Exploring and Optimizing Sparse Matrix Computations on GPUs: From SpMV to SpTRSV

Derron Li, McMaster University

This document overviews the current field of sparse matrix-vector multiplication (SpMV) and sparse lower triangular solves (SpTRSVs), particularly their implementation on GPUs and optimization techniques. The document starts with a preamble on how various implementations were compared against each other in my experiments. Next I go into the domain of SpMV and SpTRSVs. Finally, I discuss next steps for this work.

1 Experiment Design

All sparse matrices used for testing were randomly sourced from the SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection) [1]. For `SpTRSV` tests, they were intentionally selected to meet the criteria discussed in the "Challenges" section. For `SpMV` tests, the matrix was randomly selected from various application categories (i.e., Graph, Fluid Dynamics, etc.), the goal being to assess performance across various sparsity structures. The left-hand side vector is randomly filled with float values. Following, the input matrix is parsed to CSR format and in the case of `SpTRSV` tests, made lower triangular by filling any values above the diagonal with zeros. The resulting matrix is given to two implementations, `cuSPARSE` and my own ("manual") implementation. The respective algorithms are run 5 times, while tracking the start and end time of each run. The arithmetic mean of execution time across the trials is then calculated.

Throughput is calculated by

$$\frac{2 \times \text{number of non-zero elements}}{\text{execution time}},$$

memory bandwidth is calculated by

$$\frac{\text{total memory accessed during execution}}{\text{execution time}}.$$

These are the performance metrics used for comparison between any two pairs of implementations.

To ensure correctness, the two solution vectors returned by the two implementations are compared against each other by calculating the cumulative mean squared error (MSE). This approach was chosen over simply looking at relative error (the difference between every pair of values) because MSE gives a more holistic metric for the *floating point* accuracy, since it maintains a running sum of the differences. Performance metrics will only be printed if the MSE falls under a certain threshold; otherwise, an error is thrown.

All experiments were run on an NVIDIA RTX4090 provided by McMaster CAS department GPU clusters.

2 Sparse Matrix-Vector Multiplication

Sparse matrix-vector multiplication (`SpMV`) is a fundamental operation in numerous computing and engineering applications. The sparsity of a matrix is characterized by the predominance of zero entries, which allows for specialized storage formats and computational techniques that avoid unnecessary operations on zero elements. Uncovering algorithms which optimally leverage GPU resources and the inherent sparsity of `SpMV` has become the subject of extensive research.

I highlight a select few matrix storage formats and algorithms below. Several others were explored, but these were found to be the most popular and efficient.

2.1 CSR-Scalar and CSR-Vector

Compressed Sparse Row (CSR) format is likely the most popular general-purpose sparse matrix format [2]. Of which there are two commonly documented algorithms for `SpMV`: "Scalar" and "Vector" implementation.

`CSR-Scalar` is the most straightforward implementation, as it operates with the granularity of one thread per row. It has significant weak points in load balancing, strided memory access (poor memory coalescing), and thread divergence [3]. Thus, it is not discussed further.

`CSR-Vector` overcomes many of the above shortcomings by instead operating at the granularity of one warp per row. By doing this, threads within a warp will access adjacent non-zero elements to operate on which greatly improves parallelism and memory coalescing. Naturally, this takes better advantage of spatial locality, resulting in more cache hits and less latency which comes from retrieving data from global memory. Moreover, this storage format is one of the best for the general `SpMV` case, as it effectively handles cases where there are a variable number of nonzeros per row [3]. Given the popularity of the CSR format, there is usually minimal preprocessing overhead from needing to convert the matrix format. Together, this makes `CSR-Vector` both an efficient and portable `SpMV` implementation [2]. **This is one of my selected implementations for testing.**

There are however some disadvantages to this implementation, one of which is the memory access of the x vector (assuming $y = Ax$). Lookups in the x vector are entirely random and dependent on the column indices of nonzero elements in each row. This makes cache misses a potential bottleneck [4]. Additionally, there is some synchronization overhead given the complexity of coordinating threads within a warp.

