CapelliniSpTRSV: A Thread-Level Synchronization-Free Sparse Triangular Solve on GPUs

Jiya Su[◇], Feng Zhang[◇], Weifeng Liu[★], Bingsheng He[+], Ruofan Wu[◇], Xiaoyong Du[◇], Rujia Wang[‡]
[◇]Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China
[★]Super Scientific Software Laboratory, Department of Computer Sci & Tech, China University of Petroleum - Beijing
[+]School of Computing, National University of Singapore
[‡]Computer Science Department, Illinois Institute of Technology
Jiya_Su@ruc.edu.cn, fengzhang@ruc.edu.cn, weifeng.liu@cup.edu.cn, hebs@comp.nus.edu.sg, 2017202106@ruc.edu.cn, duyong@ruc.edu.cn, rwang67@iit.edu

ABSTRACT

Sparse triangular solves (SpTRSVs) have been extensively used in linear algebra fields, and many GPU-based SpTRSV algorithms have been proposed. 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 *CapelliniSpTRSV*, a thread-level synchronization-free SpTRSV algorithm. Particularly, CapelliniSpTRSV has three novel features. First, unlike the previous studies, CapelliniSpTRSV does not need preprocessing to calculate levels. Second, CapelliniSpTRSV exhibits high performance on matrices that previous SpTRSVs cannot handle efficiently. Third, CapelliniSpTRSV's optimization does not rely on 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 CapelliniSpTRSV with 245 matrices from the Florida Sparse Matrix Collection on three GPU platforms, and experiments show that our SpTRSV exhibits 6.84 GFLOPS/s, which is 4.97x speedup over the state-of-the-art synchronization-free SpTRSV algorithm, and 4.74x speedup over the SpTRSV in cuSPARSE. CapelliniSpTRSV is open-sourced in https://github.com/JiyaSu/CapelliniSpTRSV.

ACM Reference Format:
Jiya Su[◇], Feng Zhang[◇], Weifeng Liu[★], Bingsheng He[+], Ruofan Wu[◇], Xiaoyong Du[◇], Rujia Wang[‡]. 2020. CapelliniSpTRSV: A Thread-Level Synchronization-Free Sparse Triangular Solve on GPUs. 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/10.1145/3404397.3404400

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPP '20, August 17–20, 2020, Edmonton, AB, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404400

1 INTRODUCTION

Sparse triangular solves (SpTRSVs) are widely used in linear algebra fields, and have been indispensable building blocks in many numerical linear algebra routines, such as least-squares problems [4], direct methods [8], and preconditioners of sparse iterative solvers [34]. 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 [19], such as sparse matrix-matrix multiplication [22, 26], sparse matrix-vector multiplication [6, 11, 12, 23, 24, 37], and sparse transposition [45], 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. 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.

Many parallel SpTRSV algorithms on GPUs have been developed in recent years. To address the dependency problem, a level-set SpTRSV algorithm has been proposed [1, 35], 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 [20]; in our experiment, the preprocessing time could be dozens of times to the execution time of solving SpTRSV itself. Moreover, Li et al. [16] pointed out that the inter-level synchronization incurs large performance overhead in the level-set SpTRSV.

Although recent level-set SpTRSV optimizations, such as simplifying synchronization by pruning [30] and replacing synchronization by atomic operations [20], reduce the number of synchronizations, the synchronization overhead is still prohibitively high. Later, Liu and others [20] 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 consider 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 avoided for the usability and applicability of SpTRSV.

To solve the challenges above, we propose **CapelliniSpTRSV**, a thread-level synchronization-free SpTRSV algorithm, to address 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 CapelliniSpTRSV is that we use one thread to solve one component, which avoids the resource waste caused by idle threads. Moreover, in order to improve SpTRSV performance in a holistic manner, CapelliniSpTRSV has three novel features. First, unlike the previous studies, CapelliniSpTRSV does not need preprocessing phase to calculate the levels in advance. Second, CapelliniSpTRSV exhibits high performance on matrices that have high parallel granularities, which is complementary to current warp-level synchronization-free SpTRSVs. Third, CapelliniSpTRSV's optimization does not rely on 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.

We evaluate CapelliniSpTRSV with 245 matrices from the University of Florida Sparse Matrix Collection [7] on three GPU platforms, and compare our method with the state-of-the-art SpTRSV algorithm [20] and the SpTRSV in cuSPARSE [28]. The experimental results show that CapelliniSpTRSV exhibits high efficiency for the matrices that have high parallel granularity. CapelliniSpTRSV achieves on average 4.97x performance speedup over the state-of-the-art SpTRSV algorithm [20], and 4.74x speedup over the SpTRSV in cuSPARSE.

The remainder of the paper is organized as follows. Section 2 present the preliminaries and a summary about current SpTRSV algorithms. Section 3 presents the insights and experimental studies as motivations, followed by the design in Section 4. We present the experiments in Section 5, and review related work in Section 6. Finally, we conclude in Section 7.

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. Figure 1 illustrates a sparse triangular matrix L in SpTRSV. Figure 1 (a) shows an 8-by-8 sparse triangular matrix, which can be divided into four level sets, as shown in Figure 1 (b). The matrix in Figure 1 (a) can be further stored in Figure 1 (c). 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.

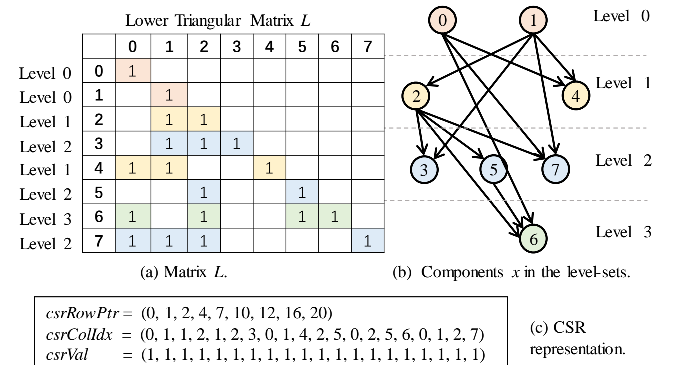


Figure 1: Lower triangular matrix L in CSR format: (a) the color shows the level of the row; (b) dependency of the components x. Each component relates to one row, and there are four level-sets in L; (c) the CSR format.

