FROGBYTE HPC F25

ANNA NORRIS
CARMEN PARK
DAVID HADDAD
STELLA GREENVOSS

December 9, 2025

WITH JEE CHOI

Abstract

This paper evaluates sparse matrix-vector multiplication (SpMV) performance across five storage formats: CSR, COO, ELL, SELL-C, and CSR5. We discuss each kernel and corresponding storage format, as well as their performance with respect to matrix size and sparsity patterns. We then implement each format with architecture-specific optimizations; benchmarking across matrices with varying sparsity patterns. We discuss how we developed our interpretations of each implementation, and what improvements could be made in the future. Our results indicate trade-offs between memory efficiency and computational throughput that inform format selection for target hardware, and help inform when each kernel is the most effective. We additionally propose further work that can be done to further evaluate and improve these kernels.

1 Introduction

Sparse Matrix-Vector (SpMV) multiplication is a fundamental operation in scientific computing and graph analytics. It efficiently computes the product $y = Ax$ of a sparse matrix A and a vector x by performing operations only on the nonzero elements. This significantly reduces the computational cost compared to carrying out useless computations on zeroes. Optimizing SpMV is also multifaceted in that there are numerous SPMV computational kernels – many of which use their own unique storage formats – and routines within larger processes that each have their own parallelization requirements.

Through the course of this project, the FrogByte team conducted a comprehensive examination of both naive and parallel implementations of standard sparse matrix-vector multiplication (SpMV) formats, including ELLPACK (ELL), Coordinate (COO), and Compressed Sparse Row (CSR). Beyond these more conventional storage formats, we further investigated

and implemented advanced hybrid formats such as the Sliced ELLPACK-C-σ (SELL-C-σ) and CSR5 formats, which are both designed to address specific performance bottlenecks on modern GPU architectures. Finally, we began to investigate and profile Vector-COO/CSR. Our research aimed to systematically compare the performance characteristics of each format's kernel implementation across a diverse range of matrix sparsity patterns, including regular, irregular, and highly irregular nonzero distributions, as well as matrices exhibiting both uniform and skewed density profiles.

All benchmarking and comparative analysis were executed on high-performance computing platforms; specifically, the University of Oregon's Talapas cluster and Portland State University's ORCA system, with particular emphasis on evaluating scalability and architectural efficiency across different parallel implementations.

2 Background and Related Work

SpMV is a crucial operation in scientific computing. It can be utilized in many scientific and mathematical disciplines, ranging between linear solvers, eigenvalue problems, graph algorithms, and more. One computational characteristic of SpMV is that it is memory-bound. This often means that irregular memory access patterns cause a significant bottleneck for computation, lowering the overall execution time. Matrix size is thus a significant slowdown factor in these operations. The SpMV problem formulation is $y = Ax$ where A is an $m * n$ sparse matrix and x is a size n vector.

2.1 Traditional SpMV Storage Formats

Traditional sparse matrix storage formats each present distinct trade-offs between structural regularity and flexibility. The Coordinate (COO) format offers maximal flexibility through simple triple storage (row, column, value) but suffers from poor cache locality and inefficient memory access patterns. In contrast, the Compressed Sparse Row (CSR) format — the CPU standard — organizes data by row for better spatial locality but introduces load imbalance challenges on parallel architectures. The ELLPACK (ELL) format enforces structural regularity through uniform row padding to enable coalesced GPU memory access, achieving high performance for uniform matrices but incurring prohibitive padding overhead for irregular patterns.

2.1.1 COO

The Coordinate (COO) format represents the most intuitive sparse matrix storage scheme. It encodes the non-zero numbers, column indices, and row indices in three separate arrays, so there are three arrays of size equal to the number of non-zeros.

2.1.2 CSR

The Compressed Sparse Row (CSR) format has emerged as the de-facto standard for sparse matrix storage in CPU-based scientific computing, organizing non-zero numbers into three arrays: row pointers indicating the start of each row, column indices for each nonzero, and corresponding values. This row-major organization provides efficient sequential access to all elements within a row, making it particularly suitable for iterative solvers and algorithms with row-wise dependencies.

2.1.3 ELL

The ELLPACK (ELL) format addresses GPU memory coalescing requirements by enforcing a regular structure through uniform row padding. In ELL representation, all rows are padded to equal length with data stored in two dense 2D arrays: one for column indices and another for values. This regularity enables perfectly coalesced memory accesses when threads within a warp process corresponding elements across different rows, maximizing memory bandwidth utilization on GPU architectures. However, the row length is determined by calculating the maximum number of nonzero entries in a row in a matrix. Thus, a matrix with a low average number of nonzero entries per row with a significant dense row outlier will waste space and time spent allocating padded rows.

3 GPU Considerations

With the rise of GPU computing, the need for faster SpMV operations has become increasingly critical. This has led to the modification and altering of traditional SpMV formats to better address factors like

- Memory Coalescing,
- Warp Divergence,
- Occupancy Limitations,
- Latency Hiding, and
- Memory Hierarchy.

