YuenyeungSpTRSV achieves a performance of 7.14 GFLOPS/s, while the SyncFree SpTRSV achieves only 1.79 GFLOPS/s on Nvidia GPUs, which implies that YuenyeungSpTRSV successfully handles the matrices that previous work cannot handle in an efficient manner. The SpTRSV in cuSPARSE can also achieve a performance of 1.93 GFLOPS/s. Our YuenyeungSpTRSV achieves the highest performance for 95.28 percent of the matrices on the four platforms. In Fig. 6, YuenyeungSpTRSV exhibits similar performance on both Volta and Turing platforms, but much lower performance on the Pascal platform. The reason is that the Pascal platform has much fewer number of GPU cores and lower



(a) Pascal (GeForce GTX 1080).

(b) Volta (Tesla V100).

(c) Turing (GeForce RTX 2080 Ti).

(d) APU (Ryzen 5 2400G).

Fig. 6. Performance for different SpTRSVs.

TABLE 4
The GFLOPS of Different SpTRSV Algorithms and the Percentage of Matrices That Achieve the Optimal Performance Using YuenyeungSpTRSV

Platform	Pascal	Volta	Turing	Average	Apu
SyncFree	0.652	2.721	1.983	1.785	0.052
cuSPARSE	0.903	3.245	1.636	1.928	none
CapelliniSpTRSV	3.413	8.091	9.028	6.844	0.333
YuenyeungSpTRSV	**3.613**	**8.601**	**9.204**	**7.139**	**0.364**
Percentage (%)	96.33	92.38	97.14	95.28	100.00

TABLE 5
The Average and Maximum Speedups Over SyncFree and cuSPARSE on Different Platforms

Platform	Pascal	Volta	Turing	Apu
Average speedup over SyncFree	5.49	4.14	5.71	8.56
Maximum speedup over SyncFree	19.89	36.50	45.93	60.00
Matrix name	*lp1*	*lp1*	*lp1*	*lp_ken_18*
Average speedup over cuSPARSE	4.20	3.12	7.16	none
Maximum speedup over cuSPARSE	22.20	25.83	75.33	none
Matrix name	*watson_2atmosmodd*		*sls*	none

memory bandwidth. However, YuenyeungSpTRSV achieves the highest performance for more than 90 percent of the matrices on all platforms.

Speedup. To further elaborate the benefits of Yuenyeung SpTRSV over the other SpTRSVs when the parallel granularity is large, we show the performance speedup of YuenyeungSpTRSV over the SyncFree and cuSPARSE algorithms in Table 5. On average, YuenyeungSpTRSV achieves 5.98x speedup over the SyncFree SpTRSV, and 4.83x speedup over the cuSPARSE SpTRSV for these matrices when the parallel granularity is larger than 0.7. We show the performance speedup of YuenyeungSpTRSV over SyncFree SpTRSV in Fig. 7, and we can see that the performance benefits increase along with the parallel granularity.

Algorithm Preference Distribution. As shown in Section 3.2, warp-level SpTRSV (SyncFree) has low performance when

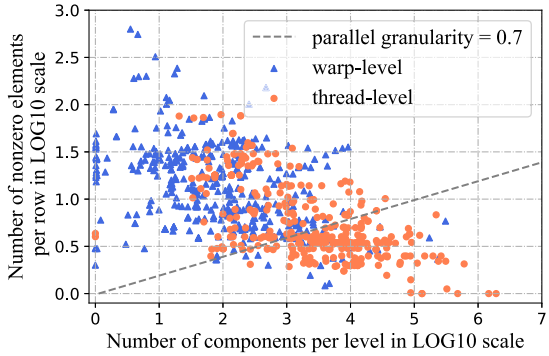


Fig. 8. Optimal algorithm distribution on Turing (GeForce RTX 2080 Ti).

the parallel granularity of matrices is high. Thread-level SpTRSV (Capellini) has high performance on matrices with high parallel granularity. Fig. 8 further proves our idea. The parameter of parallel granularity relates to two factors of 1) the average number of components per level n_{level} and 2) the average number of nonzero elements per row nnz_{row}. We show the optimal algorithm selection between warp-level design and thread-level design under different factors of n_{level} and nnz_{row} in Fig. 8 on Turing GPU. The thread-level design is better when n_{level} is high and nnz_{row} is low, while the warp-level design is better when n_{level} is low and nnz_{row} is high. YuenyeungSpTRSV integrates the advantages of both designs.