Basic SpTRSV Algorithm We show the basic SpTRSV in Algorithm 1. The algorithm traverses all rows (Line 1). In each row, it calculates all elements in the row except the last one, and stores the value in intermediate variable $left_sum$ (Lines 3-4). At last, the component of the solution vector x in the same row is solved (Lines 5-6).

Algorithm 1 Basic SpTRSV Algorithm for $Lx = b$

```
1: for i = 0 to m − 1 do                                   ▷ m is the number of rows.
2:     left_sum ← 0
3:     for j = csrRowPtr[i] to csrRowPtr[i+1]-2 do
4:         left_sum ← left_sum+csrVal[j]×x[csrColId[j]]
5:     xi ← (b[i] - left_sum) / csrVal[csrRowPtr[i+1]-1]
6:     x[i] ← xi
```

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 [1, 35], the algorithm stores the level-set number in *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 2), where *id* is the row number to solve (Line 3). After calculating the whole nonzero elements in the row (Lines 4-6), the component x_{id} is obtained (Lines 7-8). 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 9). Such synchronizations can be costly in the execution time.

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

```
1: for i = 0 to layer − 1 do
2:     for k=layer_num[i] to layer_num[i+1]-1 in parallel do
3:         id ← order[k]
4:         left_sum ← 0
5:         for j=csrRowPtr[id] to csrRowPtr[id+1]-2 do
6:             left_sum ← left_sum + csrVal[j] × x[csrColIdx[j]]
7:         xi ←(b[id]-left_sum)/csrVal[csrRowPtr[id+1]-1]
8:         x[id] ← xi
9:     _synchronize
```

2.3 Synchronization-Free SpTRSV

Because Level-Set SpTRSV method involves long preprocessing time and has a bottleneck in synchronization, Liu et al. [20] 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 [9] 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.

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 matrix and uses one warp (*warp_size* threads) to compute one row (Line 3). When calculating the nonzero elements in the row, each thread only computes part of elements in parallel (Lines 8-12). 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* is set to *true* (Lines 10-11), and then calculates the value (Line 12). Next, we add the intermediate results in the *warp_size* threads of a warp together in parallel with the shared array *left_sum* (Lines 13-17). After calculating the whole nonzero elements in row, we obtain the component x_i and set *get_value*[i] to *true* (Lines 18-22).

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

```
1: MALLOC (*get_value, m)                                  ▷ m is the number of rows.
2: MEMSET (*get_value, 0)
3: for i = 0 to m − 1 in parallel do           ▷ One concurrent warp for one
   component.
4:     MALLOC (*left_sum,WARP_SIZE)
5:     MEMSET (*left_sum, 0)
6:     for thread_id=0 to WARP_SIZE-1 in parallel do ▷ One thread for
   one nonzero
7:         sum ← 0
8:         for j=csrRowPtr[i]+thread_id to csrRowPtr[i+1]-2 Step
   WARP_SIZE do                                ▷ Step means j+=WARP_SIZE.
9:             col = csrColIdx[j]
10:            while (get_value[col] ≠ true) do
11:                // busywait
12:            sum ← sum + csrVal[j] × x[col]
13:        left_sum[thread_id] ← sum
14:        add_len ← 16
15:        while add_len > 0 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:            xi ←(b[i]-left_sum[thread_id])/csrVal [csrRowPtr[i+1]-
   1]
20:            x[i] ← xi
21:            __threadfence()
22:            get_value[i] ← true
23: FREE (*get_value)
```

2.4 cuSPARSE Library

cuSPARSE Library [28] 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 [21], 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 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.

Table 1: Case study for preprocessing time and execution time of different SpTRSV algorithms.

Algorithm	Time (ms)	nlpkkt160	wiki-Talk	cant
Level-Set [1, 35]	Preprocessing	310.07	31.09	4.81
	Execution	28.07	12.89	28.79
cuSPARSE [28]	Preprocessing	16.24	1.99	0.28
	Execution	37.98	11.88	7.69
Sync-Free [20]	Preprocessing	8.07	0.42	0.28
	Execution	27.73	10.02	5.02

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 **CapelliniSpTRSV** is a synchronization-free approach at thread level and without a preprocessing stage.

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
CapelliniSpTRSV	**none**	**CSR**	**no**	**thread**

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 CapelliniSpTRSV 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 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 thread states and thread-level synchronization within warps, which is *fine-grained*, just like *Capellini* (a very slender kind of Italian pasta between 0.85 and 0.92 millimeters in diameter).

Although the synchronization-free SpTRSV algorithm [20] 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 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 synchronization-free SpTRSV at thread level could solve the limitations of the warp-level synchronization-free SpTRSV algorithm. First, when we handle components at the thread level, we use one thread to solve one component, which is equivalent to expand $warp_size$ times the granularity that can be parallelized. This is useful when the number of components in a level is large. Second, we do not need to worry about whether the thread will be idle waiting when the average number of non-zero elements per row is small. Before we show our experimental analysis, we use a case study for illustration.