Uncoalesced memory accesses often degrade bandwidth utilization, and thread under-utilization during memory latency stalls. These limitations have catalyzed research into GPU-optimized sparse formats that impose structural regularity not unlike ELL's uniform padding, or use sophisticated load balancing strategies. This is reshaping how many sparse matrices are stored and processed to better align with GPU execution characteristics. What SELL-C-σ and CSR5 focus on is building upon existing formats, ELL and CSR respectively, and more evenly distributing the work load within the GPU architecture as well as localizing where the work is done so it can be more efficiently computed by each thread block or SIMD lane and fully saturate the GPU's compute units.

This can similarly be seen with the COO and CSR warp level approach, seen in Tsai et al. [12] where each warp processes one or more rows of the sparse matrix. This level of optimization has become a new standard to seek out with parallelized sparse vector matrix operations, and the root of our current exploration.

4 Advanced SpMV for GPUs

4.1 SELL-C-σ

Among the many optimized GPU formats in recent literature, the Sliced ELLPACK-C-σ (SELL-C-σ) format demonstrates and exploits the clear tradeoffs made by many naive formats (specifically CSR and ELL). A pseudo-hybrid model, SELL-C-σ, extends ELL by dividing the matrix into fixed-height slices of size C rows and applying row reordering locally to each slice. This reordering is dependent on σ and attempts to decrease the amount of padding overhead seen in the ELL method on irregular matrices.

ELL solves the memory coalescing problem introduced on the GPU when running CSR. This reordering of data and the benefit we see is even greater for SELL-C-σ. All threads within a warp load memory coalesced. This allowed the SELL-C-σ kernel to perform well across all our initial testing, often outperforming CSR and ELL.

4.1.1 SELL-C-σ Storage Format and Conversion

The SELL-C-σ format improves on ELL by slicing the matrix into blocks of C consecutive rows (presort). The parameter C is chosen to match the size of the warp (in our case $C = 32$). Inside each slice, the rows are sorted by length (nonzero count), and in our implementation, this is maintained locally to avoid disturbing the global row order ($\sigma = C$).

Padding is determined locally within each slice, based on the same ELL metric of the maximum row length. The effect is a drastic reduction in padding overhead compared to a naive ELL implementation, while maintaining the uniform coalesced memory access. The slices are stored in two arrays, one for column indices and one for values. Column-major ordering of these allows for threads to perform fully coalesced memory access throughout the processing of a row.

The CPU-side conversion also introduces overhead, but it is negligible after a certain number of SpMV operations (depending on the matrix size), so it is amortized in the typical use case where hundreds of operations take place on the GPU before transferring back to the host. The CPU performs slicing, nonzero counting in each slice, sorting, and matrix padding. An array for mapping row sorting is recorded to allow recovery of the original matrix row ordering after GPU computation, if needed.

4.1.2 SELL-C-σ Kernel

The SELL-C-σ kernel launches a warp per row-slice (larger workload than per row). Since C is set to the size of the warp, each thread within the warp is assigned to a specific row within the slice. All the rows are the same length, allowing identical loop bounds and eliminating branch divergence on the GPU. As stated before, this allows memory transactions to be coalesced.

The execution of the kernel involves iterating over the padded slice, loading a column index and value each iteration. The vector entry corresponding to the column is read, multiplied, and accumulated. Theoretically, the kernel maximizes memory throughput with this structure, as all threads have identical memory access patterns. After the slice has computed its values, the threads write their accumulated values to the output vector (reducing the number of atomic operations). The CPU then unscrambles the row ordering.

4.2 CSR5

CSR5 (Compressed Sparse Row version 5) is a tile-based sparse matrix format introduced by Liu and Vinter in 2015 [8] designed explicitly for efficient SpMV operations across diverse parallel architectures, including both CPUs and GPUs. Unlike traditional formats that optimize for either regularity or flexibility, CSR5 employs a balanced approach that partitions the matrix into uniform tiles while maintaining the computational simplicity of CSR through auxiliary data structures for load balancing. It transposes traditional CSR value and column arrays into $s\omega x\sigma$ sized tiles to help with contiguous memory access and to break up work into warp level chunks. There are a few additional elements that are added as a result to also speed up processing.

4.2.1 CSR5 Storage Format

The CSR5 format organizes sparse matrices through several key data structures that enable both efficient storage and parallel computation. For example, the tile size ($\sigma \times \omega$) is hardware dependent, where ω (tile width) is primarily determined by hardware characteristics such as warp size, while σ (tile height) depends on both hardware constraints and the sparsity pattern of the matrix.

To clearly process each tile, two additional peices of information, a tile pointer, and a tile descriptor, are created to speed up tile processing. The tile pointer, which is the length of the total numbers of tiles, indicates the row number of the first element in each tile.