8.3 Benefits of YuenyeungSpTRSV Over Thread-Level SpTRSV

To quantify the benefits of warp-level and thread-level fusion design over the thread-level SpTRSV design, we compare YuenyeungSpTRSV with the thread-level SpTRSV design (CapelliniSpTRSV). We show the average and maximum speedups of YuenyeungSpTRSV over CapelliniSpTRSV in Table 6, and have the following observations. First, the fusion design achieves an average speedup of 1.86× over the thread-level design, which proves the effectiveness of YuenyeungSpTRSV. Second, the number of rows that are processed by warp-level SpTRSV is limited, but the proportion of nonzero elements of these rows are large. Third, the performance behavior on different architectures varies. For example, the matrices that achieve the maximum speedup are different on the four microarchitectures.

To better show the advantages of YuenyeungSpTRSV over CapelliniSpTRSV, we accumulate the processing time for each row of the three matrices in Table 6 on the Turing

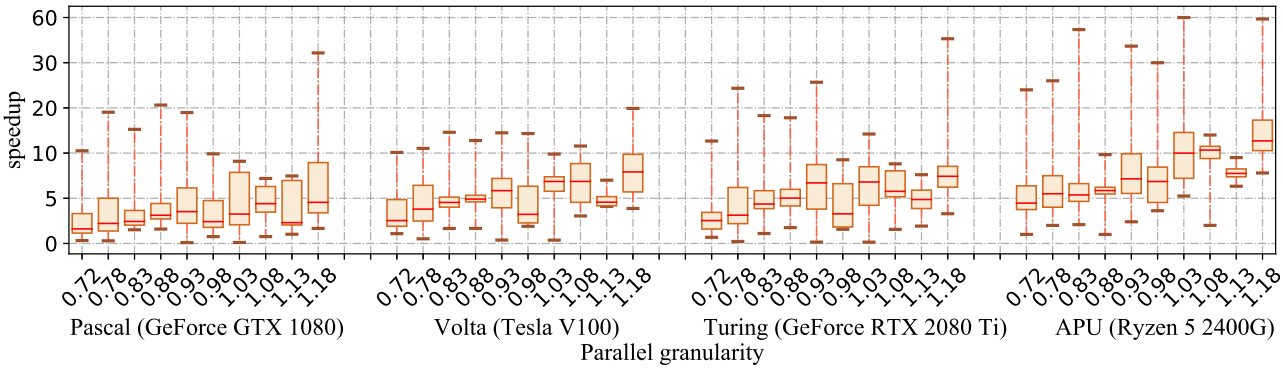


Fig. 7. Performance YuenyeungSpTRSV speedup over the SyncFree SpTRSV for different sparse matrices.

TABLE 6
The Average and Maximum Speedups Over Thread-Level SpTRSV on Different Platforms

Platform	Pascal	Volta	Turing	APU
Average speedup over thread-level	1.52	1.77	1.65	2.04
Maximum speedup over thread-level	13.00	22.13	13.00	30
Matrix name	*tp-6*	*circuit5M*	*tp-6*	*circuit_4*
Warp-level rows	160	50080	160	369
Warp-level row ratio (%)	0.11	0.90	0.11	0.46
Warp-level elements	142176	6333690	142176	44952
Warp-level element ratio (%)	32.74	19.46	32.74	23.61

Warp-level rows: the number of rows that are processed by the warp-level design. Warp-level row ratio: the proportion of warp-level rows. Warp-level elements: the number of nonzero elements processed by the warp-level design. Warp-level element ratio: the proportion of warp-level elements.

platform. We respectively accumulate the time to process the rows that should be processed in thread-level and warp-level designs, as shown in Fig. 9. Note that in CapelliniSpTRSV, both parts are processed in thread-level SpTRSV. In Fig. 9, the first part of each method (Capellini or Yuenyeung) represents the accumulated processing time of different rows that should be processed at thread level. The second part represents the accumulated time that should be processed at warp level, which accounts for about 30 percent in CapelliniSpTRSV. In YuenyeungSpTRSV, the second part has been significantly reduced to less than 4 percent. Moreover, the time in the first part of YuenyeungSpTRSV has also been reduced accordingly, which is due to the shorter time for processing the components at thread level waiting for components processed at warp level.