Case study. We show the SpTRSV workflow for different algorithms in Figure 2. We use the matrix L of Figure 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 Figure 2 (a), for Level-Set SpTRSV, although it can execute at thread level, the synchronization in the level-set design limits its parallelism. Second, in Figure 2 (b), although the warp-level synchronization-free algorithm achieves performance improvement by removing synchronizations compared to Figure 2 (a), 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 Figure 2 (c), the thread-level SpTRSV design utilizes the GPU better, but there exist inter- and intra-warp communications, 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 [7] 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 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 larger nnz_{row}, the better the performance. 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

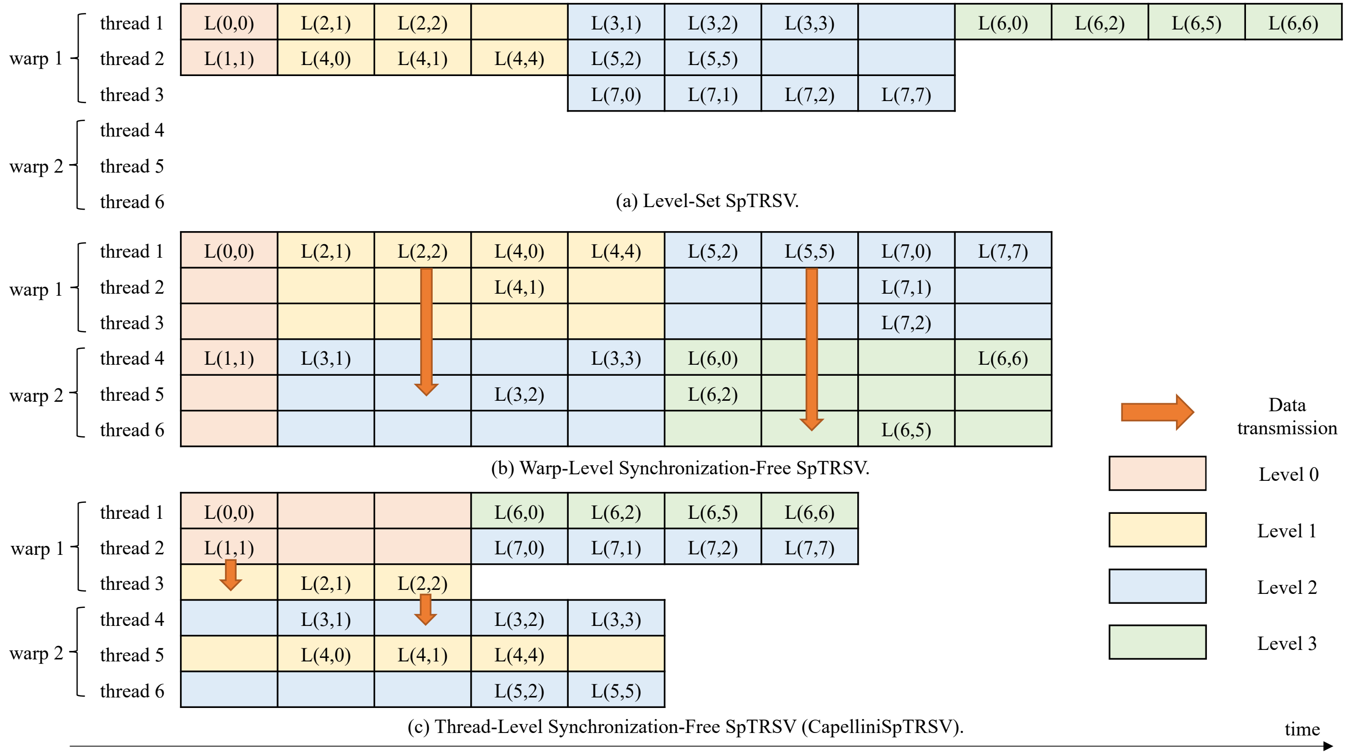


Figure 2: An example to show the benefits from our CapelliniSpTRSV.

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.

$$parallel_granularity = log_{c_1}\left(\frac{log_{c_2}(n_{level})}{log_{c_3}(nnz_{row} + b_1)} + b_2\right) \quad (1)$$

Performance trend. The performance trend of the warp-level synchronization-free SpTRSV is shown in Figure 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 and insufficient GPU parallelism happens. A thread-level synchronization-free SpTRSV could help when the performance declines.

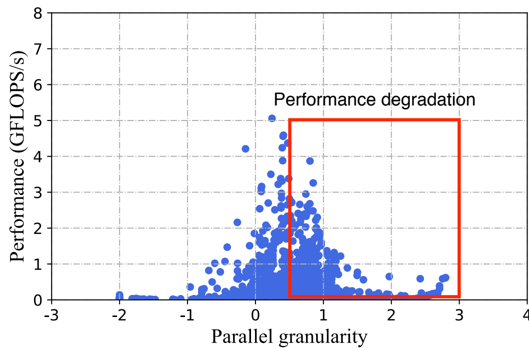


Figure 3: Performance trend of warp-level synchronization-free SpTRSV. The performance declines after reaching the peak state.

3.3 Challenges

We present the technical challenges for developing CapelliniSpTRSV.

Challenge 1: avoiding deadlocks. Previous deadlock solution designs of warp-level synchronization-free SpTRSV do not work at thread level. Previous methods [9, 20] 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 Figure 2 (c) 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 [9], $thread2$ also executes the same instructions, but does not update the status of x_1.

Challenge 2: 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 Figure 2 (c), 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% performance slowdown.

Challenge 3: thread execution model. Although we 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 Figure 2 (c), $thread6$ 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 CapelliniSpTRSV

We present CapelliniSpTRSV, our thread-level synchronization-free SpTRSV. We start with a design overview, followed by the basic CapelliniSpTRSV algorithm and then our optimized algorithm.

4.1 Design Overview

We first show our three novel designs in CapelliniSpTRSV, and then discuss how these designs solve the challenges mentioned in Section 3.3.

Design to avoid deadlocks. We propose a two-phase mechanism to avoid the deadlocks in CapelliniSpTRSV. 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 Figure 2 (c), $thread4$ 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 Figure 2 (c), $thread3$ 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 2 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 Figure 2 (c), $thread5$ 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 Figure 2 (c), 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 4.3 that threads can compute the elements and write the partial results first without waiting for the other threads. For example, in Figure 2 (c), $thread4$ 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, CapelliniSpTRSV also have the following desirable features.

