AG-SpTRSV: An Automatic Framework to Optimize Sparse Triangular Solve on GPUs

ZHENGDING HU, Computer Science and Technology, University of Science and Technology of China, Hefei, China
JINGWEI SUN, Computer Science and Technology, University of Science and Technology of China, Hefei, China
ZHONGYANG LI, Computer Science and Technology, University of Science and Technology of China, Hefei, China
GUANGZHONG SUN, Computer Science and Technology, University of Science and Technology of China, Hefei, China

Sparse Triangular Solve (SpTRSV) has long been an essential kernel in the field of scientific computing. Due to its low computational intensity and internal data dependencies, SpTRSV is hard to implement and optimize on graphics processing units (GPUs). Based on our experimental observations, existing implementations on GPUs fail to achieve the optimal performance due to their suboptimal parallelism setups and code implementations plus lack of consideration of the irregular data distribution. Moreover, their algorithm design lacks the adaptability to different input matrices, which may involve substantial manual efforts of algorithm redesigning and parameter tuning for performance consistency. In this work, we propose AG-SpTRSV, an automatic framework to optimize SpTRSV on GPUs, which provides high performance on various matrices while eliminating the costs of manual design. AG-SpTRSV abstracts the procedures of optimizing an SpTRSV kernel as a *scheme* and constructs a comprehensive optimization space based on it. By defining a unified code template and preparing code variants, AG-SpTRSV enables fine-grained dynamic parallelism and adaptive code optimizations to handle various tasks. Through computation graph transformation and multi-hierarchy heuristic scheduling, AG-SpTRSV generates schemes for task partitioning and mapping, which effectively address the issues of irregular data distribution and internal data dependencies. AG-SpTRSV searches for the best scheme to optimize the target kernel for the specific matrix. A learned lightweight performance model is also introduced to reduce search costs and provide an efficient end-to-end solution. Experimental results with SuiteSparse Matrix Collection on NVIDIA Tesla A100 and RTX 3080 Ti show that AG-SpTRSV outperforms state-of-the-art implementations with geometric average speedups of 2.12x ~ 3.99x. With the performance model enabled, AG-SpTRSV can provide an efficient end-to-end solution, with preprocessing times ranging from 3.4 to 245 times of the execution time.

This is a new article, not an extension of a conference paper.
This study is supported by the Strategic Priority Research Program of Chinese Academy of Sciences, Grant No.XDB0500102.
Authors' Contact Information: Zhengding Hu, Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China; e-mail: huzd@mail.ustc.edu.cn; Jingwei Sun (Co-corresponding author), Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China; e-mail: sunjw@ustc.edu.cn; Zhongyang Li, Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China; e-mail: lzy123lzy123@mail.ustc.edu.cn; Guangzhong Sun (Co-corresponding author), Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China; e-mail: gzsun@ustc.edu.cn.
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 the author(s) 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.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1544-3973/2024/11-ART70
https://doi.org/10.1145/3674911

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; **Linear algebra algorithms**; • **Mathematics of computing** → **Solvers;**

Additional Key Words and Phrases: Sparse matrix, triangular solve, automatic optimization, GPU

ACM Reference Format:
Zhengding Hu, Jingwei Sun, Zhongyang Li, and Guangzhong Sun. 2024. AG-SpTRSV: An Automatic Framework to Optimize Sparse Triangular Solve on GPUs. *ACM Trans. Arch. Code Optim.* 21, 4, Article 70 (November 2024), 25 pages. https://doi.org/10.1145/3674911

1 Introduction

The **sparse triangular solve (SpTRSV)** is one of the most important kernels in the field of sparse matrix computation and plays an indispensable role in many numerical algorithms, e.g., the preconditioners of sparse iterative solvers [26]. It is widely used in various scientific applications, such as **computational fluid dynamics (CFD)** [22], machine learning [18], and graph algorithms [11]. SpTRSV solves a linear system $Ax = b$, where A is a sparse lower/upper triangular matrix, b is the right-hand side vector, and x is the solution. Compared with other well-known linear algebra routines, such as **sparse matrix-vector multiplication (SpMV)** [12, 15, 17], **sparse matrix-matrix multiplication (SpMM)** [16, 21], and sparse matrix transposition [33], SpTRSV faces more difficulties in parallelization due to its internal data dependencies.

Graphics processing units (GPUs) are capable of processing floating point operations at extreme rates and have become one of the most widely used accelerators in the field of scientific computing. However, it is challenging to implement an efficient SpTRSV kernel on GPUs. Difficulties can be summarized as follows:

— *Low parallelism and high synchronization overhead.* SpTRSV has a sequential nature due to dependencies, making it difficult to exploit the parallelism. Synchronizations are involved with such data dependencies. Due to the complex synchronization mechanisms of GPUs, a meticulous design is required.
— *Complex task partitioning and scheduling.* The irregular and sparse data distribution significantly impacts performance, which is similar to many other sparse matrix kernels. Any task partitioning or scheduling potentially changes the data dependencies in the solution. Thus, determining the appropriate strategies is extremely challenging.
— *Input sensitivity.* The sparse patterns of input matrices vary significantly across different application scenarios. There is no "one-for-all"solution, which requires heavy and tedious manual efforts of algorithm redesigning and parameter fine-tuning.

To exploit the parallelism of SpTRSV, the level-set method [3, 28, 32] (also known as *wavefront parallelization*) proposes to partition the solution into multiple levels with no internal dependencies. Synchronization is only involved for inter-level dependencies. However, it cannot be directly applied on GPUs as the global synchronization involves excessive overhead. Considering the hardware characteristics of GPUs, the "synchronization-free" methods [8, 14] use fine-grained point-to-point communication to resolve data dependencies, specifically with in-memory data exchange and atomic operations. Simultaneously, different levels of parallelism can be exploited, including warp-level [8, 14], thread-level [30], and adaptive heuristics [38]. Furthermore, the parallel libraries by vendors such as cuSPARSE [23] provide well-optimized implementations of SpTRSV on GPUs.

Despite the fact that existing implementations perform well on certain matrices, they fail to consistently achieve good performance on matrices with various sparsity patterns due to the dedicated algorithm design and the lack of adaptability. Here, we test and compare the performance of the state-of-the-art implementations on GPUs, including the Compressed Sparse Column (CSC)–based synchronization-free method [14] (Sync-free for short), YuenyeungSpTRSV [38]

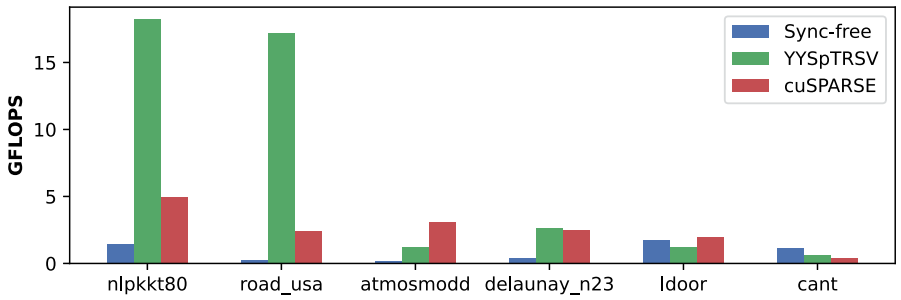


Fig. 1. Performance comparison across Sync-free, YYSpTRSV and cuSPARSE with typical matrices in scientific applications. The experiments are conducted on NVIDIA RTX 3080Ti.

(YYSpTRSV for short), and SpTRSV solve in the NVIDIA cuSPARSE library [23] (cuSPARSE for short). Test matrices are derived from applications in various fields (detailed information is listed in Table 5). As is shown in Figure 1, the three implementations exhibit performance advantages on matrices with different sparsity patterns. For example, YYSpTRSV performs well on matrices with high parallelism, whereas Sync-free is more suitable for matrices with low parallelism. However, none of them can consistently achieve high performance across all the test cases. Moreover, significant performance gaps can be observed in a single test case. To fully optimize the performance of SpTRSV on GPUs, we argue that a more comprehensive design is expected, which addresses the performance bottlenecks introduced by irregular data distribution and data dependencies while adaptively dealing with diverse sparsity patterns of different matrices.

In this article, we first measure and characterize the performance of SpTRSV. We derive several observations that provide guidance for the design of the optimization space. Based on these observations, we propose AG-SpTRSV, an automatic framework to optimize SpTRSV on GPUs. AG-SpTRSV consists of four stages. In the *Prepare* stage, AG-SpTRSV prepares a series of code variants based on a unified template that support dynamic fine-grained parallelism and enable code optimizations under specific conditions. In the *Transform* stage, the original computation graph is transformed into candidate graphs with merging and reordering operations. In the *Schedule* stage, the tasks in candidate graphs are mapped to the hardware through multi-hierarchy heuristic strategies. We refer to the entire process of code variant preparation, graph transformation, and scheduling as a *scheme*. In the *Select* stage, AG-SpTRSV finds the scheme with the best expected performance, with either exhaustive search or a learned lightweight model. AG-SpTRSV outperforms state-of-the-art SpTRSV implementations on GPUs and achieves good performance across matrices with various sparsity paterns.

The contributions of this article can be summarized as follows.

— We characterize the performance of GPU-based SpTRSV through experimental measurements and derive several observations that help with performance optimization.
— We represent the optimization space of SpTRSV as the *scheme*, which considers dynamic parallelism, adaptive code optimization, computation graph transformation, and scheduling. We design a series of strategies to construct a comprehensive space that accommodates inputs with various sparsity patterns.
— We propose AG-SpTRSV, an automatic framework to optimize GPU-based SpTRSV, which generates a series of schemes for execution and searches for the best. We also adopt a lightweight model based on historical results to reduce search costs.
— Experimental results on NVIDIA Tesla A100 and RTX 3080Ti show that AG-SpTRSV is able to outperform the state-of-the-art SpTRSV implementations, including Sync-free,

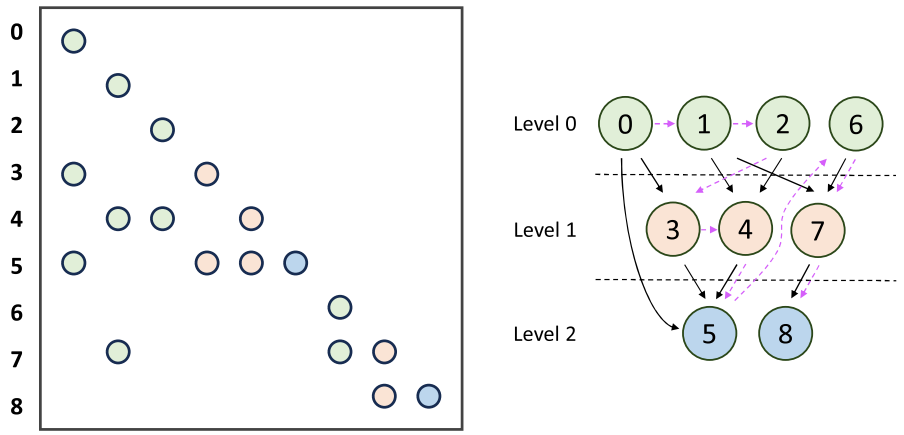


Fig. 2. A sparse lower triangular matrix and its computation graph.