8.4 Reasons for Performance Improvement

To further exhibit the reasons for higher performance of YuenyeungSpTRSV, we perform a detailed analysis for our novel designs in Section 4. In this part, we use the Turing platform for illustration. The results of the other platforms are similar.

Fusion of Thread-Level and Warp-Level Algorithms. The fusion of thread-level and warp-level algorithms apply different algorithms to handle their appropriate parts. As shown in Fig. 8, the warp-level SpTRSV is suitable for matrices with high number of nonzero elements per row, while the thread-level SpTRSV is good at processing matrices with low number of nonzero elements per row. We show the average number of nonzero elements per row before and after partitioning for different algorithms in Fig. 10. The average number of nonzero elements per row of the original matrices is 3.44. After partitioning, for the warp-level part, its number increases to 148.43, which is more suitable for warp-level SpTRSV. In contrast, for the thread-level part, its number decreases to 2.73, which is more suitable for thread-level SpTRSV.

Deadlock Avoidance With Thread-Level Better Utilization. We propose a Two-Phase SpTRSV of Algorithm 4 to avoid deadlocks at thread level. For further thread-level better utilization, we develop Algorithm 5 of Writing-First strategy, which removes the computation for elements without inter-warp dependency, as discussed in Section 5.3. Such a strategy reduces the required number of instructions and better utilizes bandwidth. Experiments show that our optimization reduces 53.55 percent GPU instructions and improves 57.00x bandwidth utilization compared to the Two-Phase SpTRSV. Accordingly, the performance of our Writing-First SpTRSV is 55.05x over that of Two-Phase SpTRSV.

Efficiency in Last Element Checking. We reduce the number of last element checkings, as discussed in Section 4, which decreases the number of instructions. Fig. 11 shows the number of executed instructions. In general, YuenyeungSpTRSV saves 72.54 percent instructions compared to the SyncFree SpTRSV, and 94.65 percent instructions compared to the cuSPARSE SpTRSV. Such results indicate the effectiveness of the last element checking design in YuenyeungSpTRSV.

Adaptation to GPU Thread Execution. As discussed in Section 4, threads in YuenyeungSpTRSV compute the elements and write partial results without waiting for the other threads. Additionally, YuenyeungSpTRSV launches fewer

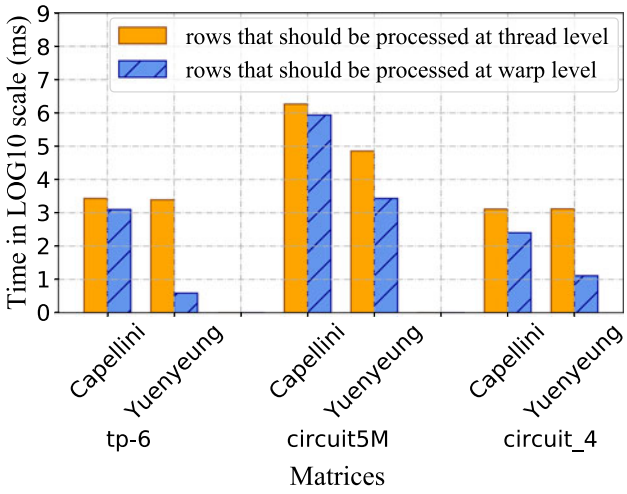


Fig. 9. Accumulated processing time for rows that should be processed at thread level and warp level.