- **No preprocessing**. CapelliniSpTRSV does not involve any preprocessing, so that our algorithm can be easily applied to various situations.
- **Strong effectiveness**. By addressing the limitations of existing approaches, CapelliniSpTRSV supports sparse matrices that have high parallel granularity, which enables the synchronization-free SpTRSV design to be efficient for various sparse matrices.
- **CSR format**. CapelliniSpTRSV adopts the most popular CSR format, so that users do not need to conduct format transformation.

4.2 Algorithm Design

Following the general design in Section 4.1, we propose a *Two-Phase CapelliniSpTRSV* in this section.

Overview. As there is no preprocessing step, CapelliniSpTRSV computes the components in the original row order of the sparse matrix. As discussed in Section 4.1, 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 CapelliniSpTRSV 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 4). We first compute the elements without the inter-warp dependency (Lines 6-13) in the first phase, since these elements do not cause the deadlock issue. In this group, we use the traditional busy-waiting method (Lines 9-10).

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 for 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 14). 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 15), then the algorithm accumulates its value (Line 16) and moves to the next element in the same row (Lines 17-18). The second condition is whether the current element is the last one in the row (Line 19). The *col* variable is the column number of the element. If *col* is equal to the last one of the row, then, the algorithm will calculate and save the component's related value (Lines 20-22), and set the array *get_value* to *true* (Line 23) to tell the other threads that the component is solved.

Algorithm 4 Two-Phase CapelliniSpTRSV

```
1: MALLOC (*get_value, m)
2: MEMSET (*get_value, 0)
3: for i = 0 to m − 1 in parallel do        ▷ One thread for one component
4:     warp_begin ← (i/WARP_SIZE)×WARP_SIZE
5:     left_sum ← 0
6:     for j = csrRowPtr[i] to csrRowPtr[i+1]-1 do            ▷ Phase 1
7:         col ← csrColIdx[j]
8:         if col < warp_begin then
9:             while get_value[col] ≠ true do
10:                // busywait
11:            left_sum ← left_sum+csrVal[j]×x[col]
12:        else
13:            break
14:    for k = 0 to WARP_SIZE-1 do                            ▷ Phase 2
15:        while get_vlaue[col] = true do
16:            left_sum ← left_sum + csrVal[j]×x[col]
17:            j ← j + 1
18:            col ← csrColIdx[j]
19:        if col = i then
20:            xi ←(b[i]-left_sum)/csrVal[csrRowPtr[i+1]-1]
21:            x[i] ← xi
22:            __threadfence()
23:            get_value[i] ← true
24:            j ← j + 1
25:            break
26: FREE (*get_value)
```

4.3 Optimization on Control Flow

In this part, we optimize the control flow of Algorithm 4, Two-Phase CapelliniSpTRSV, introduced in Section 4.2.

Limitation in Algorithm 4. For the first phase, The *while*-loop (Line 9) has a runtime issue due to the busy waiting for the threads in the warp: before the computation in Line 11, 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 warp execution manner in GPUs. For example, in Figure 2, $thread6$ waits until the fourth cycle to process $L(5, 2)$; however, due to the *while*-loop (Line 9), 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 14), 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 for-loop in Line 14. For example, in Figure 2, $thread5$ 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 $L(5, 2)$.

Overview. To solve the above performance limitation, we design a Writing-First CapelliniSpTRSV, 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 phases) to the whole elements in the row.

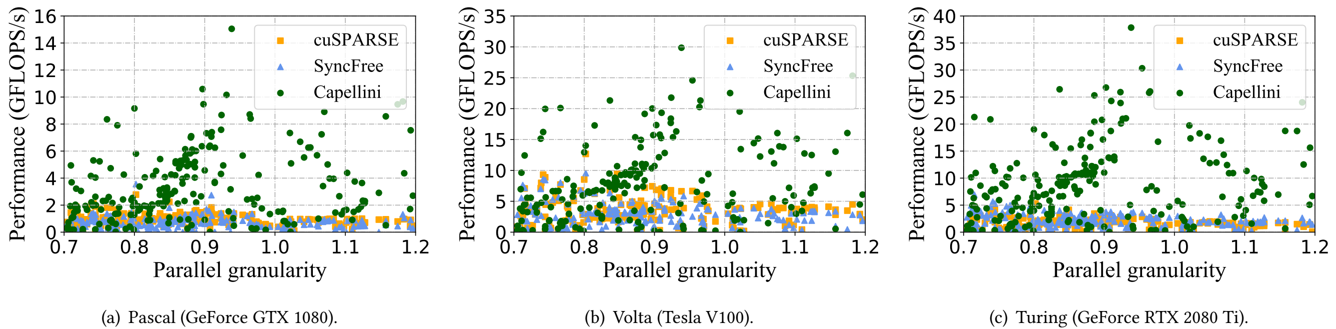
Detailed algorithm. We show our Writing-First CapelliniSpTRSV in Algorithm 5. In this algorithm, each thread computes a component, which relates to a row, in the original row order of the matrix (Line 3). The variable j is equal to the location of the current computing element in the CSR-format matrix (Line 5), and the variable *col* is equal to the column number of the current element (Line 7). There are two conditions to check. The first one is about whether the current computing element is solved. If it is $true$ (Line 8), then the algorithm accumulates its value (Line 9) and moves to the next element in the same row (Lines 10-11). 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 12), then, the algorithm shall calculate and save the related values of the component (Lines 13-15), and set the related value in the array *get_value* to $true$ (Line 16) to tell the other threads that the component is ready.