YYSpTRSV and cuSPARSE, with geometric average speedups of 2.12x ∼ 3.99x. With the proposed performance model, the preprocessing time of AG-SpTRSV ranges from 3.4 to 245 times of the execution time.

— The source code of AG-SpTRSV is available at https://github.com/USTC-ADA/AG-SpTRSV.git.

2 Background

2.1 SpTRSV and Level-Set Method

The SpTRSV solves a linear system $Ax = b$, where A is the lower triangular matrix of size $m \times m$, b is the dense **right-hand-side (rhs)** vector, and x is the dense solution vector. Without loss of generality, we assume that the matrix is stored in **Compressed Sparse Row (CSR)** format in our illustration. The CSR format stores the sparse matrix by three arrays, where $ColIdx$ and Val store the column index and the value of each non-zero, respectively, and $RowPtr$ stores the start point of each row in the above two arrays. An example of a sparse lower triangular matrix is shown on the left-hand side of Figure 2. In this example, $RowPtr = \{0, 1, 2, 3, 5, 8, 12, 13, 16\}$ and $ColIdx = \{0, 1, 2, 0, 3, 1, 2, 4, 0, 3, 4, 5, 6, 1, 6, 7, 4, 7, 8\}$.

The serial SpTRSV algorithm with the CSR format is illustrated in Algorithm 1. Here, we assume that the diagonal elements of each row are present by default. In serial SpTRSV, components of x are calculated in the order of row number. The component $x[j]$ are guaranteed to be finished when calculating $x[i]$, where $0 < j < i < m$, the data dependency is thus satisfied.

However, the sequential computing order of serial SpTRSV poses challenges to efficient parallelization. To exploit the parallelism of the problem, the level-set method [3, 28] has been proposed. The level-set method partitions the solution into several levels. A level is composed of components independent of each other. Dependencies may exist among different levels. The right part of Figure 2 shows the level-set partition as a directed computation graph. Each directed edge in the computation graph represents a non-zero element in A, indicating a data dependency relationship. The level-set method solves independent components inside every single level in parallel while processing levels in a sequential order to ensure dependencies.

The level-set method can efficiently exploit parallelism in SpTRSV. However, it requires synchronization at the end of each level to avoid violation of data dependencies. This may introduce extra costs on GPUs with huge amounts of parallel units, especially when the number of levels is large.

ALGORITHM 1: Serial Triangular Solve with CSR Format

```
1  Function serial_SpTRSV(A, b, x):
2      for row = 0, m do
3          sum ← 0;
4          for idx = A.RowPtr[row], A.RowPtr[row + 1] - 1 do
5              sum ← sum + A.Val[idx] ∗ x[A.ColIdx[idx]];
6          end
7          diag ← A.Val[A.RowPtr[row + 1] − 1];
8          x[row] ← (b[row] − sum)/diag;
9      end
10 End Function
```

2.2 SpTRSV on GPUs

Directly applying level-set methods on GPUs cannot achieve satisfactory performance due to excessive costs for global synchronization. Liu [14] proposed the synchronization-free method using the CSC format. The concept of the synchronization-free method does not imply the complete elimination of synchronization, but rather the replacement of global synchronization in level-set methods with fine-grained point-to-point synchronization. Dufrechou and Ezzatti [8] and Su et al. [30] proposed synchronization-free methods using the CSR format with warp-level and thread-level parallelism, respectively. The two algorithms are designed for matrices with different sparsity patterns. YYSpTRSV [38] dynamically switches between the two algorithms according to the average number of non-zeros per row. Dufrechou and Ezzatti [8] also proposed to process multiple rows with a single warp. The pseudocode of the warp-level and thread-level methods with the CSR format is shown in Algorithm 2, where an auxiliary array named $flag$ is used to mark the completion of each component. In both methods, memory fences (Line 13 & Line 30) are used to strictly ensure that the flags are updated after the results of the components are written back. In the warp-level method, the non-zeros of each row are partitioned and assigned to different threads (Line 5). A warp-level reduce routine is introduced (Line 11) to sum up the results of each thread. Thread-level method assigns every t rows to a warp and each thread corresponds to one or more rows. t is set as the size of the warp in [30]. As all threads in a single warp follow the same control flow in GPU programming, a while loop is used to avoid deadlock. A more detailed description of the two algorithms can be found in [38]. In this work, we introduce a unified and parameterized template for parallelism (discussed in Section 4.2). As a superset of existing methods, the template extends the design space with finer granularity and exhibits adaptability to different tasks in the solution.

3 Motivation

To better characterize the performance of existing SpTRSV implementations, we conduct a series of experiments with a set of commonly used sparse matrices with different sparsity patterns on NVIDIA RTX 3080Ti. Detailed matrix characteristics are listed in Table 5. The interpretation of those characteristics can be found in Table 3. Based on the experimental results, we can derive several observations that help with performance optimization.

3.1 Dilemma of Parallelism Granularity

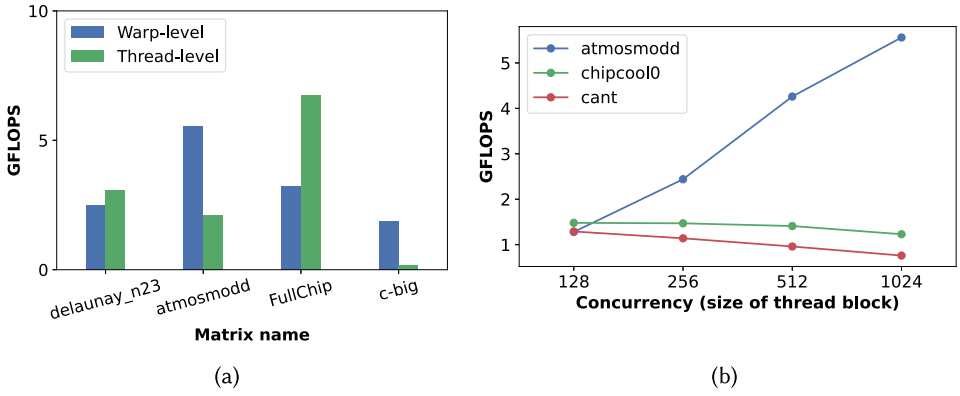
Based on the different parallel levels of GPUs, existing work mainly adopts two parallelism granularities, including warp level [8, 14] and thread level [30]. Many employ the fixed parallelism granularity setup, which cannot achieve good performance across various matrices. YYSpTRSV

ALGORITHM 2: Warp-Level and Thread-Level Triangular Solve on GPUs with Fine-Grained Synchronization

```
1  Function warp_level_SpTRSV(A, b, x, flag):
2      // Processing each row in one warp
3      for row = 0, m in warp-parallel do
4          local_sum ← 0;
5          for idx = A.RowPtr[row], A.RowPtr[row + 1] - 1 in thread-parallel do
6              while flag[A.ColIdx[idx]] == 0 do
7                  Busy wait;
8              end
9              local_sum ← sum + A.Val[idx] ∗ x[A.ColIdx[idx]];
10         end
11         warp_reduce_add(local_sum);
12         x[row] ← (b[row] − sum)/A.Val[A.RowPtr[row + 1] − 1];
13         mem_fence();
14         flag[row] ← 1;
15     end
16 End Function
17 Function thread_level_SpTRSV(A, b, x, flag):
18     // Processing every t rows in one warp
19     for row_block = 0, m step t in warp-parallel do
20         for row = row_block, row_block + t in thread-parallel do
21             idx ← A.RowPtr[row];
22             sum ← 0;
23             while idx < A.RowPtr[row+1] do
24                 if flag[A.ColIdx[idx]] then
25                     sum ← sum + A.Val[idx] ∗ x[A.ColIdx[idx]];
26                     pos ← pos + 1;
27                 end
28                 if pos == A.RowPtr[row+1]-1 then
29                     x[row] ← (b[row] − sum)/A.Val[idx];
30                     mem_fence();
31                     flag[row] ← 1;
32                 end
33             end
34         end
35     end
36 End Function
```

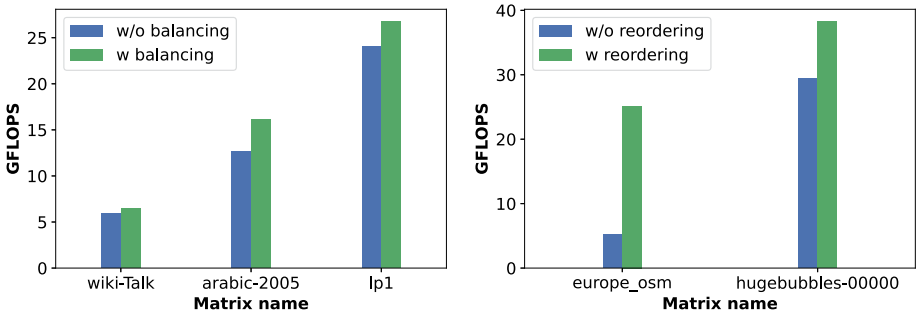
[38] determines the parallelism granularity based on the number of non-zeros per row. However, such approaches can be suboptimal. We evaluate four structured matrices with similar average non-zeros per row as an example. As is shown in Figure 3(a), *atmosmodd* and *c - big* performs better with the warp-level algorithm, whereas *delaunay_n23* and *FullChip* are better suited for the thread-level algorithm.

We also find that some cases cannot be fully optimized with either warp-level or thread-level parallelism. Taking *atmosmodd* as an example, though the warp-level algorithm performs better than the thread-level algorithm, the average non-zero per row of the matrix is only 3.97. This results in nearly 75% of threads in the warp remaining idle, leading to very low utilization.