2.2 ELL and Hybrid

The primary advantage of the Ellpack (ELL) format arises from nonzero elements instead being stored in column major format. Where all rows are padded to a length K (where K is the maximum number of nonzeros in a row). This structure results in similar advantages to `CSR-Vector` with the addition of less thread divergence and better load balancing. This is because the padding helps to fix the number of iterations per thread. The primary concern with ELL is the risk of wasting significant amounts of memory since all rows will be padded to match the longest row. In these irregular cases, there would also be load imbalance between threads. This makes ELL most efficient on structured data where the number of nonzero elements does not differ largely between rows [4]. The ELL format also continues to have the disadvantage of random memory lookups in the x-vector, as described above.

However, the principal shortcoming created by padding can be addressed by combining the ELL format with the coordinate (COO) format. The COO format simply stores each row, column index as a pair. By storing the "typical" number of nonzeros in ELL and applying the COO format on elements that are part of extra long rows, we eliminate the possibility of wasting memory on large amounts of padding and thus improve efficiency on more general/irregular matrices. The disadvantage to this Hybrid approach is the added complexity in identifying when exactly to apply ELL and when to use COO [2, 5]. There is also the aspect of needing to convert between different matrix formats, if the input were given in CSR for example.

2.3 Results

Performance metrics can be seen in Table 1. Speedup can also be visualized in Figure 1. We can see that in all cases, `CSR-Vector` resulted in a speedup over `cuSPARSE`, going as high as 13.73x faster. The magnitude of speedup likely depends on sparsity structure, exploring the factors which contribute to this are a worthy next step.

2.4 Applications of `SpMV`

The focus of this subsection is to touch upon `SpMV`'s importance in artificial intelligence (AI) applications, especially as they relate to large-language models. One such application is in the transformer model, a deep learning architecture developed by Google in 2017 [6]. Transformer models leverage a set of techniques called "attention", which helps LLMs to understand the relationship between words. Specifically, the sparse attention [7] mechanism utilizes `SpMV`. Furthermore, graph neural networks (GNNs) are another important application. This is because all graphs can be represented in the form of an adjacency matrix, which is often sparse. GNNs leverage `SpMV` for message passing between nodes and are a powerful tool for natural language processing tasks.

Other relevant use cases include optimization problems and Latent Semantic Indexing (LSI) [8]. This section only scrapes the surface of `SpMV` applications.

3 Sparse Lower Triangular Solves

Sparse triangular solves (`SpTRSV`) solve linear systems of the form $Lx = b$, where L is a sparse lower triangular matrix, b is a dense vector, and x is a vector of unknowns to be solved. `SpTRSV`s are of vital importance to linear algebra fields but are one of the most challenging operations to optimize through parallelism and GPU programming due to the inherent dependencies between components of the solution vector.

3.1 Background Overview

`SpTRSV` in its most basic form is solved completely sequentially, row by row. Without leveraging parallelism, this algorithm is extremely inefficient. Level-set `SpTRSV` is a better approach which partitions components (rows) into different "level-sets" based on their dependencies on other components. This enables components in the same level-set to be solved in parallel, greatly improving throughput [9]. However, the need for a costly preprocessing step and thread synchronizations before the next level set can be started, are considered the primary disadvantages to this implementation [9, 10].

3.2 Synchronization-Free `SpTRSV`

To address the aforementioned disadvantages of the level-set implementation, synchronization-free `SpTRSV` was proposed by Liu et al. [10]. To overview the algorithm, a much shorter preprocessing stage acts to identify the in-degree of each component (i.e., how many things need to be computed prior to this component). Then in the solve phase, each component, x_i, is processed by a GPU warp. Initially, busy-waiting until all of its dependencies have been resolved. Finally, once the component x_i has been solved, it notifies all the later entries that depend on x_i using atomic updates. Thus, effectively eliminating the need for expensive synchronizations. Presently this algorithm is considered the state-of-the-art `SpTRSV` algorithm [9]. There are, however, still some flaws to this algorithm specifically when the average number of components is large and the average number of nonzero elements per row is small. Su et al. tackle this in their synchronization-free thread-level `SpTRSV` [9].