Algorithm 5 Writing-First CapelliniSpTRSV

```
1: MALLOC (*get_value, m)
2: MEMSET (*get_value, 0)
3: for i = 0 to m − 1 in parallel do        ▷ One thread for one component
4:     left_sum ← 0
5:     j ← csrRowPtr[i]
6:     while j < csrRowPtr[i+1] do
7:         col ← csrColIdx[j]
8:         while get_value[col] = true do
9:             left_sum ← left_sum + csrVal[j]×x[col]
10:            j ← j + 1
11:            col ← csrColIdx[j]
12:        if i = col then
13:            xi ←(b[i]-left_sum)/csrVal[csrRowPtr[i+1]-1]
14:            x[i] ← xi
15:            __threadfence()
16:            get_value[i] ← true
17:            j ← j + 1
18:            break
19: FREE (*get_value)
```

4.4 Discussions

Currently, CapelliniSpTRSV only considers thread-level algorithm designs. A common question could be whether the warp-level and thread-level synchronization-free SpTRSV algorithms can be combined together. The answer is *yes*, and it needs a preprocessing step to analyze the number of nonzero elements in each row, since we need to decide whether a row should be processed at warp-level or thread-level based on the number of nonzero elements in the row. For further optimization, we can decide the processing granularity, warp-level or thread-level, for a set of consecutive rows. Moreover, we can define a threshold: if the average number of nonzero elements is lower than the threshold, we use the thread-level SpTRSV



(a) Pascal (GeForce GTX 1080). (b) Volta (Tesla V100). (c) Turing (GeForce RTX 2080 Ti).

Figure 4: Performance for different SpTRSVs.

(CapelliniSpTRSV) to process the set of rows; otherwise, we use the warp-level synchronization-free SpTRSV. Since this work focuses on SpTRSV without preprocessing, we leave the fusion optimization in our future work.

5 EVALUATION

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

5.1 Experimental Setup

Methods. Our SpTRSV algorithm is denoted as "Capellini". We compare our SpTRSV with the state-of-the-art synchronization-free SpTRSV algorithm [21], which is denoted as "SyncFree". Because for SpTRSV, precision is very important [20, 21, 46], we mainly focus on the double precision. We do not further analyze level-set based methods due to their excessive preprocessing time. Moreover, because cuSPARSE [28] is very popular and has been widely used in various areas, we also compare our algorithm with the SpTRSV in cuSPARSE.

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

Table 3: Platform configuration.

Platform	Pascal	Volta	Turing
GPU	GTX 1080	V100	RTX 2080 Ti
Memory Type	GDDR5X	HBM2	GDDR6
CPU	i7-7700K	E5-2640	i9-9900K
OS	Ubuntu 16.04.4	Ubuntu 16.04.1	Ubuntu 18.04.4
NVCC	8	9	10.2

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 [7], which have been widely used in previous research [20, 21]. 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.

5.2 Performance

GFLOPS. As Figure 3 in Section 3.2, the performance of SyncFree SpTRSV decreases after the parallel granularity is larger than 0.7. Therefore, CapelliniSpTRSV mainly focuses on the sparse matrices with parallel granularity larger than 0.7, which include 245 matrices. These matrices come from various domains: 42.0% from graph applications, 13.9% from circuit simulations, 11.0% from combinatorial problems, 9.4% from linear programming problems, and 8.6% from optimization problems. Experiments show that on all platforms, CapelliniSpTRSV exhibits the highest performance. We show the average performance for different algorithms on the three platforms in Table 4. On average, CapelliniSpTRSV achieves a performance of 6.84 GFLOPS/s, while the SyncFree SpTRSV achieves only 1.78 GFLOPS/s in such matrices, which implies that CapelliniSpTRSV successfully handles the matrices that previous work cannot handle in an efficient manner. The SpTRSV in cuSPARSE can also achieve a performance of 1.92 GFLOPS/s. CapelliniSpTRSV achieves the highest performance for 87% of the matrices. We show the performance results for different algorithms on various GPU platforms when the parallel granularity ranges from 0.7 to 1.2 in Figure 4, which shows that CapelliniSpTRSV brings significant performance benefits.

Table 4: The GFLOPS of different SpTRSV algorithms and the percentage of matrices that achieve the optimal performance using CapelliniSpTRSV.

Platform	Pascal	Volta	Turing	Average
SyncFree	0.65	2.72	1.98	1.78
cuSPARSE	0.90	3.24	1.63	1.92
CapelliniSpTRSV	**3.41**	**8.09**	**9.03**	**6.84**
Percentage	89.39%	81.43%	91.02%	87.28%

Speedup. To further elaborate the benefits of CapelliniSpTRSV over the other SpTRSVs when the parallel granularity is large, we show the performance speedup of CapelliniSpTRSV over the SyncFree and cuSPARSE algorithms in Table 5. On average, CapelliniSpTRSV achieves 4.97x speedup over the SyncFree SpTRSV, and 4.74x speedup over the cuSPARSE SpTRSV for these matrices. We show the performance speedup of CapelliniSpTRSV over SyncFree SpTRSV in Figure 5, and we can see that the performance benefits increase along with the parallel granularity. Specifically, at the parallel granularity of 1.18, for the matrix $lp1$, CapelliniSpTRSV reaches an average 34.77x performance speedup. For the rest of the paper, we analyze CapelliniSpTRSV on the Pascal platform; the analysis results of other platforms are similar.

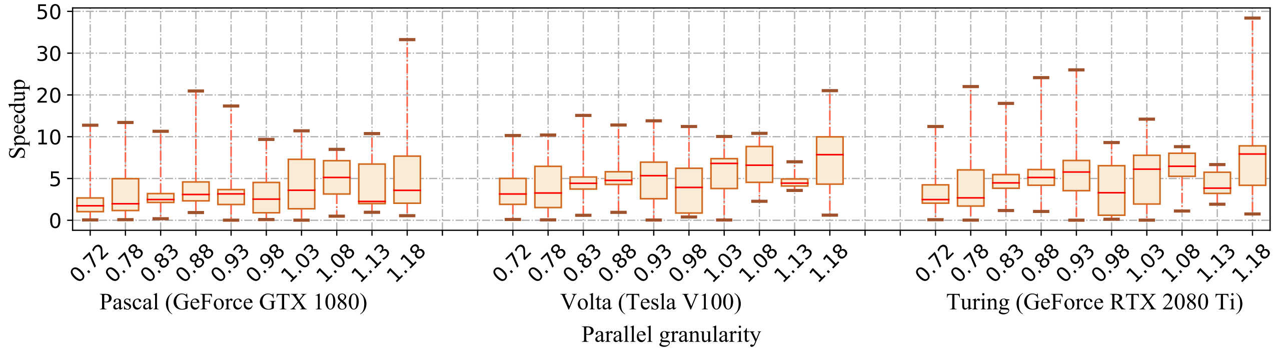


Figure 5: Performance speedup over the SyncFree SpTRSV for different sparse matrices.