(a) (b)

Fig. 3. Performance comparison across different parallelism setups.



(a) (b)

Fig. 4. Performance gains brought by handling irregularity.

Moreover, the parallelism of the matrix affects the optimal concurrency. Figure 3(b) shows how the performance varies when different sizes of thread-block are used. *atmosmodd* has very high parallelism in each level (though limited among consecutive rows); thus, the performance improves as the concurrency increases. By contrast, *chipcool*0 and *cant* are band matrices with low parallelism. When the concurrency is high, most threads are idle waiting for data dependencies to be resolved, which leads to unnecessary resource contention and thus brings performance degradation. We argue that a more flexible and comprehensive parallelism setup is required.

3.2 Challenges of Irregularity

Real-world matrices in real-world applications often exhibit irregular distribution of non-zeros. This introduces more challenges to performance optimization due to the mismatch between irregular computation patterns and the GPU hardware design. Our experiments show that the issues of load balancing and data locality brought by irregularity need to be carefully handled.

Figure 4(a) illustrates the impact of load balancing on performance. *wiki-Talk*, *arabic-2005* and *lp1* are matrices with uneven distribution of non-zero elements among the rows, which easily leads to runtime load imbalance. Here, we compare the performance between scheduling with and without the load-balancing strategy (discussed in Section 4.4), which brings speedups of up to 20%.

We also evaluate the impact of data locality by comparing the performance with and without matrix reordering (discussed in Section 4.3). Matrix reordering can efficiently improve data locality

by placing tasks with a similar computational order together. Figure 4(b) shows that reordering can bring significant speedups for matrices with high parallelism.

3.3 Suboptimal Code Implementation

The code implementation of the existing algorithms (shown in Algorithm 2) can be further refined. We observe at least the following three performance bottlenecks through profiling.

- Memory access to sparse matrix and components, especially in the thread-level algorithm, exhibit non-aligned and non-coalesced patterns. This leads to extra memory transactions, which greatly impacts memory access efficiency.
- Fine-grained synchronizations, such as reading/writing flags and performing memory fences, introduce non-negligible costs. In fact, some of them are unnecessary.
- Division operations are performed to calculate the value of the components. On GPUs, division instruction has high latency, which easily impacts the instruction pipeline.

3.4 Non-orthogonal Optimization Space

In the previous observations, we identify several factors that affect the performance of SpTRSV and summarize their intuitive relationships with the sparsity patterns of matrices. We also find that their effects on performance are not independent. For example, when the granularity of the task changes, the optimal parallelism setup and scheduling strategy may vary. According to our experiments, simply combining the respectively optimal choices of each factor cannot achieve a globally optimal performance and, if anything, falls far short of being satisfactory. All those factors collectively form a vast space, which makes manual optimization extremely tedious and challenging.

4 AG-SpTRSV: An Automatic Framework to Optimize SpTRSV on GPUs

4.1 Overview

To fully consider the aforementioned factors and bridge the performance gaps, we propose AG-SpTRSV, an automatic framework to optimize SpTRSV on GPUs. Compared with existing work [8, 14, 30, 38], we consider a much larger optimization space, which carefully handles the observed performance issues and consistently achieves good performance on matrices with various sparse patterns. By automatic scheme searching and selecting, AG-SpTRSV is able to provide an efficient target kernel for a specific matrix, with eliminated costs of manual algorithmic design and parameter tuning.

AG-SpTRSV consists of four stages. In the *Prepare* stage, a series of code variants with different parallelism setups and optimization methods are prepared. These code variants are used for different sparsity patterns. In the *Transform* stage, the sparse matrix is represented as a computation graph. Through graph operations, including node merging and reordering, the original computation graph is transformed into a series of candidate graphs for subsequent optimization and evaluation. In the *Schedule* stage, the nodes of the candidate graphs are mapped to GPU warps and hardware through multi-hierarchy heuristic scheduling. In the *Select* stage, AG-SpTRSV exhaustively tests and evaluates the combination of strategies (referred to as *schemes*). The scheme with the optimal performance is selected for target solving kernel. In this stage, a lightweight learning model is proposed to reduce preprocessing time.

4.2 Code Variant Preparation

As discussed in Section 3, the code implementations in existing work [8, 14, 30, 38] are not consistently optimal in all cases. In the *Prepare* stage, we abstract the implementation of SpTRSV as

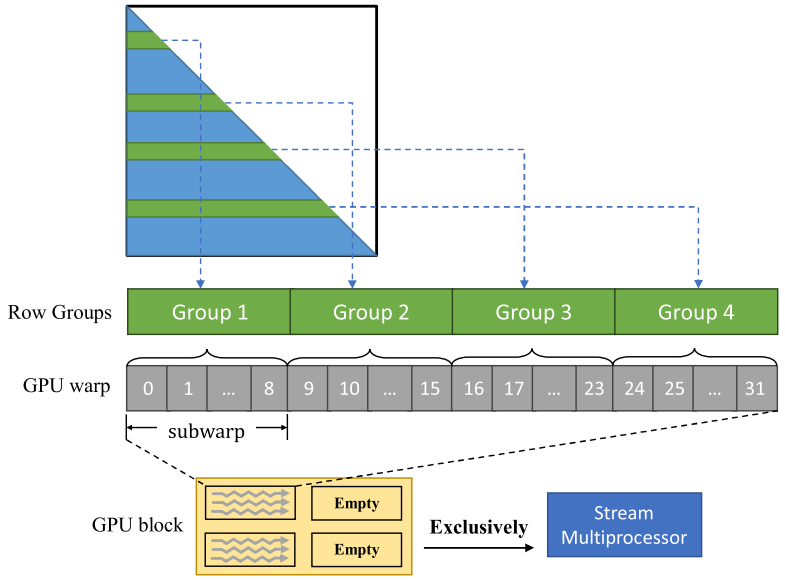


Fig. 5. An example of the fine-grained dynamic parallelism when $ng = 4$ and $nw = 2$. Each warp processes 4 independent row groups and each stream multiprocessor is exclusively assigned a GPU block of 2 executing warps.

a unified code template to better represent the solving behavior. Based on the template, we design and implement a series of solution codes with various parallelism and optimization details, referred to as *Code Variants*, to enhance performance for different sparsity patterns.

4.2.1 Fine-Grained Dynamic Parallelism. Through the experiments in Section 3.1, we have demonstrated that exclusively utilizing warp-level or thread-level parallelism cannot achieve optimal performance. Although YYSpTRSV [38] proposes to switch between the two algorithms, the parallel granularity still faces a dilemma (discussed in Section 3.1). To fill the gap between the previous two algorithms, we adopt fine-grained dynamic parallelism to better utilize the GPU resources.

As is shown in Figure 5, a GPU warp is divided into ng smaller units for parallelism, referred to as *subwarps*. Each subwarp is assigned a row group, which contains several consecutive rows for solution. The ng groups have no inter-group data dependencies so that each subwarp can process an independent task in parallel in a deadlock-free manner. ng is a configurable parameter and remains fixed for each code variant. Generally, $1 \leq ng \leq 32$, and ng is set as a power of 2 to achieve better performance of reduction.

Within each row group, the subwarps can choose to either collaboratively process each row one by one or individually process separate rows. The choice is based on the average number of non-zeros per row. We use a configurable parameter sw_rnnz as the threshold, which is also fixed for each code variant. When the average non-zero number of the rows in a row group is larger than sw_rnnz, the threads in the subwarp collaboratively process one row to enhance memory access efficiency with more coalesced memory access. Otherwise, the threads in the subwarp individually process separate rows to improve warp utilization and eliminate reduction overheads. We set the value of sw_rnnz from $\{0, \infty\} \cup \{2^i, i \in \mathbb{N} \ \& \ 2^i \leq \frac{32}{ng}\}$. The upper bound $\frac{32}{ng}$ indicates that each thread in the subwarp processes at least one non-zero on average.

It is worth noting that when $ng = 1$, the method becomes the warp-level [8, 14] and thread-level [30] implementations. When $ng > 1$ and each row group consists of only one row, the method

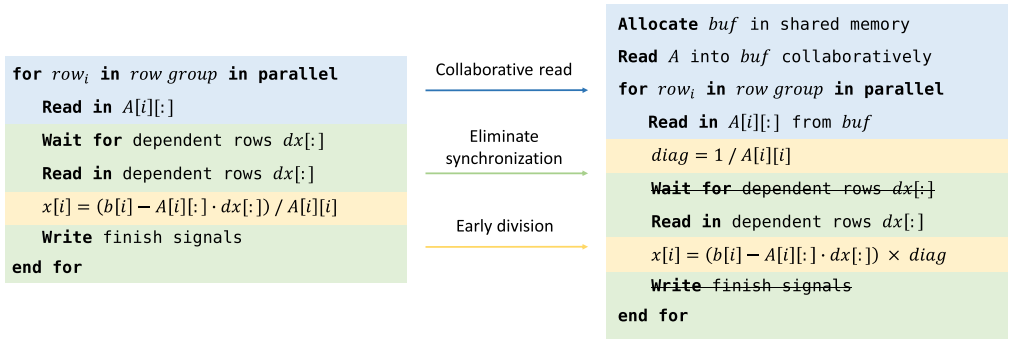


Fig. 6. Applying adaptive optimizations on the code template.

becomes the multiple-row-per-warp implementation mentioned in [8]. Our method follows a more comprehensive design with fine-grained parallelism, which also encompasses existing algorithms.

Section 3.1 also demonstrates that the best performance is not necessarily achieved under the maximum concurrency. Considering that, AG-SpTRSV dynamically adjusts the hardware-level concurrency. Specifically, we set the block size to the maximum possible number (1024 on Ampere). Each thread block contains nw warps that perform the solution, whereas the remaining warps are empty and do nothing. Then, we enforce a thread block to execute exclusively on a stream processor, thus explicitly controlling the number of executing warps.

4.2.2 Adaptive Code Optimization. When processing a specific row (assumed to be the i-th row), the code implementation can be represented as follows. Firstly, data of the i-th row of the sparse matrix is read in, including the start and end row pointers as well as the column indices of the non-zeros. Next, the warp waits for dependent components to finish processing. After finishing, the values of those components are read in. Then, the computation phase begins. The warp calculates and writes back the value of component $x[i]$. Lastly, the necessary synchronization information, such as $flag[i]$ mentioned in Section 2.2, is written back.