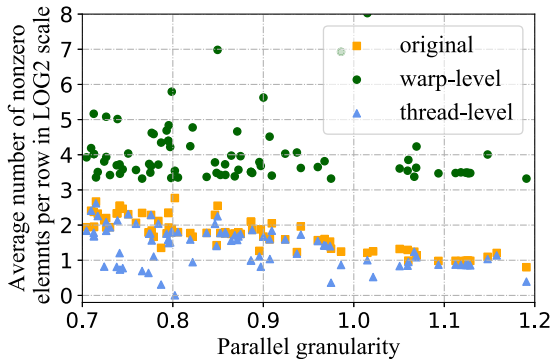


Fig. 10. Matrix partitioning for thread-level and warp-level algorithms.

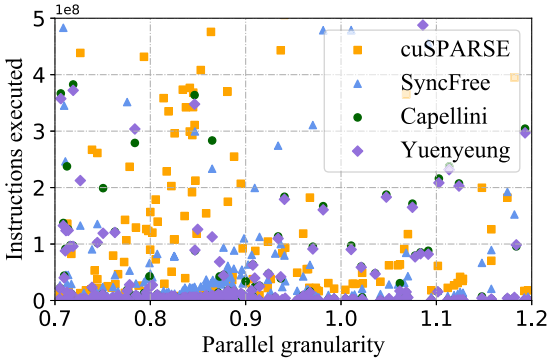


Fig. 11. Number of GPU instructions executed.

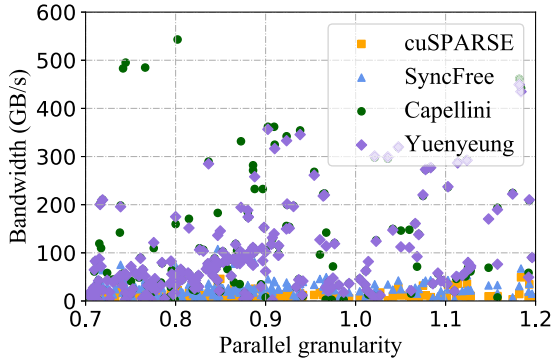


Fig. 13. Bandwidth utilization (sum of read and write bandwidth).

warps than the previous SyncFree SpTRSV, and our algorithm is also more concise. We compare the instruction stall percentage of different algorithms to show the benefits of the adaptation design in YuenyeungSpTRSV. Fig. 12 shows the instruction stall percentage. The value of our YuenyeungSpTRSV is 0.52 percent, which is 80.74 percent lower than that of SyncFree SpTRSV and 71.23 percent lower than that of cuSPARSE SpTRSV.

8.5 Detailed Analysis

In this section, we show the bandwidth utilization, the preprocessing time, and a case study for detailed analysis on the Turing platform.

Bandwidth. Fig. 13 shows the bandwidth utilization on the Turing platform. We use the Nvidia performance analysis tool, *ncu*, to obtain the DRAM read and write bandwidth. YuenyeungSpTRSV achieves an average bandwidth of 97.15 GB/s for the matrices whose parallel granularities are larger than 0.7. The bandwidth utilization of YuenyeungSpTRSV is 53.23x higher than that of the cuSPARSE SpTRSV, 4.60x higher than the SyncFree SpTRSV, and 1.59x higher than CapelliniSpTRSV, which proves the effectiveness of YuenyeungSpTRSV.

Preprocessing Time. We show the average preprocessing time in different algorithms in Table 7. YuenyeungSpTRSV exhibits the lowest preprocessing time. The reason is that YuenyeungSpTRSV only needs to scan the buffer that stores the number of nonzero elements, which is extremely lightweight.

Case Study. We randomly select six matrices, and show the detailed parameters of different SpTRSVs for the six matrices in Table 8. The matrices with high parallel granularities

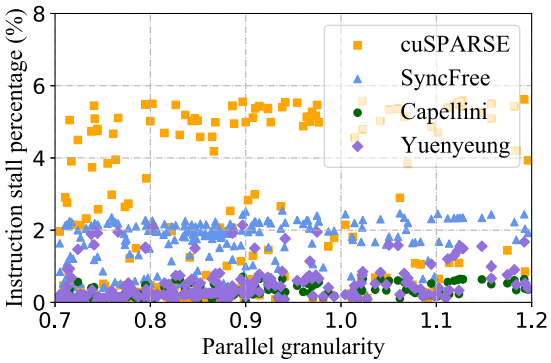


Fig. 12. Percentage of instruction dependency stalls.