Table 5: The average and maximum speedups over SyncFree and cuSPARSE on different platforms.

Platform	Pascal	Volta	Turing
Average speedup over SyncFree	5.26	4.08	5.56
Maximum speedup over SyncFree	21.02	36.48	46.8
Matrix name	*lp1*	*lp1*	*lp1*
Average speedup over cuSPARSE	4.00	3.13	7.09
Maximum speedup over cuSPARSE	23.46	29.83	107
Matrix name	*neos*	*atmosmodd*	*bayer01*

Algorithm preference distribution. Because 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 CapelliniSpTRSV and SyncFree under different factors of n_{level} and nnz_{row} in Figure 6. CapelliniSpTRSV is the best choice when n_{level} is high and nnz_{row} is low.

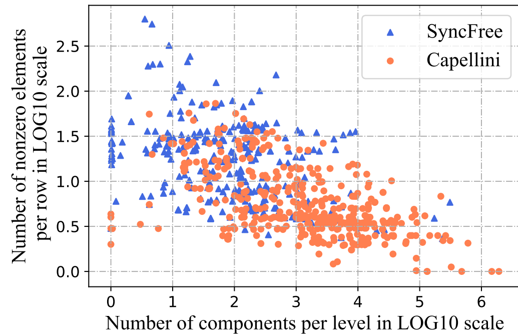


Figure 6: Optimal algorithm distribution.

5.3 Detailed Analysis

To further analyze the benefits of CapelliniSpTRSV, we perform a detailed performance analysis in this part.

Bandwidth. Figure 7 shows the bandwidth utilization. We use the Nvidia performance analysis tool, *nvprof*, to obtain the DRAM read and write bandwidth. CapelliniSpTRSV achieves an average bandwidth of 56.09 GB/s for the matrices whose parallel granularities are larger than 0.7. The bandwidth utilization of CapelliniSpTRSV is 5.17x higher than that of the SyncFree SpTRSV and 5.25x higher than that of the cuSPARSE SpTRSV, which proves the effectiveness of CapelliniSpTRSV.

GPU instructions. CapelliniSpTRSV launches fewer warps than the previous SyncFree SpTRSV, and our algorithm is also more concise. We measure the number of GPU instructions executed and the percentage of instruction stalls to exhibit the instruction executions of different algorithms. Figure 8 (a) shows the number of executed instructions; in general, CapelliniSpTRSV saves 76.02% instructions compared to the SyncFree SpTRSV, and 56.02% instructions compared to the cuSPARSE SpTRSV. Figure 8 (b) shows the instruction stall percentage. The value of our CapelliniSpTRSV is 12.55%, which is 25.60% lower than that of SyncFree SpTRSV and 65.40% lower than that of cuSPARSE SpTRSV.

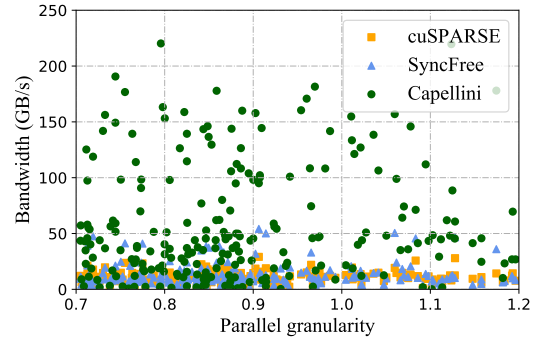
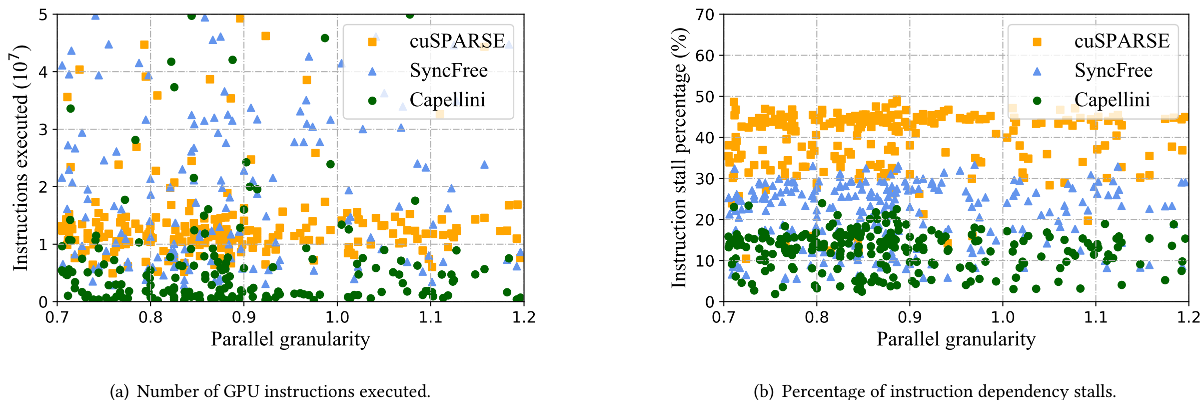


Figure 7: Bandwidth utilization (sum of read and write bandwidth).

Case study. We randomly select three matrices, and show the detailed parameters of different SpTRSVs for these matrices in Table 6. The matrices with high parallel granularities 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 CapelliniSpTRSV are also better.