We have shown in Section 3.3 that the existing code implementations suffer from non-coalesced memory access, excessive synchronizations, and long-latency division operations. To eliminate these performance bottlenecks, we designed several optimization methods that can be applied under specific circumstances, as is shown in Figure 6.

Collaborative read. Since SpTRSV is a memory-bound kernel, its memory access efficiency has a predominant impact on its performance. In the existing implementations, each thread within a single warp independently reads the required sparse matrix data on demand, which may lead to inefficient non-coalesced memory access. We use collaborative reading to address the issue. Specifically, each GPU warp allocates a buffer (specifically a piece of shared memory on GPUs) to store data of the sparse matrix. All threads in the warp first cooperatively read data of each row group in sequence. This enables coalesced memory access and improves the utilization of memory bandwidth. When reading finishes, each subwarp begins processing its own row group. The optimization method is applied when the sparse matrix data required by the rows does not exceed the maximum available buffer size, and each row group has sufficient non-zeros to enable coarse-grained coalesced memory access.

Elimination of synchronization. Fine-grained synchronizations for data dependencies (including waiting for dependent rows and writing synchronization information) also incur non-negligible

Table 1. Comparison Across Code Variant of AG-SpTRSV and Other GPU SpTRSV Implementations

SpTRSV implementation	Target sparsity pattern	Dynamic parallelism	Adaptive optimization
Warp-level algorithm [8, 14]	Rows with large # of non-zeros	✗	✗
Thread-level algorithm [30]	Blocks of rows with low # of non-zeros and balanced distributions	✗	✗
YYSpTRSV [38]	Blocks of rows suitable to process with warp-level or thread-level algorithm	✓	✗
Multiple-row-per-warp [8]	Distributed rows with the same level and small # of rows	✓	✗
AG-SpTRSV	Various non-zero distributions	✓	✓

overheads. We introduce the following rules to eliminate synchronization. When all the rows solved by the current kernel are located within the same level, we can eliminate the logic of waiting as all the dependent rows have been processed beforehand. When no rows depend on the current row or the waiting of the rows that depend on the current row has already been eliminated, we can eliminate the logic of writing back synchronization information. This optimization method can be applied when rows of different levels are independently scheduled, which will be further discussed in Section 4.4. AG-SpTRSV adaptively adjusts the synchronization behavior based on the sparse pattern of the matrix, which balances between the fine-grained synchronization methods [8, 14, 30, 38] and the level-set methods [3, 28].

Early division. When computing component results, the long latency of division operations potentially impacts performance. Therefore, we adopt the optimization to perform divisions earlier. Before waiting for dependent data, the warp/thread calculates the reciprocal of the diagonal non-zero element of the current row. This allows us to use multiplication with the reciprocal instead of division when calculating the final result. This enables warps/threads that need to wait for dependent data (which cannot start their computation immediately) to perform expensive division operations in advance, thereby enhancing computational parallelism. We find that this modification can bring performance improvement of over 20% on certain matrices.

In Table 1, we compared the code variant of AG-SpTRSV with existing GPU SpTRSV implementations. While all existing implementations focus on some specific sparsity patterns, AG-SpTRSV enables dynamic parallelism and adaptive optimizations to prepare code variants, which can achieve adaptability to matrices with various sparsity patterns.

4.3 Computation Graph Transformation

With fine-grained dynamic parallelism (discussed in Section 4.2.1), a GPU warp is assigned one or multiple row groups to process. How to partition rows of a sparse matrix into row groups crucially impacts performance, as it is not only related to parallelism and task granularity but also affects the data dependencies in the overall solution.

AG-SpTRSV uses computation graphs to represent task partitioning and data dependencies. In a computation graph, a node (noted as N_i) represents a row group, and two types of edges are used to represent the relationships between nodes. A *dependency edge* connects N_i to N_j when there is a

non-zero element at the position (r_1, r_2) of the matrix, where $r_1 \in N_j$ and $r_2 \in N_i$. The dependency edge indicates that the computation of N_i can only begin after the computation of N_j finishes. A *data edge* connects N_i to N_j when the last row in N_i and the first row in N_j are contiguous. The data edge indicates that the two row groups are stored continuously in memory and can be further merged into a larger one. The level of node is defined as the topological order of the node in the computation graph. The computation graph also records the level of each node, as we must assign nodes within the same level to one GPU warp to avoid deadlocks.

Directly parallelizing the original computation graph may lead to unsatisfactory performance, as is shown in Section 3 (as well as existing work [20, 30, 38]). In the *Transform* stage, AG-SpTRSV performs two types of transformation operations on the original computation graph: node merging and reordering. A set of new graphs with different topological structures are generated, which we refer to as *candidate graphs*. Those graphs will be further scheduled and measured to achieve better potential performance.

The merging operation traverses the graph along data edges and merges several nodes into a larger one. AG-SpTRSV leverages the following merging strategies for matrices with different sparsity patterns.

- *Merge by fixed block size.* The $merge\_fixed$ strategy merges every rb node along the data edge path starting from the first one. After this, each row group contains rb rows. This strategy is suitable for matrices with irregular distribution of non-zeros as each row group contains a similar number of non-zeros and load balancing is achieved. When $rb = 1$ and 32, the strategy is equivalent to those in [8] and [30], respectively.
- *Merge by average non-zeros per row.* The $merge\_avg$ strategy calculates the average number of non-zeros of every rb node along the data edge path. If the average number is no more than $thresh$, the nodes are merged into a larger one. It is based on the intuition that when the numbers of non-zeros in multiple contiguous rows are small, we should group them to avoid idle threads in subwarps. Otherwise, we keep each row as a row group because one single row can be processed by all the threads in a subwarp collaboratively. When $rb = 32$, this strategy is equivalent to that in [38].
- *Merge by the threshold of the number of non-zeros.* Sparse matrices usually form irregular distributions of non-zeros, especially for those associated with power-law graphs [27]. The $merge\_thresh$ strategy is designed for rows with a vast number of non-zeros, which have great impact on load balancing. Specifically, the strategy continually merges nodes along the data edge path, as long as the encountered node has no more than $thresh$ non-zeros and the number of merged nodes does not exceed rb. Each long row with more than $thresh$ non-zeros is then kept as a single row group.

After merging operations, levels of nodes should be recalculated as multiple rows from different levels can be merged into the same node.

The $reorder$ operation renumbers the nodes according to the topological order of the graph. After reordering, rows with the close computation order are put together so that parallelism and locality are ensured simultaneously. Node-merging operations are further performed to enhance the performance. Noting that reordering changes the storage data of the original matrix and brings extra costs for preprocessing, we consider it as an optional choice and evaluate its performance gain separately.

Figure 7 shows the resulting candidate graphs with different graph transformation operations. The original computation graph is shown in Figure 2. After applying the graph operations described above, AG-SpTRSV generates a set of candidate graphs varying in sizes of nodes and connections of dependency and data edges. Each candidate graph (including the original one)

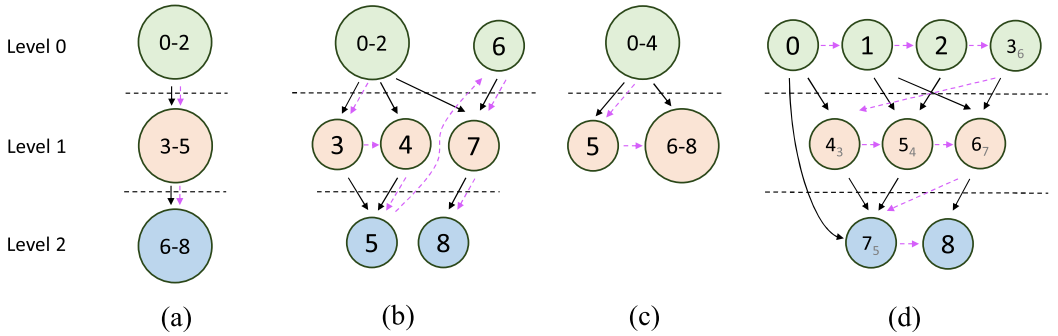


Fig. 7. Examples of computation graph transformation. The candidate graphs are generated by (a) merging by fixed block size where $rb = 3$, (b) merging by average non-zeros per row where $rb = 3$ and $thresh = 1$, (c) merging by the threshold of the number of non-zeros where $rb = 5$ and $thresh = 3$, and (d) reordering by topological order. The nodes are renumbered. The gray subscripts indicate the numbering of the node in the original computation graph.

represents one specific way of parallelization and has the potential to achieve high performance. AG-SpTRSV then schedules the graphs onto GPU parallelization units to generate schemes (this will be discussed in Section 4.4).

4.4 Multi-hierarchy Heuristic Scheduling

Existing work [14, 30, 38] adopts a simple scheduling strategy, which launches blocks/threads corresponding to the number of computation tasks. In this way, task scheduling is entirely handled by the hardware. However, the hardware scheduler may not always bring the best performance. There are possibilities that blocks with smaller indices but later computation order (i.e., a lower topological order in the computation graph with dependency edges) need to wait for dependency data, whereas blocks with larger indices but earlier computation order (i.e., a higher topological order) are not given enough priority in scheduling. Moreover, the hardware scheduler has no information about data exchange requirements or computation task workloads, thus cannot achieve deep performance optimization.

In the *Schedule* stage, AG-SpTRSV dives deeper into scheduling from bottom to top, addressing the following issues across three hierarchies: how nodes in the computation graph are grouped, how grouped nodes are scheduled onto warps, and how grouped nodes are partitioned into different kernels. Based on the three hierarchies, we propose a series of heuristic strategies to enhance the performance.

In the first hierarchy, nodes in the computation graph are grouped into node groups. Each node group contains up to *ng* nodes, where *ng* is the number of subwarps. A node group is regarded as a computation task of a warp, and each node in the group corresponds to one subwarp. To avoid deadlocks, nodes in one group must be independent of each other. Therefore, AG-SpTRSV group nodes in the same level based on the topological order of the computation graph. We consider two heuristic strategies for grouping:

— The *rg_sequential* strategy groups nodes sequentially in the order of their starting row indices. Spatially close nodes are more likely to be grouped together, thus resulting in better data locality.
— The *rg_balance* strategy further sorts the nodes within the same level based on the number of non-zeros and assigns them in decreasing order. Nodes with a similar number of non-zeros are more likely to be assigned to the same warp, thus load balancing can be enhanced.