TABLE 7
The Preprocessing Time in Different Algorithms

Preprocessing time	cuSPARSE	SyncFree	Yuenyueng
Average	1.11	0.36	0.30
Minimum	0.03	0.02	0.00
Maximum	24.01	8.63	10.56

TABLE 8
Detailed Performance Indicators for Six Matrices

Algorithm	Performance (GFLOPS/s)	Bandwidth (GB/s)	Instructions (10^6)	Stall (%)
cvxbqp1 (δ: 0.73; α: 4.00; β: 2000.00)				
cuSPARSE	2.49	7.55	18.05	2.32
SyncFree	4.04	24.80	5.66	2.05
Capellini	4.45	42.52	2.24	0.18
Yuenyeung	**6.37**	**42.53**	**2.17**	**0.16**
ncvxqp3 (δ: 0.76; α: 4.00; β: 3000.00)				
cuSPARSE	2.82	13.26	25.28	3.85
SyncFree	4.49	29.26	7.60	2.18
Capellini	6.19	57.66	3.26	0.17
Yuenyeung	**8.98**	**57.74**	**3.11**	**0.16**
luxembourg_osm (δ: 0.88; α: 2.04; β: 268.38)				
cuSPARSE	0.43	0.44	174.93	0.13
SyncFree	0.29	2.61	61.45	0.44
Capellini	0.68	8.05	17.86	0.11
Yuenyeung	**0.99**	**8.22**	**17.00**	**0.09**
rajat29 (δ: 0.78; α: 4.89; β: 14636.23)				
cuSPARSE	2.59	0.51	2932.32	0.08
SyncFree	0.84	7.44	351.50	0.41
Capellini	10.43	109.69	18.57	0.15
Yuenyeung	**12.65**	**121.25**	**16.79**	**0.22**
bayer01 (δ: 0.87; α: 3.39; β: 9622.50)				
cuSPARSE	2.55	12.26	16.32	4.99
SyncFree	4.22	27.28	5.60	2.24
Capellini	11.8	104.76	0.77	0.38
Yuenyeung	**12.76**	**78.14**	**0.86**	**0.79**
circuit5M_dc (δ: 0.92; α: 3.02; β: 12812.06)				
cuSPARSE	1.74	4.76	2981.90	1.45
SyncFree	2.06	28.07	536.12	1.53
Capellini	14.57	200.06	47.41	0.35
Yuenyeung	**20.11**	**201.47**	**46.72**	**0.49**

δ: parallel granularity. α: average number of nonzero elements per row. β: average number of components per level.

usually have low average number of nonzero elements per row and high average number of components per level. For these matrices, the bandwidth utilization and instruction efficiency of our YuenyeungSpTRSV are also better.

9 Related Work

SpTRSV is an important function in the matrix computing field, and has attracted a lot of research efforts.

Level-Set SpTRSV. Anderson and others [14] and Saltz and others [15] proposed that level-set methods can be used for the parallelism in sparse triangular solves. However, the synchronization barrier often limits the performance of parallel SpTRSV [23]. To address this problem, Naumov and others [26] developed a GPU-based level-set SpTRSV with a tradeoff to reduce the number of synchronizations. Further, Park and others [18] proposed a synchronization-sparsification optimization, which can largely decrease the synchronization overhead and improve the scalability.

Color-Set and Other SpTRSVs. Schreiber and Tang [27] first constructed color-sets for SpTRSV on multiprocessors by graph coloring. And Suchoski and others [28] extended the method to GPUs. Besides, Anzt and others [29] applied an iterative approach for an approximate SpTRSV solution using GPUs.

Synchronization-Free SpTRSV. Liu and others replaced the synchronization with atomic operations [16], [30] and developed a strategy for further parallelizing multiple right-hand sides [23] for a synchronization-free SpTRSV at warp level, which is the state-of-the-art SpTRSV algorithm. However, because this work is based on the warp level, for sparse matrices with high parallel granularity, this algorithm cannot fully utilize the GPU capacity. Different from this work, we propose YuenyeungSpTRSV, a thread-level and warp-level fusion SpTRSV targeting the sparse matrices with high parallel granularity, which can handle the limitation of the previous work.