Optimization analysis. The performance of our Writing-First CapelliniSpTRSV is 28.9x over that of Two-Phase CapelliniSpTRSV. To analyze the optimization benefits of Algorithm 5 of Writing-First strategy over Algorithm 4 of Two-Phase strategy, we compare their bandwidth and instructions. Experiments show that our optimization improves 4.57x bandwidth utilization, and reduces 56.16% GPU instructions, which implies that our optimized CapelliniSpTRSV, Algorithm 5, can better utilize the GPU computing resource and bandwidth than the basic SpTRSV of Algorithm 4.



(a) Number of GPU instructions executed.

(b) Percentage of instruction dependency stalls.

Figure 8: Instruction analysis.

Table 6: Detailed performance indicators for three matrices. δ: parallel granularity. α: average number of nonzero elements per row. β: average number of components per level.

Algorithm	Performance (GFLOPS/s)	Bandwidth (GB/s)	Instructions (10^7)	Stall (%)
rajat29 (δ: 0.78; α: 4.89; β: 14636.23)				
cuSPARSE	0.77	7.23	0.61	42.80
SyncFree	1.67	7.41	0.70	29.06
Capellini	**7.91**	**17.75**	**0.06**	**15.65**
bayer01 (δ: 0.87; α: 3.39; β: 9622.50)				
cuSPARSE	0.65	12.31	1.17	33.50
SyncFree	0.90	10.25	3.20	24.54
Capellini	**3.95**	**48.16**	**0.80**	**14.55**
circuit5M_dc (δ: 0.92; α: 3.02; β: 12812.06)				
cuSPARSE	1.07	8.72	0.87	44.81
SyncFree	1.08	9.22	1.49	29.06
Capellini	**8.67**	**56.15**	**0.10**	**9.50**

6 RELATED WORK

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

Level-set SpTRSV. Anderson and others [1] and Saltz and others [35] proposed that level-set methods could reveal the parallelism in sparse triangular solves However, the synchronization barrier often limits the performance of parallel SpTRSV [16]. To address this problem, Maumov and others [27] implemented a GPU-based SpTRSV with a tradeoff to reduce the number of synchronizations. Further, Park and others [30] proposed a synchronization sparsification technique that significantly reduces the overhead of synchronization and improves its scalability.

Color-set and other SpTRSVs. Schreiber and Tang first used graph coloring to construct color-sets for SpTRSV on multiprocessors [36]. And Suchoski and others [43] extended the method to GPUs. Besides, Anzt and others [2] applied an iterative approach for an approximate SpTRSV solution using GPUs.

Synchronization-free SpTRSV. Liu and others replaced the synchronization with atomic operations [20, 25] and developed a strategy for further parallelizing multiple right-hand sides [21] 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 CapelliniSpTRSV, a thread-level SpTRSV targeting the sparse matrices with high parallel granularity, which can handle the limitation of the previous work.

Matrix optimization. In addition to the algorithms, researchers also proposed other strategies to accelerate the matrix computing, such as the storage format of the matrix and the access speed to the memory. Many applications are implemented based on matrices, such as linear algebra and graph kernels [3, 5, 38, 39, 50, 52]. Kulkarni and others [14] designed the Galois system, an object-based optimistic parallelization system 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 may be exploited by compile-time, inspector-executor or optimistic parallelization [32]. Zhang and others [49] removed dynamic irregularities through data reordering and job swapping to improve the performance on GPUs. Similarly, Wu and others [47] developed two new data reorganization algorithms to eliminate non-coalesced memory accesses that are caused by irregular references. Picciau and others [31] recently proposed a method that partitions the graphical form of an input matrix into multiple sub-graphs to obtain better data access locality and higher concurrency. Rodríguez and others [33] partitioned the irregular computation of sparse matrices into a union of regular (polyhedral) pieces which can then be optimized by o-the-shelf polyhedral compilers. Besides, the memory management and scheduling schemes on GPU is constantly improved [10, 13, 15, 17, 18, 29, 40–42, 44, 48, 51], which could reduce the synchronization time.

7 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 CapelliniSpTRSV that efficiently supports the sparse matrices with high parallel granularities, which cannot be handled efficiently by previous algorithms. CapelliniSpTRSV can be applied to a wide range of HPC

applications, such as iterative solver and direct solver. Experiments show that CapelliniSpTRSV achieves 4.97x performance speedup over the state-of-the-art synchronization-free SpTRSV and 4.74x speedup over the SpTRSV in NVIDIA cuSPARSE. Moreover, our proposed CapelliniSpTRSV is based on the most popular CSR format and does not require preprocessing.

ACKNOWLEDGMENTS

This work is supported by the Beijing Natural Science Foundation (L192027), Science Challenge Project (No. TZZT2016002), the National Natural Science Foundation of China (No. 61972415, 61802412, 61732014), and the Science Foundation of China University of Petroleum, Beijing (No. 2462019YJRC004, 2462020XKJS03). Bingsheng's research is in part supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore. Feng Zhang is the corresponding author of this paper.