In the second hierarchy, node groups are scheduled onto different warps. Scheduling is done in the topological order of the node groups (based on the levels of computation graph). The scheduling decisions have a significant impact on performance, as the distribution of tasks across different stream multiprocessors (SMs) determines load balancing. Moreover, threads on the same SM share the on-chip L1 cache, which is closely associated with data locality. AG-SpTRSV implements scheduling with an explicit approach. Specifically, it launches a fixed number of thread blocks (no more than the capacity of the whole GPU). Each thread block corresponds to one specific SM. Warps in the thread block are statically assigned a set of node groups. In this way, AG-SpTRSV is able to determine rigidly on which thread block and in what order the computation tasks are scheduled and executed. For better load balancing, parallelism, and data locality, we design and adopt the following three heuristic scheduling strategies:

— The *ws_rr* strategy schedules node groups onto warps in a round-robin way. It is the simplest way to schedule yet sufficient for matrices with a regular distribution of non-zeros.
— The *ws_balance* strategy aims to improve load balancing. For each node group, the strategy chooses the warp currently with minimum workload. The workload of each node group is approximated as its total number of non-zeros.
— The *ws_locality* strategy aims to improve data locality. The strategy schedules the current node group onto the adjacent warp of its parent. An adjacent warp refers to a warp within the same thread block. The intuition of this strategy is that adjacent warps share an L1 cache, which enables cache-level data sharing. When a node group has more than one parent, the warp with the minimum workload is selected for scheduling.

In the third hierarchy, node groups of different levels are partitioned and scheduled in one or more kernels. AG-SpTRSV analyzes the number of node groups of each level in the computation graph. For adjacent levels with a small number of node groups, we merge them into one kernel for execution. For levels with a large number of node groups, we execute each of them in a single kernel so that costs of synchronization can be eliminated, as is discussed in Section 4.2.2. We use a threshold $thresh\_level$ to determine that

$$thresh\_level = \alpha\_level \times \text{maximum \# of warps},$$

where α_level is a variable parameter. The maximum number of warps is a hardware-dependent parameter, calculated by the maximum number of resident threads on the GPU. If the total number of node groups within the current level is more than $thresh\_level$, it is independently processed in a single kernel so that part of the synchronization overhead can be eliminated. When $\alpha\_level = 0$, each level is processed independently, which is equivalent to the level-set methods [3, 28]. When $\alpha\_level = \infty$, all levels are processed by one kernel, which is equivalent to the fine-grained synchronization methods [8, 14, 30, 38].

Figure 8 shows examples of how candidate graphs are scheduled. With heuristic scheduling strategies, AG-SpTRSV maps computation tasks to the hardware in a multi-hierarchy manner. By now, AG-SpTRSV generates a series of schemes, evaluates the performance, and searches for the best one.

4.5 Automatic Scheme Selection

Compared with the existing work [8, 14, 30, 38], AG-SpTRSV considers a much larger optimization space. As is shown in Table 2, the optimization space consists of parallelism setups (including *nw*, *ng*, and *sw_rnnz*), transformation operations with their parameters, and multi-hierarchy scheduling strategies. We refer to the combination of strategies and parameters as a *scheme*. Within this optimization space, AG-SpTRSV can generate a series of schemes with the potential to achieve

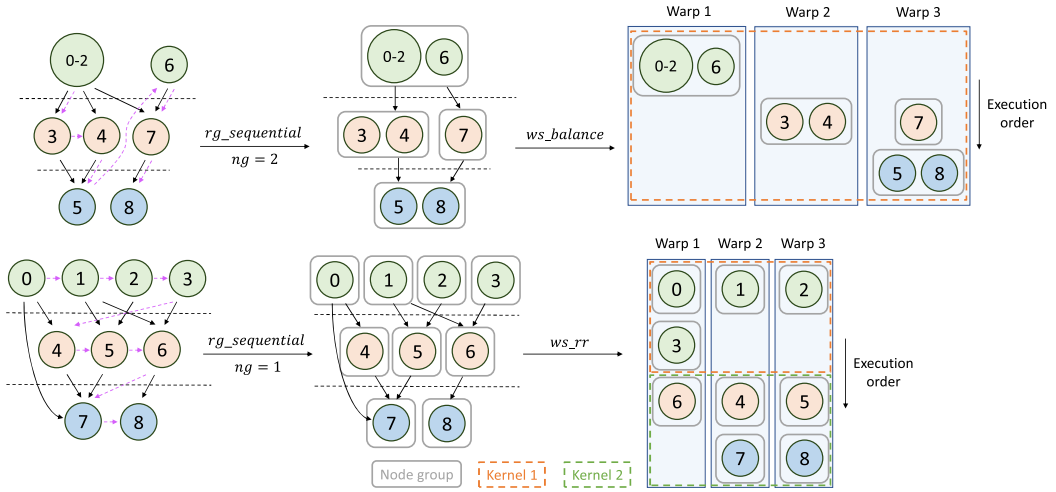


Fig. 8. Examples of multi-hierarchy scheduling. The gray rounded rectangles indicate the node groups, and the dashed rectangles indicate the CUDA kernels. The upper part shows the scheduling of the candidate graph in Figure 7(b) with $ng = 2$, $\alpha\_level = 1$ and 2 warps in total. The *ws_rr* strategy schedules node groups onto warps in sequence, and the first layer is processed independently because it exhibits enough parallelism. The lower part shows the scheduling of the candidate graph in Figure 7(d) with $ng = 1$, $\alpha\_level = 1$ and 3 warps in total. The *ws_balance* schedules node groups onto the warp with the fewest non-zero elements each time, and all node groups are processed by one kernel.

Table 2. The Optimization Space of AG-SpTRSV

Setting	Description	Values
nw	The number of active warps per block	$nw \in \{2, 4, 8, 16, 32\}$
ng	The number of subwarps per warp	$ng \in \{2^i, 0 \le i \le 5\}$
sw_rnnz	The threshold to determine whether the threads in a subwarp collaboratively process the rows	$sw\_rnnz \in \{0, \infty\} \cup \{2^i, i \in \mathbb{N} \ \& \ 2^i \le \frac{32}{ng}\}$
$trans$	The computation graph transformation strategy	$\{reorder, merge\_fixed\ (rb \in \{2^i, 0 \le i \le 5\}),$ $merge\_avg\ (thresh \in \{2^i, 0 \le i \le 5\}, rb = 32),$ $merge\_thresh\ (thresh \in \{2^i, 0 \le i \le 5\}, rb = 32)\}$
$sched$	The heuristic scheduling strategy	$\{rg\_sequential, rg\_balance\} \times$ $\{ws\_rr, ws\_balance, ws\_locality\} \times$ $\alpha\_level \in \{0, 2, 4, 8, \infty\}$

high performance. In the last *Select* stage, AG-SpTRSV executes solving kernels and measures the performance of all the generated schemes. The scheme with the best performance is selected and used for the final target kernel.

Given the extremely large optimization space, the exhaustive search brings excessive preprocessing overhead. According to Table 2, there are over 3.2e5 schemes to be evaluated. In practice, we reduce the search space by empirically eliminating schemes that perform poorly on most matrices. The reduced number of schemes to be evaluated is around 1.6e3. However, the number is still too large to provide an efficient end-to-end solution. The introduced overhead may significantly

Table 3. Input Matrix Features in the Performance Model

Feature	Description
m	Number of rows.
nnz	Number of non-zeros.
rnnz	Average number of non-zeros per row.
rnnz max	Maximum number of non-zeros per row.
rnnz cov	Coefficient of variation of non-zeros per row.
lnnz	Average number of non-zeros per level.
lnnz max	Maximum number of non-zeros per level.
lnnz cov	Coefficient of variation of non-zeros per level.
level num	Number of levels in the original computation graph.
dep dist	The average distance between each row and its nearest dependency row.

impact the end-to-end executing time, making AG-SpTRSV impractical for real-world applications. Thus, we further introduce a lightweight performance model based on historical performance results, which enables quick scheme selection.

The performance model takes matrix information as input. We empirically use 10 features to represent the input matrix, as described in Table 3. These features can effectively represent the matrix size, parallelism, and data dependencies, which can be closely related to the performance of SpTRSV.

m and *nnz* indicate the size of the matrix. *rnnz*, *rnnz max*, and *rnnz cov* indicate the distribution of non-zeros among rows. *lnnz*, *lnnz max*, *lnnz cov*, and *layer num* indicate the distribution of non-zeros among levels. *dep dist* is a customized metric to describe dependency relationships between different rows, calculated as

$$\frac{\sum_{0\le k<m} dep(k)}{m}, \quad dep(k) = \begin{cases} \frac{1}{k-ColIdx[RowPtr[i+1]-2]}, \textbf{if } RowPtr[i+1]-RowPtr[i]>1, \\ 0, \textbf{otherwise}. \end{cases} \tag{1}$$

When *dep dist* is large (close to 1), the parallelism among adjacent rows is limited. For such matrices, using larger block sizes (e.g., larger *rb* for *merge_fixed* strategy) may increase the number of levels in the computation graph, which leads to reduced parallelism. Thus, using smaller block sizes (smaller *rb*) typically achieves better performance. When *dep dist* is small (close to 0), sufficient parallelism can be exploited among adjacent rows. For such matrices, using larger block sizes enables coalesced memory access and exploits better parallelism.

We use a three-layer **multilayer perception (MLP)** as the performance model. The dimension of each hidden layer is 32. The input features are first transformed into degree-2 polynomial features and then normalized with standard deviation. The output data is a 0-1 vector with a dimension equal to the total number of schemes (corresponding to the size of the optimization space). If the performance of a scheme achieves 85% of the best, we consider it a "satisfactory" performance and set the value of the corresponding position in the vector to 1. Otherwise, the value is set to 0. The hidden layers use the ReLU activation, while the output layer uses the sigmoid activation. To find out the best scheme for a specific matrix, we input the matrix information and obtain the prediction result. The scheme corresponding to the position with the highest value in the output vector is selected for the target solving kernel. With the proposed performance model, the preprocessing stage only involves extraction of matrix features, performance model prediction, and selected scheme generation. The costs of exhaustive search can be significantly reduced, whereas the selected scheme can still achieve good performance.