3.3 Initial Implementation

`SpTRSV` implementation was generally a lot more challenging than `SpMV`. Several implementations were attempted. To start, the simplest form `csr_sptrsv_sequential_kernel()` was implemented. This was followed by `cusparse_sptrsv()`, NVIDIA's built-in library for optimized solving. These two implementations were compared against each other and as expected, the `cuSPARSE` library was several times

faster than the sequential implementation (results were not reported because this was deemed trivial).

Next was `csr_sptrsv_syncfree_kernel()`. This kernel went through a great deal of iterations. Stemming from the concepts presented by Liu et al. [10]. When testing began, the outcome would be one of two cases: 1. The process would continue past a reasonable length of time, without producing any output. 2. An output would be produced but the performance metrics of the synchronization-free (manual) implementation would be drastically longer than the `cuSPARSE` metrics. See figures in Appendix A. Such results are a clear indication of some sort of synchronization issue between processing threads/warps. Since this algorithm uses mechanisms such as `threadFence()` and updating an `is_solved` flag once a component is solved, I predict there are race conditions in the code due to how `is_solved` is being updated/accessed by the concurrent threads.

As to what the specific issue was, I couldn't figure out, and perhaps beyond my level of knowledge. I attempted various implementations, sourced from 3 different articles [10, 11, 12], but all of them resulted in the same two issues mentioned above.

Challenges: In addition to likely race conditions as described above, there were several other notable challenges/considerations. Input matrices needed to meet a strict set of requirements for these simple lower `SpTRSV` to be able to solve the system. To begin, the input matrix had to be parsed into a lower triangular matrix. Next, the input matrix must not be singular (cannot have zeros on the diagonal) because singular matrices have no unique solutions. There were also data type restrictions on the values in the input matrix. These were all considerations discovered as I coded my `SpTRSV` implementations. It is also very possible, that the issues seen above stemmed from mistakes in this input handling, which is a limitation to my experiments. I also found that the sparsity structure of the input matrix was especially important to `SpTRSV` performance, which tended to confuse my interpretation of results. To address these challenges, I imposed filters on the matrices I selected from the `SuiteSparse` website and also had extensive input processing in the `read_matrix_market()` function.

3.4 Recursive Block `SpTRSV`

Despite the success of the above mentioned `SpTRSV`s, a key optimization can be observed by understanding the mechanisms behind how a forward substitution solve works. In essence, everything prior to the diagonal element in a row needs to have its dot product calculated. Perfect for an `SpMV` algorithm, which we know, is much easier to parallelize and is generally faster [11]. This is the basis of the block approach to `SpTRSV`. We divide the lower triangular matrix into a series of smaller triangular and square submatrices. By doing this, we get significantly better spatial locality and increase the cache hit rate. Three different blocking approaches were discussed by Lu et al. [11], and the recursive approach was identified to be optimal. This is the approach I attempt to implement in my manual `SpTRSV` implementation.

My implementation is generally simpler than presented by Lu et al. I simply divide the input by 2 to create blocks, rather than adapting is based on non-zero count. I also hardcoded the maximum depth of recursion to be 2. A future implementation would preferably adapt this depth based on the size of the matrix, but I found this required a lot more consideration as there was a large trade-off between recursion overhead and the size of each subproblem.

3.5 Results

Performance metrics for `cuSPARSE` vs. Recursive Block `SpTRSV` are presented in Table 2. Several other matrices were tested, but these two matrices are nearly perfectly representative of my observations. There would be situations where Recursive Block would significantly outperform `cuSPARSE`, but there would also be situations where Recursive Block would be substantially slower. Based on my experiments, I predict this has something to do with the size of the matrix and/or the number of non-zero elements in the matrix. Naturally, smaller matrices likely don't benefit as much from the recursive block algorithm, and perhaps just fall victim to extra overhead. Whereas larger matrices have more opportunities to take advantage of the higher throughput `SpMV`.

3.6 Applications

`SpTRSV` are a key component in iterative methods for solving large, sparse linear systems, such as those arising from the discretization of partial differential equations in computational fluid dynamics and structural mechanics [13]. Additionally, `SpTRSV` finds applications in network analysis, graph processing, machine learning, and optimization problems where sparse matrix computations are prevalent [12].