REFERENCES

[1] Edward Anderson and Youcef Saad. 1989. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing* (1989).
[2] Hartwig Anzt, Edmond Chow, and Jack Dongarra. 2015. Iterative sparse triangular solves for preconditioning. In *European Conference on Parallel Processing*.
[3] David A Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. 2007. Approximating betweenness centrality. In *International Workshop on Algorithms and Models for the Web-Graph*.
[4] Åke Björck. 1996. *Numerical methods for least squares problems.*
[5] Aydin Buluç and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. In *SC*.
[6] Mayank Daga and Joseph L Greathouse. 2015. Structural agnostic SpMV: Adapting CSR-adaptive for irregular matrices. In *HiPC*.
[7] Timothy A Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* (2011).
[8] Iain S Duff, Albert Maurice Erisman, and John Ker Reid. 2017. *Direct methods for sparse matrices*.
[9] Ernesto Dufrechou and Pablo Ezzatti. 2018. Solving sparse triangular linear systems in modern GPUs: a synchronization-free algorithm. In *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing*.
[10] Zhibin Fang, Xian-He Sun, Yong Chen, and Surendra Byna. 2009. Core-aware memory access scheduling schemes. In *IPDPS*.
[11] Joseph L Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *SC*.
[12] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. 2007. Efficient Gather and Scatter Operations on Graphics Processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*.
[13] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. 2011. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *MICRO*.
[14] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. 2007. Optimistic parallelism requires abstractions. In *PLDI*.
[15] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-driven dynamic GPU cache bypassing. In *ICS*.
[16] Ruipeng Li and Yousef Saad. 2013. GPU-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing* (2013).
[17] Xinyu Li, Lei Liu, Shengjie Yang, Lu Peng, and Jiefan Qiu. 2019. Thinking about A New Mechanism for Huge Page Management. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*.
[18] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. 2019. Hierarchical hybrid memory management in os for tiered memory systems. *TPDS* (2019).
[19] Weifeng Liu. 2015. *Parallel and scalable sparse basic linear algebra subprograms.* Ph.D. Dissertation.
[20] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S Duff, and Brian Vinter. 2016. A synchronization-free algorithm for parallel sparse triangular solves. In *European Conference on Parallel Processing*.
[21] Weifeng Liu, Ang Li, Jonathan D Hogg, Iain S Duff, and Brian Vinter. 2017. Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides. *Concurrency and Computation: Practice and Experience* (2017).
[22] Weifeng Liu and Brian Vinter. 2015. A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors. *JPDC* (2015).
[23] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *ICS*.
[24] Weifeng Liu and Brian Vinter. 2015. Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Comput.* (2015).
[25] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. 2020. Efficient Block Algorithms for Parallel Sparse Triangular Solve. In *ICPP*.
[26] Kiran Matam, Siva Rama Krishna Bharadwaj Indarapu, and Kishore Kothapalli. 2012. Sparse matrix-matrix multiplication on modern architectures. In *19th International Conference on High Performance Computing*.
[27] Maxim Naumov. 2011. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. *NVIDIA Corp., Westford, MA, USA, Tech. Rep.* (2011).
[28] M Naumov, LS Chien, P Vandermersch, and U Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.
[29] Xiang Pan and Radu Teodorescu. 2014. Using STT-RAM to enable energy-efficient near-threshold chip multiprocessors. In *PACT*.
[30] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *International Supercomputing Conference*.
[31] Andrea Picciau, Gordon E Inggs, John Wickerson, Eric C Kerrigan, and George A Constantinides. 2016. Balancing locality and concurrency: solving sparse triangular systems on GPUs. In *HiPC*.
[32] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, et al. 2011. The tao of parallelism in algorithms. In *PLDI*.
[33] Gabriel Rodríguez and Louis-Noël Pouchet. 2018. Polyhedral modeling of immutable sparse matrices. In *8th International Workshop on Polyhedral Compilation Techniques. Manchester, UK*.
[34] Yousef Saad. 2003. *Iterative methods for sparse linear systems.*
[35] Joel H Saltz. 1990. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM journal on scientific and statistical computing* (1990).
[36] Robert Schreiber and Wei-Pei Tang. 1982. Vectorizing the conjugate gradient method. *Unpublished manuscript, Department of Computer Science, Stanford University* (1982).
[37] Naser Sedaghati, Te Mu, Louis-Noel Pouchet, Srinivasan Parthasarathy, and P Sadayappan. 2015. Automatic selection of sparse matrix representation on GPUs. In *ICS*.
[38] Yogesh Simmhan, Neel Choudhury, Charith Wickramaarachchi, Alok Kumbhare, Marc Frincu, Cauligi Raghavendra, and Viktor Prasanna. 2015. Distributed programming over time-series graphs. In *IPDPS*.
[39] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. Goffish: A sub-graph centric framework for large-scale graph analytics. In *Euro-Par*.
[40] Clinton W Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R Stan. 2011. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *HPCA*.
[41] Fengguang Song, Shirley Moore, and Jack Dongarra. 2007. Feedback-directed thread scheduling with memory considerations. In *HPDC*.
[42] Fengguang Song, Shirley Moore, and Jack Dongarra. 2007. L2 cache modeling for scientific applications on chip multi-processors. In *ICPP*.
[43] Brad Suchoski, Caleb Severn, Manu Shantharam, and Padma Raghavan. 2012. Adapting sparse triangular solution to GPUs. In *ICPP Workshop*.
[44] Bin Wang, Weikuan Yu, Xian-He Sun, and Xinning Wang. 2015. Dacache: Memory divergence-aware GPU cache management. In *ICS*.
[45] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel transposition of sparse data structures. In *ICS*.
[46] Xinliang Wang, Weifeng Liu, Wei Xue, and Li Wu. 2018. swSpTRSV: a fast sparse triangular solve with sparse level tile layout on sunway architectures. In *ACM SIGPLAN Notices*.
[47] Bo Wu, Zhijia Zhao, Eddy Zheng Zhang, Yunlian Jiang, and Xipeng Shen. 2013. Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU. *PPoPP* (2013).
[48] Chenhao Xie, Fu Xin, Mingsong Chen, and Shuaiwen Leon Song. 2019. OO-VR: NUMA friendly object-oriented VR rendering framework for future NUMA-based multi-GPU systems. In *ISCA*.
[49] Eddy Z Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. *ASPLOS* (2011).
[50] Feng Zhang, Weifeng Liu, Ningxuan Feng, Jidong Zhai, and Xiaoyong Du. 2019. Performance evaluation and analysis of sparse matrix and graph kernels on heterogeneous processors. *CCF Trans. HPC* (2019).
[51] Wei Zhang, Sudhanva Gurumurthi, Mahmut T Kandemir, and Anand Sivasubramaniam. 2003. ICR: In-Cache Replication for Enhancing Data Cache Reliability. In *DSN*.
[52] Li Zhou, Ren Chen, Yinglong Xia, and Radu Teodorescu. 2018. C-graph: A highly efficient concurrent graph reachability query framework. In *ICPP*.