Table 4. Configurations of Experimental Platforms

GPU	Architecture	Memory	FP64 Peak	Bandwidth Peak
Tesla A100	Ampere	SXM4, 80 GB	9.746 TFLOPS	1.935 TB/s
RTX 3080Ti		GDDR6X, 12 GB	532.8 GFLOPS	912.4 GB/s

5 Evaluation

5.1 Experimental Setup

We evaluate AG-SpTRSV on two NVIDIA GPUs: Tesla A100 and RTX 3080Ti. The platform configurations are listed in Table 4. The nvcc compiler version is 11.4+. The host program and the preprocessing stages are both performed on Intel Core i9-12900KF (Alder Lake architecture, 76.8 GB/s DDR4 memory) with Ubuntu 18.04 and gcc 7.5. The optimization flag is set as -O3.

We compare the performance of AG-SpTRSV with three state-of-the-art implementations: the Sync-free method [14], YYSpTRSV [38], and SpTRSV Solve in the NVIDIA cuSPARSE library [23]. To achieve the best performance of YYSpTRSV, we search for its best parameter to switch algorithms from $\{0, 1, 2, 4, 8, 16, 32\}$ and use the best one for evaluation. We employ *cusparseSbsrsv2_solve* in cuSPARSE as the baseline routine and utilize two built-in algorithms: using the level information (*USE_LEVEL*) and not using the level information (*NO_LEVEL*).

We first evaluate 20 matrices, which are representative of a wide range of scientific fields. Information regarding the matrices is listed in Table 5, sorted by *lnnz* (an approximation of the overall parallelism). We divide the matrix into the following three types. *TYPE 1* comprises structured matrices with multiple diagonal-like patterns or banded matrices. This type of matrix has a regular distribution of non-zeros but typically exhibits low parallelism. *TYPE 2* comprises matrices derived from graph problems, with very high sparsity and high parallelism. *TYPE 3* consists of matrices derived from various problems, with uneven and special sparsity patterns, such as locally dense blocks of non-zeros or rows with an excessively high number of non-zeros. We illustrate in Section 5.2 that the performance of AG-SpTRSV is improved with different optimization spaces. We also evaluate the overall performance with 2,219 large matrices (with more than 10^3 non-zeros) in the SuiteSparse Matrix Collection [5] (also known as the University of Florida Sparse Matrix Collection) and show the overall performance comparison in Section 5.3. We also evaluate the ability of the performance model to reduce the search costs while achieving satisfactory performance in Section 5.4.

Without loss of generality, we reserve the lower-triangular-part non-zero elements of all the matrix and add diagonal elements if necessary. All the components in the right-hand-side vector are randomly generated. All experiments are conducted in double precision.

5.2 Performance Gains with Different Optimization Spaces

This subsection analyzes the performance of AG-SpTRSV with the set of matrices listed in Table 5. All presented times in this subsection are for execution only and do not account for preprocessing time. We compare several versions of AG-SpTRSV with different optimization spaces, constructed by adjusting the searching range of parameters. We evaluate four optimization spaces as follows: **MF** indicates using only the $merge\_fixed$ strategy and scheduling by hardware. This is equivalent to the tiling strategy in existing implementations [14, 30, 38]. **NM** indicates using other node-merging operations of computation graphs in the *Transform* stage. Scheduling is still handled by hardware. **MHS** indicates using multi-hierarchy heuristic scheduling strategies in the *Schedule* stage. **RL** indicates using the $reorder$ operation to reorder the computation graph by level. We separately evaluate the performance gain of the $reorder$ operation as it introduces

Table 5. Matrix Information of Test Cases

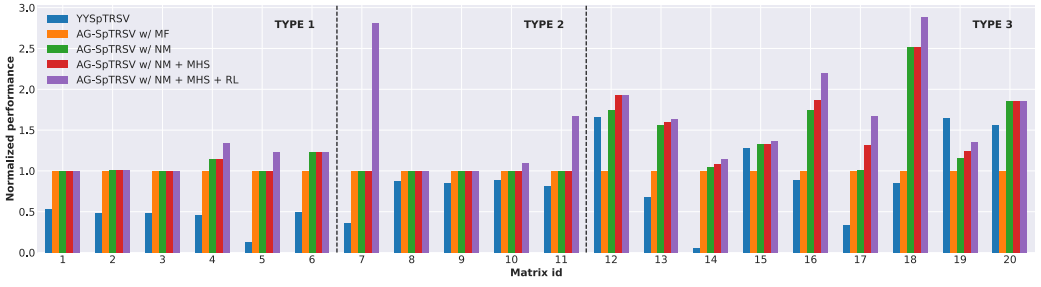
Type	ID	Matrix name	m	mnz	rnnz	lnnz	dep dist
	1	*tmt_sym*	726,713	2,903,837	4.0	1.0	1.00
	2	*cant*	62,451	2,034,917	32.6	26.1	0.79
	3	*chipcool0*	20,082	150,616	7.5	37.6	0.11
TYPE 1	4	*delaunay_n23*	8,388,608	33,554,392	4.0	2,858.1	0.61
	5	*atmosmodd*	1,270,432	5,042,656	4.0	3,609.2	0.99
	6	*nlpkkt80*	1,062,400	14,883,536	14.0	531,200.0	0.00
	7	*europe_osm*	50,912,018	104,966,678	2.1	14,710.2	0.33
	8	*hugetrace* - 00000	4,588,484	11,467,617	2.5	26,677.2	0.62
TYPE 2	9	*hugebubbles* - 00000	18,318,143	45,788,224	2.5	35,227.2	0.62
	10	*road_central*	14,081,816	31,015,229	2.2	238,674.8	0.09
	11	*road_usa*	23,947,347	52,801,659	2.2	311,004.5	0.21
	12	*kron_g500 - logn21*	2097,152	93,138,084	44.4	483.21	0.00
	13	*wiki - Talk*	2,394,385	3,072,221	1.3	4,649.3	0.01
	14	*arabic* - 2005	22,744,080	330,977,515	14.6	7,070.0	0.33
TYPE 3	15	*FullChip*	2,987,012	14,804,570	5.0	9,219.2	0.42
	16	*ASIC_680ks*	682,712	1,505,944	2.2	13,932.9	0.10
	17	*bundle_adj*	513,351	10,360,701	20.2	57,039.0	0.67
	18	*lp1*	534,388	1,088,904	2.0	59,376.5	0.49
	19	*c - big*	345,241	1,343,126	3.9	172,620.5	0.00
	20	*circuit5M*	5,558,326	32,542,244	5.85	308,795.9	0.00

additional preprocessing overhead. To avoid the effect of extreme values, all averages presented in this subsection are geometric averages.

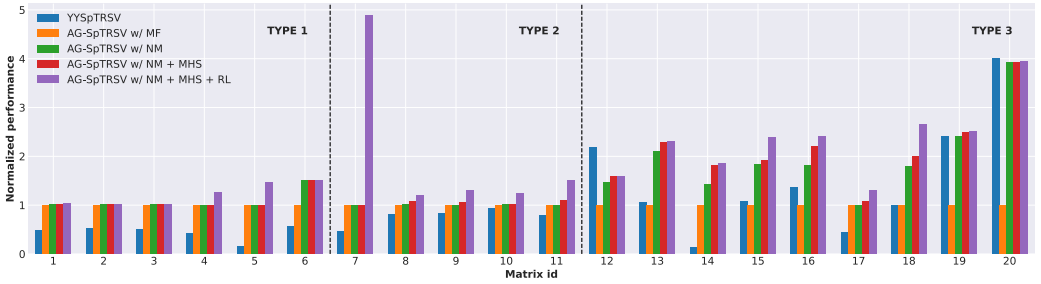
The performance comparison across the different optimization spaces on NVIDIA Tesla A100 is shown in Figure 9(a). For *TYPE 1* matrices, using only the *merge_fixed* strategy can achieve sufficiently good performance. This is because matrices of this type have a regular distribution of non-zeros, which is suitable for the fixed tiling size. Reordering brings some performance gains for matrices with higher parallelism. For *TYPE 2* matrices, reordering can bring significant performance gains. This is due to the great importance of data locality in matrices derived from graphs. For *TYPE 3* matrices, using other merge operations and multi-hierarchy heuristic scheduling can bring good performance gains. This indicates that the designed strategies can effectively handle the irregularity of sparse matrices. Moreover, reordering can still yield significant improvements. Compared with the state-of-the-art YYSpTRSV, enabling the optimization spaces of **MF**, **NM**, **WHS**, and **RL** can achieve speedups of 1.66x, 2.02x, 2.08x, and 2.38x, respectively.

The performance comparison across the different optimization spaces on NVIDIA RTX 3080 Ti is shown in Figure 9(b). The performance results exhibit similar trends as those observed on A100. Enabling the four optimization spaces achieves speedups of 1.37x, 1.85x, 1.95x and 2.37x over YYSpTRSV respectively. It is worth noting that in many cases, the different optimization spaces may achieve varying levels of performance gains on the two hardware platforms. The best schemes differ across hardware platforms, which indicates the importance of strategy searching and auto-tuning to achieve performance scalability.

Based on experimental data, AG-SpTRSV can achieve promising performance on almost all matrices. However, Figure 9 also shows slight performance degradation on a few matrices compared with YYSpTRSV, such as matrices 12 and 19. Based on our observation, AG-SpTRSV may not perform that well on some matrices with small *dep_dist*. We speculate that such matrices have simpler



(a)



(b)

Fig. 9. Performance comparison across different configurations of the optimization space. All performance data are normalized by that of the MF version. The experiments are conducted on (a) NVIDIA Tesla A100 and (b) NVIDIA RTX 3080 Ti.

data dependencies and unrestricted parallelism. The graph transformation and heuristic scheduling strategies of AG-SpTRSV may introduce unnecessary overhead.

5.3 Overall Performance Comparison

This subsection evaluates the performance of AG-SpTRSV with matrices in the SuiteSparse Matrix Collection. All presented times here are for execution only and do not account for preprocessing time. To avoid interference from extremely small matrices during the evaluation, we filter out matrices with less than 10^3 non-zeros. Except for a small number of matrices that cannot be evaluated due to insufficient GPU memory, we use 2,219 matrices in total and sort them based on their number of non-zeros.