4 Next Steps

While this report has demonstrated the performance variability of sparse matrix-vector multiplication (`SpMV`) and sparse triangular solve (`SpTRSV`) implementations across different matrix structures, a key takeaway is that no single format or algorithm is universally optimal. Each storage format and solving technique exhibits strengths and weaknesses that depend heavily on matrix characteristics such as sparsity pattern, dimensionality, and non-zero distribution. As such, future work should try to uncover more of the factors influencing performance. And with more sophisticated kernels we can explore dynamic format selection that allow compilers to adaptively choose the most appropriate sparse kernel and format based on input characteristics. Recent advances in domain-specific languages like TVM for compiler infrastructure offer promising pathways to embed such decision logic directly into optimization pipelines for GPU-accelerated code generation [14].

Extending this work into the context of high-performance computing and machine learning presents substantial opportunities. Sparse computations are increasingly vital in AI workloads such as transformers with sparse attention, graph neural networks, and large-scale optimization problems—all of which require scalable, fine-tuned sparse primitives [7]. Further optimization of `SpTRSV` and `SpMV` on heterogeneous architectures could significantly reduce training and inference time for these models. Exploring how sparse solvers can be compiled into end-to-end pipelines for scientific computing or neural network inference is a highly promising direction [15].

References

[1] "SuiteSparse Matrix Collection." Accessed: Apr. 10, 2025. [Online]. Available: https://sparse.tamu.edu/

[2] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Portland Oregon: ACM, Nov. 2009, pp. 1–11. https://doi.org/10.1145/1654059.1654078.

[3] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, vol. 2, no. 5, 2008.

[4] G. Etvushenko, "Sparse Matrix-Vector Multiplication with CUDA," Analytics Vidhya. Accessed: Apr. 10, 2025. [Online]. Available: https://medium.com/analytics-vidhya/sparse-matrix-vector-multiplication-with-cuda-42d19187f8f

[5] "Sparse matrix vector multiplication - part 1 - AMD lab notes," AMD GPUOpen. Accessed: Apr. 10, 2025. [Online]. Available: https://gpuopen.com/learn/amd-lab-notes/amd-lab-notes-spmv-docs-spmv_part1/

[6] R. Merritt, "What Is a Transformer Model?," NVIDIA Blog. Accessed: Apr. 10, 2025. [Online]. Available: https://blogs.nvidia.com/blog/what-is-a-transformer-model/

[7] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating Long Sequences with Sparse Transformers," Apr. 23, 2019, *arXiv*:1904.10509. https://doi.org/10.48550/arXiv.1904.10509.

[8] "Sparse Matrix-Vector Multiplication - an overview — ScienceDirect Topics." Accessed: Apr. 10, 2025. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/sparse-matrix-vector-multiplication

[9] J. Su et al., "CapelliniSpTRSV: A Thread-Level Synchronization-Free Sparse Triangular Solve on GPUs," in *49th International Conference on Parallel Processing - ICPP*, Edmonton AB Canada: ACM, Aug. 2020, pp. 1–11. https://doi.org/10.1145/3404397.3404400.

[10] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves," in *Euro-Par 2016: Parallel Processing*, P.-F. Dutot and D. Trystram, Eds., Cham: Springer International Publishing, 2016, pp. 617–630. https://doi.org/10.1007/978-3-319-43659-3_45.

[11] Z. Lu, Y. Niu, and W. Liu, "Efficient Block Algorithms for Parallel Sparse Triangular Solve," in *49th International Conference on Parallel Processing - ICPP*, Edmonton AB Canada: ACM, Aug. 2020, pp. 1–11. https://doi.org/10.1145/3404397.3404413.

[12] E. Duffrechou and P. Ezzatti, "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)*, Mar. 2018, pp. 196–203. https://doi.org/10.1109/PDP2018.2018.00034.

[13] C. Xie et al., "Fast and Scalable Sparse Triangular Solver for Multi-GPU Based HPC Architectures," in *50th International Conference on Parallel Processing*, Lemont IL USA: ACM, Aug. 2021, pp. 1–11. https://doi.org/10.1145/3472456.3472478.