Non-Uniform Distribution in Sparse Matrices. Irregular distribution in sparse matrices is a performance bottleneck on GPUs. There are many related studies, especially for sparse solvers. Yan *et al.* [31] proposed yaSPMV, which solves SpMV's load imbalance and high memory bandwidth problems through a segmented scan approach. Liu *et al.* [32] presented a GPU-based SpGEMM algorithm to handle irregularity from nonzero entries, parallel insertions, and load balancing. In addition to algorithm adaptation, programming model (Groute) [33], task aggregation (ATA) [34], and irregular input transformation (Tigr) [35] have been developed to make irregular applications more efficient on GPUs. Different from these works, YuenyeungSpTRSV needs to handle mixed operation of warp level and thread level algorithms under dependent conditions on GPU, which is much more complicated.

Matrix Optimization. In addition to the algorithms, researchers also proposed other strategies to accelerate matrix computing, such as the storage format of the matrix and the access speed to the memory. Kulkarni and others [36] designed an optimistic parallelization system, called Galois, for irregular applications. They also introduced a structural analysis and a data-centric formulation of algorithms for the irregular data structures, which reveal a generalized form of data-parallelism and this parallelism can be used by inspector-executor, compiling, or optimistic parallelization [37]. Zhang and others [38] removed dynamic irregularities through data reordering and job swapping to improve the performance on GPUs. Similarly, Wu and others [39] proposed novel data reorganization algorithms to minimize the non-coalesced memory accesses caused by irregular references. Picciau and others [40] recently proposed a method that partitions the graphical form of an input matrix into multiple subgraphs for balancing concurrency and data access locality. Rodríguez and others [41] partitioned the irregular computation of sparse matrices into a union of regular parts, which can then be optimized by polyhedral compilers.

10 Conclusion

SpTRSVs have been extensively used in linear algebra fields, and many GPU-based SpTRSV algorithms have been proposed. In this paper, we identified their limitations, and developed YuenyeungSpTRSV that efficiently supports the sparse matrices with high parallel granularities, which cannot be handled efficiently by previous algorithms. YuenyeungSpTRSV involves novel cross-GPU optimizations, including data structures to represent different processing levels, a lightweight model to predict the configuration, and adaptation to GPU architectures, and we provide cross-platform implementations. YuenyeungSpTRSV can be applied to a wide range of HPC applications, such as iterative solver and direct solver. Experiments show that YuenyeungSpTRSV achieves 5.98x performance speedup over the state-of-the-art synchronization-free SpTRSV and 4.83x speedup over the SpTRSV in Nvidia cuSPARSE. Moreover, our proposed YuenyeungSpTRSV is based on the most popular CSR format and does not require preprocessing to calculate levels.

Reproducibility

We support reproducible science. YuenyeungSpTRSV is available as a free open-source SpTRSV solve on GitHub (https://github.com/JiyaSu/YuenyeungSpTRSV), Mulan Open Source Community (https://toscode.gitee.com/JiyaSu/YuenyeungSpTRSV), and Code Ocean.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1004401, in part by the National Natural Science Foundation of China under Grant 61802412, Grant 61732014, and Grant 61972415, in part by the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant CARCHA202007, in part by the and Science Challenge Project under Grant TZZT2016002. The work of Bingsheng He was part supported by a MoE AcRF Tier 1 Grant (T1 251RES1824) and Tier 2 Grant (MOE2017-T2-1-122) in Singapore.

References

[1] Å. Björck, *Numerical Methods for Least Squares Problems*. Philadelphia, PA, USA: SIAM, 1996.
[2] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*. Oxford, U.K.: Oxford Univ. Press, 2017.
[3] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: SIAM, 2003.

[4] W. Liu, "Parallel and scalable sparse basic linear algebra subprograms," PhD dissertation, Faculty Sci., Univ. Copenhagen, Copenhagen, Denmark, 2015.