Performance results with SuiteSparse on NVIDIA Tesla A100 are shown in Figure 10(a). The geometric average speedups of AG-SpTRSV over Sync-free, cuSPARSE and YYSpTRSV are 3.81x, 3.99x, and 2.14x, respectively. The arithmetic average speedups over the three baselines are 10.56x, 7.35x, and 2.59x. Out of 2,219 test cases, AG-SpTRSV achieves better performance than the three baselines in 96.8% of the cases. The figure also shows that when the number of non-zeros of the matrix increases, the speedups of AG-SpTRSV become more evident and stable, which indicates the advantages of AG-SpTRSV on larger matrices. For large matrices (with more than 10^6 non-zeros), AG-SpTRSV can achieve the geometric speedups of 7.97x over Sync-free, 4.47x over cuSPARSE, and 2.77x over YYSpTRSV.

Performance results with SuiteSparse on NVIDIA RTX 3080 Ti are shown in Figure 10(b). The geometric average speedups of AG-SpTRSV over Sync-free, cuSPARSE, and YYSpTRSV are 2.98x, 3.57x, and 2.12x, and the arithmetic average speedups are 8.25x, 5.36x, and 2.59x, respectively.

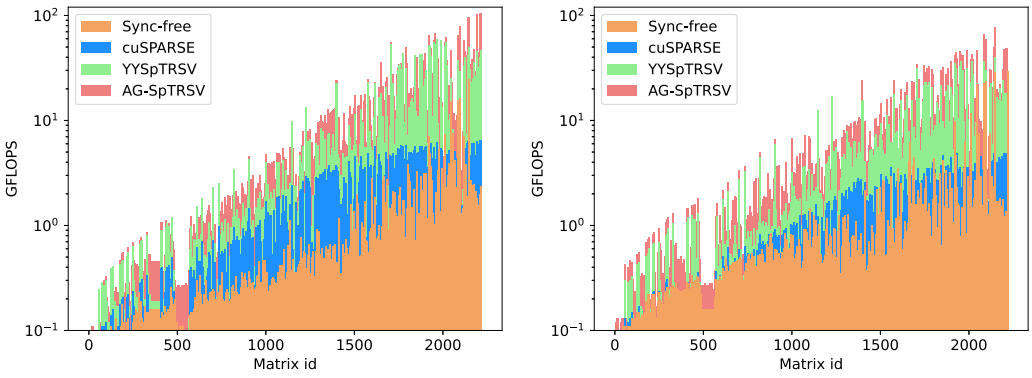


Fig. 10. Performance comparison across Sync-free, cuSPARSE, YYSpTRSV, and AG-SpTRSV with matrices ($> 10^3$ non-zeros) in SuiteSparse Matrix Collection. The experiments are conducted on (a) NVIDIA Tesla A100 and (b) NVIDIA RTX 3080 Ti.

AG-SpTRSV achieves better performance than the two baselines in 94.8% test cases. The figure also indicates the performance advantages of AG-SpTRSV on larger matrices. For matrices with more than 10^6 non-zeros, the geometric average speedups over the three baselines are 8.26x, 3.74x, and 2.95x, respectively.

5.4 Evaluation of Performance Model

This subsection evaluates the performance model described in Section 4.5. For each matrix, we compare the best performance of all schemes (noted as *Best Perf*) with the performance of the scheme selected by the performance model (noted as *Selected Perf*). The dataset contains all performance measurement results in Section 5.3. Specifically, we use performance data of all SuiteSparse matrices. For each matrix, the execution times with all possible schemes are evaluated and collected. We divide 70% of the dataset into the training set and 30% into the test set. The **Mean Absolute Percentage Error (MAPE)** between *Selected Perf* and *Best Perf* is 8.19%. Moreover, *Selected Perf* achieves at least 85% of *Best Perf* in 87.1% of the test cases. The results indicate that the performance model is able to generate the scheme with performance competitive to the best one.

We compare the preprocessing and solving time of AG-SpTRSV using **exhaustive search (ES)** and using the **performance model (PM)** with existing GPU implementations including **cuSPARSE (CUSP)** [23], **YYSpTRSV (YY)** [38], and the **Recursive Block Algorithm (REC)** [20].

The results on NVIDIA RTX 3080 Ti are shown in Table 6(a). By using the performance model, the average preprocessing time of AG-SpTRSV is reduced by a factor of more than 392x, while the average execution time increases by 17.3%. The results on NVIDIA Tesla A100 are shown in Table 6(b). The average preprocessing time is reduced by a factor of more than 428x, whereas the average execution time increases by 13.9%.

The preprocessing time ranges from 2.7 to 245 times of the execution time. Compared with auto-tuning frameworks for other matrix computation kernels ([4, 7, 25]), which take hours to days, AG-SpTRSV provides a more efficient end-to-end solution. Many real-world applications [6, 22] involve multiple iterations of SpTRSV on the same sparse matrix, which can amortize the costs of preprocessing.

Compared with other GPU implementations, AG-SpTRSV achieves a balance between the execution performance and preprocessing costs. The preprocessing time of AG-SpTRSV is longer than

Table 6. Average Preprocessing and Execution Time Across Different SpTRSV Implementations

(a)

Matrix type	Preprocessing time(s)					Execution time(s)				
	CUSP	YY	REC	ES	PM	CUSP	YY	REC	ES	PM
TYPE 1	7.98e-1	9.98e-3	39.9	158.62	4.04e-1	3.98e-1	2.44e-1	1.58e-1	1.14e-1	1.21e-1
TYPE 2	4.81e-1	9.72e-2	306.2	1,227.32	2.41	2.79e-2	1.82e-2	8.72e-3	0.98e-3	1.07e-2
TYPE 3	3.30e-1	2.16e-2	–	376.47	8.87e-1	9.06e-1	6.49e-2	–	1.98e-2	2.94e-2

(b)

Matrix type	Preprocessing time(s)					Execution time(s)				
	CUSP	YY	REC	ES	PM	CUSP	YY	REC	ES	PM
TYPE 1	7.31e-1	7.36e-3	18.98	158.62	3.71e-1	2.92e-1	2.94e-1	9.57e-2	1.39e-1	1.55e-1
TYPE 2	5.90e-1	7.22e-2	141.34	1,227.32	2.36	5.17e-2	2.52e-2	6.52e-3	1.25e-2	1.27e-2
TYPE 3	2.43e-1	1.63e-2	–	376.47	1.48	5.93e-1	7.54e-2	–	2.25e-2	2.76e-2

The experiments are conducted on (a) NVIDIA Tesla A100 and (b) NVIDIA RTX 3080 Ti. Missing data points indicate crashes caused by runtime errors.

that of cuSPARSE and YYSpTRSV (by 2.44x and 43.28x on average), but the execution performance of AG-SpTRSV is higher (with speedups of 10.69x and 2.09x on average). The REC achieves better performance in execution time (1.39x over AG-SpTRSV on average). However, its preprocessing overhead is too high for real-world applications. On average, the preprocessing time of REC is over 10^5x of the execution time. The number of iterations of solvers in practice rarely achieves this order of magnitude.

Our experimental results also provide suggestions for the selection of SpTRSV implementations: For direct solvers or iterative solvers with few iterations, YYSpTRSV performs better due to its lightweight preprocessing stage. For cases with a significant number of iterations and unchanged distribution of non-zeros, the REC may achieve better execution performance. For typical iterative solvers (with the number of iterations ranging $10^2 \sim 10^5$), AG-SpTRSV is a better choice as it offers input-adaptive optimization at a relatively low preprocessing cost.

Furthermore, we find that for *TYPE 1* and *TYPE 2* matrices, performance models can predict the schemes with performance close to the optimal one (within a margin of error of 5.5%). For *TYPE 3* matrices, the margin of error becomes larger (up to 32.8%), as the irregular distribution of non-zeros affects the accuracy of prediction results.

We also evaluate the time consumed by each preprocessing stage of AG-SpTRSV using PMs. The results are shown in Figure 11. The time consumed by the *Select* stage remains basically unchanged, as the overhead of the performance model prediction is fixed. On most large matrices (e.g., with more than 10^7 non-zeros), the *Transform* stage and *Schedule* stage consume the majority of the time. The time consumption of the above two stages is closely related to the matrix sizes (the number of nodes and the number of non-zeros). The relationship is not strictly linear, as different non-zero distributions and different strategy selections lead to variation in time consumption.

6 Related Work

How to implement efficient SpTRSV on modern processors has long been a topic in the field of sparse matrix computation. Existing methods are generally based on level-set [3, 28] and color-set [29] methods. Level-set methods divide the problem into several levels, which represent the order of computation. No intra-level data dependencies exist so that computation tasks inside one single level can be fully parallelized. Different levels have to be solved in a sequential order to ensure inter-level dependencies. Color-set methods view the solution of a linear system as a graph coloring problem and reorder the original matrix to enhance parallelism [31]. Color-set

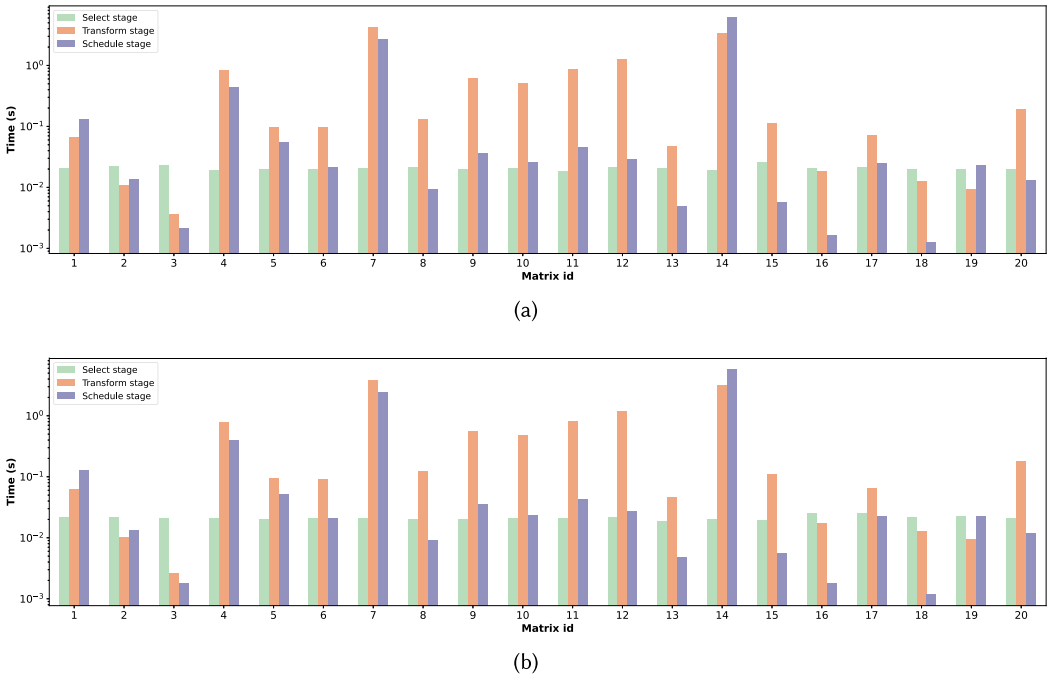


Fig. 11. Breakdown of preprocessing time. The experiments are conducted on (a) NVIDIA Tesla A100 and (b) NVIDIA RTX 3080 Ti.

methods are mostly used in iterative solvers instead of triangular solve kernels; thus, we focus on the implementation and optimization of level-set methods.

