Sync-Free GPU Parallelization of Sparse Kernels from Sequential Python Code

Malko-Bani Somo
SwiftWare Lab
McMaster University
Hamilton, Canada
somom1@mcmaster.ca

Kazem Cheshmi (advisor)
SwiftWare Lab
McMaster University
Hamilton, Canada
cheshmi@mcmaster.ca

1 Introduction

The demand for fast and scalable matrix kernels is higher than ever, driven by the rapid growth in fields such as artificial intelligence, bioinformatics, and climate modeling. Examples of sparse kernels that are critical in many scientific and engineering applications include matrix-vector multiplication (SpMV), triangular solves (SpTRSV), and the Gauss-Seidel method. Without engineers with a background in low-level programming models such as CUDA, the performance tuning of sparse kernels becomes difficult and time-consuming.

The parallel execution of a matrix kernel must ensure that dependencies of a calculation are resolved before a given operation executes. This ordering requirement is central to maintaining correctness in parallel code generation. Current compiler transformations rely heavily on affine array accesses, where array indices can be statically predicted and dependencies inferred through compile-time analysis [1]. These affine assumptions work well for dense computations, where array indices are regular and predictable, but they break down in sparse contexts.

Sparsity in matrices fundamentally changes the optimization landscape. On one hand, sparsity reduces the number of true dependencies, creating more opportunities for parallelism. On the other hand, sparsity introduces irregular memory access patterns, often through indirection arrays [2]. For example, sparse kernels commonly operate on compressed sparse column (CSC) or compressed sparse row (CSR) formats, where index arrays determine the location of nonzero entries. This indirection leads to non-affine memory accesses, which defy traditional compile-time dependence analysis [1]. As a result, existing compilers tend to conservatively leave sparse code unoptimized, forfeiting significant performance gains that could be achieved if the irregular structure were better understood.

This work proposes a sync-free, runtime-based transformation that automates loop parallelization for sparse kernels with loop-carried dependencies. It focuses on sparse matrix-vector multiplication, sparse triangular solves, and the Gauss-Seidel method for both CSC and CSR matrices in order to develop a framework that can be generalized to other sparse kernels with similar properties.

2 Methodology

The core of this transformation begins with a trace step: the input code is executed once to capture all memory reads and writes, producing corresponding read and write sets. The transformed output code then introduces an array of flags, where each flag signals when a memory location has been fully computed and is safe to use. Structurally, the output kernel consists of three components: a flag checker, the core computation, and a flag setter. The trace information serves as input to these three components.

Several matrix properties influence how efficiently this transformed code executes. If the read set is affine, as in SpTRSV on CSC matrices after simplification, dependencies can be verified in constant time, making the flag checker efficient. Conversely, non-affine read sets, such as those in SpTRSV on CSR matrices, require iteration to confirm all dependencies are satisfied. Similarly, affine write sets (e.g., SpTRSV on CSR) allow constant-time flag updates, while non-affine write sets (e.g., SpTRSV on CSC) require iterating over dependent iterations. Finally, if updates to a memory location use an associative operation, as in SpTRSV, execution order no longer matters. Reads can safely consume partial sums, effectively reducing the number of dependencies. This property is what simplifies the CSC case, turning its read set into an affine one.

After the simplifications are applied, the compiler outputs a Triton kernel in a source-to-source transformation. The generated Triton kernel leverages a sync-free approach to parallelism. Traditional parallelization often relies on synchronization primitives (such as barriers) to ensure dependencies are respected, but these mechanisms introduce performance overhead and can stall threads unnecessarily. In contrast, the sync-free method ensures that threads can proceed as soon as their dependencies have been resolved according to the DAG. This leads to higher utilization of hardware resources and improved scalability on GPUs.

3 Results

All matrices are highly sparse (97–98%) and PSD, with 54k–1.6M nonzeros. CPU sequential solves are extremely slow (0.003 GFLOP/s) due to memory-bound behavior. CuPy GPU solves achieve moderate acceleration (up to 0.135 GFLOP/s), limited by irregular memory access. Triton's sync-free kernel significantly outperforms both CPU and CuPy, giving 1.5–6× speedup over CuPy depending on wavefront length. Speedup is largest for small wavefronts where Triton fully exploits GPU parallelism; for larger wavefronts CuPy

SC '25, November 2025, St. Louis, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

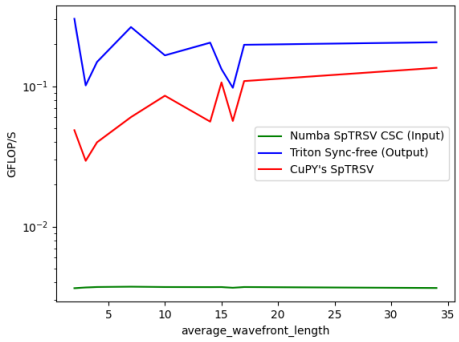

Figure 1: Performance comparison of SpTRSV across implementations.

closes the gap slightly. Overall, Triton demonstrates that wavefront parallelism combined with a sync-free design substantially improves sparse triangular solve throughput on GPUs. Results are summarized in Figure 1.

4 Future Work

Future work will focus on generalizing the identified properties to enable parallelization of ILU0, a widely used preconditioner for iterative solvers. From there, the approach can be extended beyond SpMV, SpTRSV, and Gauss-Seidel to a broader class of sparse kernels. Another direction is integrating the framework into existing compilers such as Numba, making these transformations more accessible to practitioners in Python-based HPC workflows.

5 Conclusion

Ultimately, this line of work contributes toward a future where high-performance sparse computation is both accessible and scalable, enabling continued progress across disciplines that depend on sparse linear algebra.

References

[1] Yaoqing Lin and David Padua. 2000. Compiler Analysis of Irregular Memory Accesses. *SIGPLAN Notices* 35, 5 (2000), 157–168. https://doi.org/10.1145/358438.349322

[2] Mohammad Sadegh Mohammadi, Tomofumi Yuki, Kazem Cheshmi, Eric C. Davis, Mary Hall, Maryam Mehri Dehnavi, Prasanna Nandy, Catherine Olschanowsky, Arvind Venkat, and Michelle Mills Strout. 2019. Sparse Computation Data Dependence Simplification for Efficient Compiler-generated Inspectors. In *PLDI '19: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 594–609. https://doi.org/10.1145/3314221.3314646