[5] W. Liu and B. Vinter, "A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors," *J. Parallel Distrib. Comput.*, vol. 85, pp. 47–61, 2015.

[6] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli, "Sparse matrix-matrix multiplication on modern architectures," in *Proc. 19th Int. Conf. High Perform. Comput.*, 2012, pp. 1–10.

[7] W. Liu and B. Vinter, "CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proc. 29th ACM Int. Conf. Supercomput.*, 2015, pp. 339–350.

[8] W. Liu and B. Vinter, "Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors," *Parallel Comput.*, vol. 49, pp. 179–193, 2015.

[9] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 769–780.

[10] M. Daga and J. L. Greathouse, "Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices," in *Proc. IEEE 22nd Int. Conf. High Perform. Comput.*, 2015, pp. 64–74.

[11] N. Sedaghati *et al.*, "Automatic selection of sparse matrix representation on GPUs," in *Proc. 29th ACM Int. Conf. Supercomput.*, 2015, pp. 99–108.

[12] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *Proc. ACM/IEEE Conf. Supercomput.*, 2007, pp. 1–12.

[13] H. Wang *et al.*, "Parallel transposition of sparse data structures," in *Proc. Int. Conf. Supercomput.*, 2016, Art. no. 33.

[14] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *Int. J. High Speed Comput.*, vol. 1, pp. 73–95, 1989.

[15] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM J. Sci. Statist. Comput.*, vol. 11, pp. 123–144, 1990.

[16] W. Liu *et al.*, "A synchronization-free algorithm for parallel sparse triangular solves," in *Proc. Eur. Conf. Parallel Process.*, 2016, pp. 617–630.

[17] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *J. Supercomput.*, vol. 63, pp. 443–466, 2013.

[18] J. Park *et al.*, "Sparsifying synchronization for high-performance shared-memory sparse triangular solver," in *Proc. Int. Supercomput. Conf.*, 2014, pp. 124–140.

[19] J. Su *et al.*, "CapelliniSpTRSV: A thread-level synchronization-free sparse triangular solve on GPUs," in *Proc. 49th Int. Conf. Parallel Process.*, 2020, Art. no. 2.

[20] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, 2011, Art. no. 1.

[21] M. Naumov *et al.*, "Cusparse library," in *Proc. GPU Technol. Conf.*, 2010.

[22] E. Dufrechou and P. Ezzatti, "Solving sparse triangular linear systems in modern GPUs: A synchronization-free algorithm," in *Proc. 26th Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2018, pp. 196–203.

[23] W. Liu *et al.*, "Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides," *Concurrency Comput.: Pract. Experience*, vol. 29, 2017, Art. no. e4244.

[24] R. C. Murphy *et al.*, "Introducing the Graph 500," *Cray User's Group (CUG)*, vol. 19, pp. 45–74, 2010.

[25] X. Wang *et al.*, "swSpTRSV: A fast sparse triangular solve with sparse level tile layout on sunway architectures," *ACM SIGPLAN Notices*, vol. 53, pp. 338–353, 2018.

[26] M. Naumov, "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU," NVIDIA Corp., Westford, MA, Tech. Rep. NVR-2011–001, 2011.

[27] R. Schreiber and W.-P. Tang, "Vectorizing the conjugate gradient method," Unpublished manuscript, Department of Computer Science, Stanford University, 1982.

[28] B. Suchoski, C. Severn, M. Shantharam, and P. Raghavan, "Adapting sparse triangular solution to GPUs," in *Proc. 41st Int. Conf. Parallel Process. Workshops*, 2012, pp. 140–148.

[29] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *Proc. Eur. Conf. Parallel Process.*, 2015, pp. 650–661.

[30] Z. Lu, Y. Niu, and W. Liu, "Efficient block algorithms for parallel sparse triangular solve," in *Proc. 49th Int. Conf. Parallel Process.*, 2020, Art. no. 63.

[31] S. Yan, C. Li *et al.*, "yaSpMV: Yet another SpMV framework on GPUs," in *Proc. 19th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2014, pp. 107–118.

[32] W. Liu and B. Vinter, "An efficient GPU general sparse matrix-matrix multiplication for irregular data," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 370–381.