[14] T. Chen et al., "(TVM): An Automated End-to-End Optimizing Compiler for Deep Learning," 2018, pp. 578–594. Accessed: Apr. 10, 2025. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[15] A. Ashari et al., "Accelerating Sparse Deep Neural Networks," 2021, *arXiv*. https://doi.org/10.48550/ARXIV.2104.08378.

Supplemental Materials

Table 1: SpMV performance metrics across various matrices for two different implementations

Matrix	**Algorithm**	**Execution Time (ms)**	**Throughput (GFLOPS)**	**Memory Bandwidth (GB/s)**
ML_Laplace	CSR-Vector	0. 287392	192.698	732.518
	cuSPARSE	0.839872	65.9386	250.657
Lp_ken_11	CSR-Vector	0. 0483648	2.02867	11.4655
	cuSPARSE	0.663962	0.147774	0.835175
Freescale1	CSR-Vector	0. 656781	57.6154	272.978
	cuSPARSE	0.748544	50.5524	239.514
Thermal2	CSR-Vector	0.201645	48.6418	249.267
	cuSPARSE	0.597363	16.4194	84.1422
kmnist_norm_10NN	CSR-Vector	0.04912	3.19487	14.1771
	cuSPARSE	0.590438	0.265789	1.17943

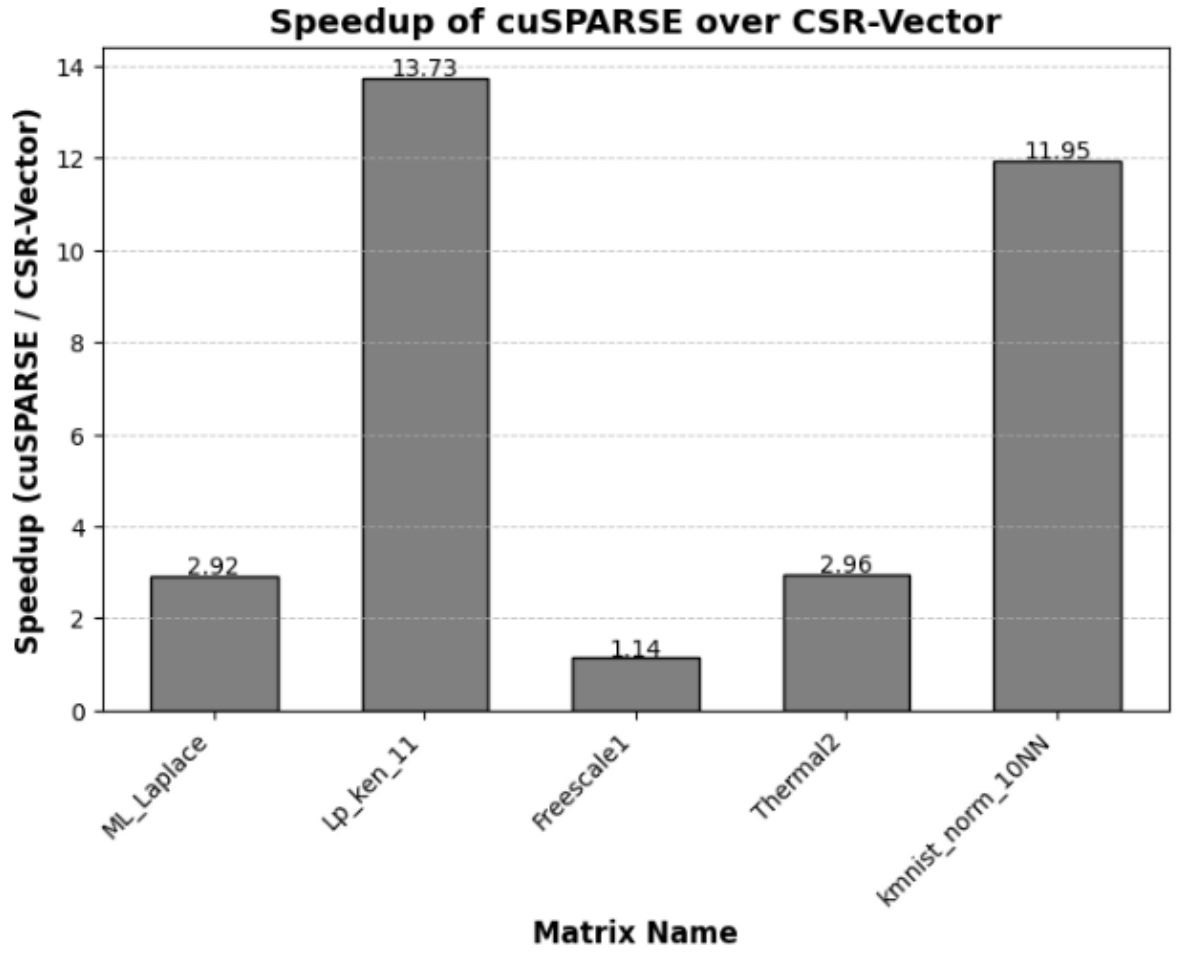

Figure 1: SpMV speedup of CSR-Vector implementation compared to cuSPARSE

Table 2: SpTRSV performance metrics across various matrices for two different implementations

Matrix	**Algorithm**	**Execution Time (ms)**	**Throughput (GFLOPS)**	**Memory Bandwidth (GB/s)**
tmt_sym	Recursive Block	206.654	0.0281034	0.143994
	cuSPARSE	271.451	0.0213949	0.109622
chipcool0	Recursive Block	11.1989	0.0268984	0.120245
	cuSPARSE	0.504499	0.597091	2.66921

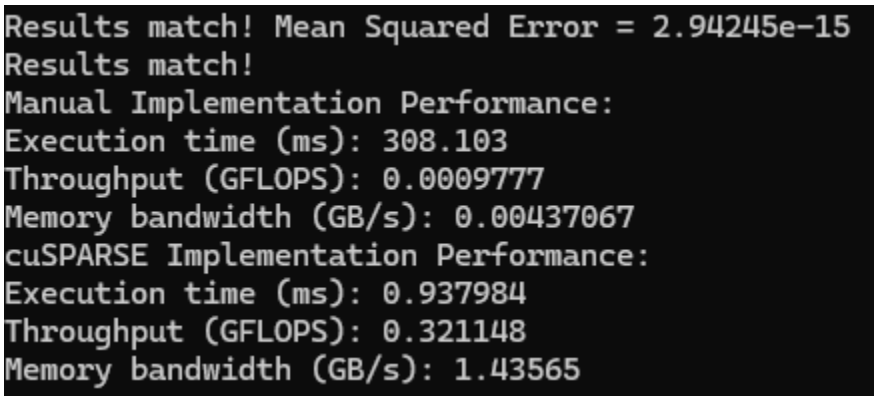

Figure 2A: Unusual results produced by synchronization free (manual) implementation of SpTRSV. Abnormally long times indicate a likely race condition

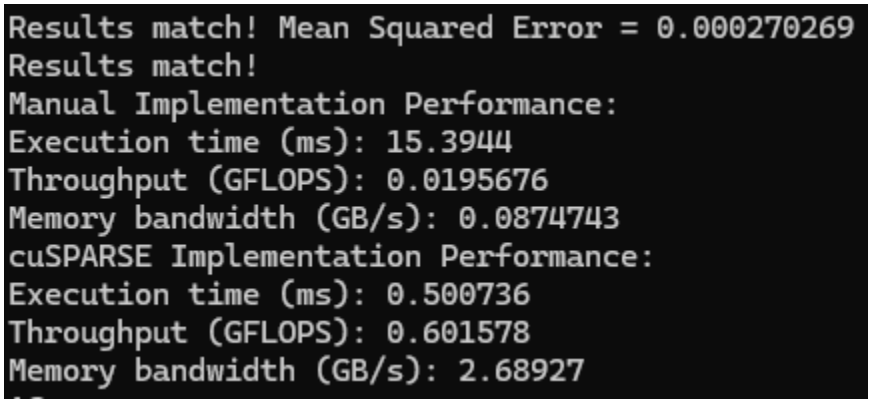

Figure 2B: Unusual results produced by synchronization free (manual) implementation of SpTRSV. Abnormally long times indicate a likely race condition