Level-set methods require synchronizations at the end of each level, which introduces high extra costs. Park et al. [24] proposed to replace barrier synchronization with **peer-to-peer (P2P)** communication on a central processing unit (CPU) and further reduce the cost with heuristic sparsification. On GPUs, Liu et al. [14] proposed a warp-level synchronization-free method to eliminate unnecessary cost between level-sets based on the CSC format. The method parallelizes the solution of the component of each row inside one CUDA warp, using shared memory and global memory for intra-block and inter-block communication, respectively. Dufrechou and Ezzatti[8] proposed a similar sync-free method for the CSR format, with row-wise flags to indicate dependencies. Li and Zhang [13] compared the performance between the sync-free method and the level-set method when using the CSR and CSC formats, in real-world numerical scenarios such as Gauss-Seidel iterations. Su et al. [30] further proposed a fine-grained thread-level method, which solves multiple rows within one single thread for matrices with high sparsity or many extra levels. Zhang et al. [38] proposed YuenyeungSpTRSV, which combines warp-level and thread-level methods and chooses between them based on the average number of non-zeros per row. However, optimization space in existing work is not enough to reach the best performance. AG-SpTRSV considers adaptive parallelization, graph transformation and multi-hierarchy scheduling, and outperforms the state-of-the-art SpTRSV implementations.

Considering the performance differences between various SpTRSV implementations on different matrices, dynamic selection strategies have been proposed. Ahmad et al. [1] proposed to switch between CPU and GPU routines based on a trained machine learning model (Random Forest). Dufrechou et al. [9, 10] employed and evaluated several machine learning models for the optimal

selection of GPU implementations. Due to the lack of real matrix datasets, artificial matrix data was generated to improve prediction accuracy. The Split Model [2] proposed to decompose SpTRSV into multiple executions based on the data dependency graph. Heuristic strategies are introduced to select the optimal CPU/GPU implementation for each execution.

Implementations of SpTRSV with various data formats have also been studied well in recent years. Lu et al. [20] proposed an efficient recursive block algorithm to reduce memory footprints and enhance data locality. Yamazaki et al. [35] developed the sparse triangular solver based on supernode structures, which can improve computational intensity and provide performance portability. Wang et al. [34] designed the Sparse Level Tile Layout to enable data reuse and reduce communication costs on Sunway manycore processors. tileSpTRSV [19] stores the sparse matrix in a 2D block and uses heuristic strategies to specify the optimal format for each block. Yilmaz [36, 37] aimed to reduce the synchronization overhead of the level-set method through dependency graph transformation and equation rewriting. It should be noted that costs of accessing sparse matrix data can be significantly reduced through specialized code generation. However, using a specific format limits the generality of the implementation and introduces extra costs in format transformation. Our work focuses on the most commonly used CSR format and provides an efficient solution for general purposes.

7 Conclusion

SpTRSV plays an important role in scientific applications. However, implementing efficient SpTRSV on GPUs faces challenges. In this article, We attempt to characterize the performance of SpTRSV through experiments and identify several key factors, including parallelism setup, data distribution and code implementation. We propose AG-SpTRSV, an automatic framework to optimize SpTRSV on GPUs, which takes those factors into consideration. Through the four stages — code variant preparation, computation graph transformation, multi-hierarchy heuristic scheduling, and scheme selection — AG-SpTRSV is able to generate a highly optimized solution kernel for the specific matrix with input adaptability. The tedious manual tuning efforts can be fully eliminated. Experimental results with SuiteSparse Matrix Collection on two NVIDIA GPUs show speedups over the state-of-the-art implementations and capability of providing efficient end-to-end solution.

The underlying idea behind our work is not limited to the scope of SpTRSV and GPU. In future work, we will attempt to extend the proposed framework to various platforms and other matrix computation kernels in scientific applications, such as sparse LU or QR factorization.

ACKNOWLEDGMENT

We would like to thank the editor and the anonymous reviewers for their valuable feedback that improved the article.

References

[1] Najeeb Ahmad, Buse Yilmaz, and Didem Unat. 2020. A prediction framework for fast sparse triangular solves. In *European Conference on Parallel Processing*. Springer, 529–545.

[2] Najeeb Ahmad, Buse Yilmaz, and Didem Unat. 2021. A split execution model for SpTRSV. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (2021), 2809–2822.

[3] Edward Anderson and Youcef Saad. 1989. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing* 1, 01 (1989), 73–95.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[5] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38, 1 (2011), 1–25.

[6] Richard C. Dorf. 2018. *Electronics, Power Electronics, Optoelectronics, Microwaves, Electromagnetics, and Radar*. CRC press.
[7] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. 2022. AlphaSparse: Generating high performance SpMV codes directly from sparse matrices. *arXiv preprint arXiv:2212.10432* (2022).
[8] Ernesto Dufrechou and Pablo Ezzatti. 2018. Solving sparse triangular linear systems in modern GPUs: a synchronization-free algorithm. In *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE, 196–203.
[9] Ernesto Dufrechou, Pablo Ezzatti, Manuel Freire, and Enrique S. Quintana-Ortí. 2021. Machine learning for optimal selection of sparse triangular system solvers on GPUs. *J. Parallel and Distrib. Comput.* 158 (2021), 47–55.
[10] Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí. 2019. Automatic selection of sparse triangular linear system solvers on GPUs through machine learning techniques. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 41–47.
[11] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra*. SIAM.
[12] Ang Li, Weifeng Liu, Mads R. B. Kristensen, Brian Vinter, Hao Wang, Kaixi Hou, Andres Marquez, and Shuaiwen Leon Song. 2017. Exploring and analyzing the real impact of modern on-package memory on HPC scientific kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[13] Ruipeng Li and Chaoyu Zhang. 2020. Efficient parallel implementations of sparse triangular solves for GPU architectures. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 106–117.
[14] Weifeng Liu, Ang Li, Jonathan Hogg, Iain S. Duff, and Brian Vinter. 2016. A synchronization-free algorithm for parallel sparse triangular solves. In *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22*. Springer, 617–630.
[15] Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.
[16] Weifeng Liu and Brian Vinter. 2015. A framework for general sparse matrix–matrix multiplication on GPUs and heterogeneous processors. *J. Parallel and Distrib. Comput.* 85 (2015), 47–61.
[17] Weifeng Liu and Brian Vinter. 2015. Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Comput.* 49 (2015), 179–193.
[18] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2010. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Vol. 20.
[19] Zhengyang Lu and Weifeng Liu. 2023. TileSpTRSV: A tiled algorithm for parallel sparse triangular solve on GPUs. *CCF Transactions on High Performance Computing* 5, 2 (2023), 129–143.
[20] Zhengyang Lu, Yuyao Niu, and Weifeng Liu. 2020. Efficient block algorithms for parallel sparse triangular solve. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.
[21] Kiran Matam, Siva Rama Krishna Bharadwaj Indarapu, and Kishore Kothapalli. 2012. Sparse matrix-matrix multiplication on modern architectures. In *2012 19th International Conference on High Performance Computing*. IEEE, 1–10.
[22] L. McInnes, B. Norris, Sanjukta Bhowmick, and P. Raghavan. 2003. Adaptive sparse linear solvers for implicit CFD using Newton-Krylov algorithms. In *Proceedings of the 2nd MIT Conference on Computational Fluid and Solid Mechanics*, Vol. 2. 1024–1028.
[23] Maxim Naumov, L. Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.
[24] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *International Supercomputing Conference*. Springer, 124–140.
[25] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices* 48, 6 (2013), 519–530.
[26] Yousef Saad. 2003. *Iterative Methods for Sparse Linear Systems*. SIAM.
[27] Alessandra Sala, Haitao Zheng, Ben Y. Zhao, Sabrina Gaito, and Gian Paolo Rossi. 2010. Brief announcement: Revisiting the power-law degree distribution for social graph analysis. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. 400–401.
[28] Joel H. Saltz. 1990. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM Journal on Scientific and Statistical Computing* 11, 1 (1990), 123–144.
[29] Robert Schreiber and W. Tang. 1982. Vectorizing the conjugate gradient method. *Symposium on CYBER 205 Applications* (1982).
[30] Jiya Su, Feng Zhang, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. 2020. CapelliniSpTRSV: A thread-level synchronization-free sparse triangular solve on GPUs. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[31] Brad Suchoski, Caleb Severn, Manu Shantharam, and Padma Raghavan. 2012. Adapting sparse triangular solution to GPUs. In *2012 41st International Conference on Parallel Processing Workshops*. IEEE, 140–148.

[32] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. 2016. Automating wavefront parallelization for sparse matrix computations. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 480–491.

[33] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel transposition of sparse data structures. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–13.

[34] 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 *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 338–353.

[35] Ichitaro Yamazaki, Sivasankaran Rajamanickam, and Nathan Ellingwood. 2020. Performance portable supernode-based sparse triangular solver for manycore architectures. In *Proceedings of the 49th International Conference on Parallel Processing*. 1–11.

[36] Buse Yilmaz. 2021. Graph transformation and specialized code generation for sparse triangular solve (SpTRSV). *arXiv preprint arXiv:2103.11445* (2021).

[37] Buse Yılmaz. 2023. A novel graph transformation strategy for optimizing SpTRSV on CPUs. *Concurrency and Computation: Practice and Experience* 35, 24 (2023), e7761.

[38] Feng Zhang, Jiya Su, Weifeng Liu, Bingsheng He, Ruofan Wu, Xiaoyong Du, and Rujia Wang. 2021. YuenyeungSpTRSV: A thread-level and warp-level fusion synchronization-free sparse triangular solve. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2321–2337.

Received 18 January 2024; revised 6 May 2024; accepted 5 June 2024