[33] T. Ben-Nun *et al.*, "Groute: An asynchronous multi-GPU programming model for irregular computations," in *Proc. 22nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2017, pp. 235–248.

[34] A. E. Helal, A. M. Aji, M. L. Chu, B. M. Beckmann, and W. Feng, "Adaptive task aggregation for high-performance sparse solvers on GPUs," in *Proc. 28th Int. Conf. Parallel Archit. Compilation Techn.*, 2019, pp. 324–336.

[35] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for GPU-friendly graph processing," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2018, pp. 622–636.

[36] M. Kulkarni *et al.*, "Optimistic parallelism requires abstractions," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2007, pp. 211–222.

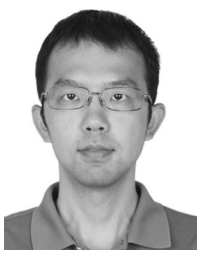
[37] K. Pingali *et al.*, "The tao of parallelism in algorithms," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2011, pp. 12–25.

[38] E. Z. Zhang *et al.*, "On-the-fly elimination of dynamic irregularities for GPU computing," in *Proc. 16th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2011, pp. 369–380.

[39] B. Wu *et al.*, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU," in *Proc. 18th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2013, pp. 57–68.

[40] A. Picciau, G. E. Inggs *et al.*, "Balancing locality and concurrency: Solving sparse triangular systems on GPUs," in *Proc. IEEE 23rd Int. Conf. High Perform. Comput.*, 2016, pp. 183–192.

[41] G. Rodríguez and L.-N. Pouchet, "Polyhedral modeling of immutable sparse matrices," in *Proc. 8th Int. Workshop Polyhedral Compilation Techn.*, 2018.



Feng Zhang received the bachelor's degree from Xidian University, China, in 2012, and the PhD degree in computer science from Tsinghua University, China, in 2017. He is an associate professor with the Key Laboratory of Data Engineering and Knowledge Engineer (MOE), Renmin University of China, China. His major research interests include high performance computing, heterogeneous computing, and parallel and distributed systems.

Jiya Su received the bachelor's degree from the School of Information, Renmin University of China, China, in 2020. She is currently working toward the graduate degree in computer science at the Illinois Institute of Technology, Chicago, Illinois. She joined the Key Laboratory of Data Engineering and Knowledge Engineer (MOE), in 2019, and now works with Rujia. Her research interests include high performance computing, distributed and parallel systems, and processing in memory.

Weifeng Liu received the BE and ME degrees in computer science, both from the China University of Petroleum, Beijing, China, in 2002 and 2006, respectively, and the PhD from the Niels Bohr Institute, University of Copenhagen, Denmark, in 2016. He is a full professor with the Department of Computer Science and Technology, China University of Petroleum, Beijing, China. His major research interests include high performance computing and mathematical software.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University, China, in 2003, and the PhD degree in computer science from the Hong Kong University of Science and Technology, Hong Kong, in 2008. He is an associate professor with the School of Computing, National University of Singapore, Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.

Ruofan Wu is currently working toward the fourth-year undergraduate degree in the School of Information, Renmin University of China, China. She joined the Key Laboratory of Data Engineering and Knowledge Engineer (MOE), in 2019. Her research interests include high performance computing, heterogeneous computing, and parallel accelerating.

Xiaoyong Du received the the BS degree from Hangzhou University, Zhengjiang, China, in 1983, the ME degree from the Renmin University of China, Beijing, China, in 1988, and the PhD degree from the Nagoya Institute of Technology, Nagoya, Japan, in 1997. He is currently a professor with the School of Information, Renmin University of China, China. His current research interests include databases and intelligent information retrieval.

Rujia Wang (Member, IEEE) received the bachelor's degree from Zhejiang University, China, and the MS and PhD degrees in electrical and computer engineering from the University of Pittsburgh, Pittsburgh, Pennsylvania. She is now an assistant professor in computer science at the Illinois Institute of Technology, Chicago, Illinois. Her research interests are in the broader computer architecture and systems area, including scalable, secure, reliable and high-performance memory systems and architectures.