The tile descriptor is composed of 4 parts: `bit_flag`, `y_offset`, `seg_offset`, and `empty_offset`. Each plays a key role in assuring accurate results and optimizing parallelization.

- `bit_flag`: The `bit_flag` is an indicator of the start of a new row. This covers the size of `num_tiles` $\times\omega \times \sigma$ but often with at least half the space of double precision floats. We experimented with storing `bit_flag` in `uint32_t` as well as `uint8_t` tiles.
- `y_offset`: The `y_offset` can be obtained by counting the number of true flags in the columns preceding it. This array is size ω for every tile, and is helpful in indicating where partial sums can best be stored.
- `seg_offset`: The segmentation offset, which is also size ω for each tile, indicates how many tile columns to its right are within the same row in the sparse matrix. The purpose of this allows the kernel to take each tile column and break its product into a set of prefix sum operations. These partial sums across each tile contribute to further optimizations in performance.
- `empty_offset`: The `empty_offset` array is the length of the number of rows in a tile, and is only needed if there exists an empty row in the tile. This array is usually the smallest of the descriptors, but crucial for accurate results on CSR5.

The CSR5 SpMV kernel employs a two-phase execution strategy optimized for parallel architectures:

Phase 1: Tile Processing

- Each thread block or SIMD lane processes one or more complete tiles
- Threads within a warp cooperate on tile computation using warp-level primitives
- The bit-flag array directs threads to valid elements, avoiding computation on padding
- Partial results are accumulated using efficient warp-level reduction operations

Phase 2: Row Accumulation

- Partial results from tiles contributing to the same output row are combined
- Atomic operations or segmented reduction techniques merge contributions
- Row versioning information ensures correct mapping to output vector y

Key Optimizations:

1. **Warp-Centric Execution**: Each warp processes entire tiles, maintaining SIMD efficiency
2. **Bit-Level Masking**: The bit-flag array acts as a map to help detect when
3. **Memory Coalescing**: Tile organization ensures contiguous memory access patterns
4. **Load Balancing**: The σ parameter controls tile grouping to balance workloads across warps

5 methodology

Our computational approach employs multiple sparse matrix storage formats to analyze their suitability for specific algorithmic patterns. In the following section we discuss the implementation of the five key representations from above, CSR, COO , ELL, SELL-C, and CSR5. This evaluation is conducted through systematic benchmarking on diverse sparse matrices, measuring both memory footprint and computational throughput.

To ensure correctness across different architectures, these kernels were profiled on two different HPC clusters: Talapas, the University of Oregon's cluster, and ORCA, PSU's cluster. On Talapas, we used a single NVIDIA A100 GPU (Ampere architecture) with 40 GB HBM2 memory, 1.6 TB/s memory bandwidth, and 9.7 TFLOPS double-precision peak performance. On ORCA, we used an L40S, which uses NVIDIA Ada Lovelace Architecture, has 40GB GDDR6 memory, and 864 GB/s memory bandwidth. For each metric and comparison, we specify which cluster it was collected on. We did not include multi-node functionality in order to control the scope of our project.

5.1 COO

There are a number of ways to implement COO SpMV in GPU programming. In this instance, we attempted to implement three versions: scalar COO, vector COO, and another more complex load-balancing vector COO. In the case of scalar COO programmed in CUDA, it resembles the sequential version the most. In this case, a thread is assigned to an index of the arrays and performs atomic addition to add the product to the result vector. We find the number of blocks by dividing the size of the rows by the number of threads in a block. This implementation has a number of obvious drawbacks, including poor memory locality and the frequent use of atomic addition leads to significant performance overhead.

The vector COO kernel we implemented assigns one warp of 32 threads to a chunk of COO entries. Then, it iterates in a grid-stride loop and after loading data, performs warp-level reduction to find the partial-sum of the warp. Then, the last thread in the segment uses `atomicAdd` to add the value to the result vector. This kernel improves on the format of the scalar COO kernel by reducing the number of `atomicAdd` from one per COO entry to one per warp. However, the issue of poor memory locality remains, and this version of vector COO SpMV results in a load imbalance between warps so performance is worsened.

We were unable to achieve accurate results with our implementation of a third load-balancing COO kernel, this will be revisited in future work. We followed the algorithm given in the Tsai Y., et al. paper [12]. One significant detail for this kernel is that it expects sorted COO arrays by row then column. This is because it assumes the arrays are sorted to compute whether the warp will encounter the next row. If it does, then it performs a segmented scan to collect the partial-sum of the current row, and the first thread of the segment performs the atomic addition. One deviation in our implementation is that the last thread in the segment performs the atomic addition and adjusted the segmented scan accordingly, but again, due to inaccurate results with our implementation it was not included when measuring performance.

Algorithm 1 Load-balancing COO kernel algorithm

```
1: ind ← index of the first element to be processed by this thread
2: current_row ← rowidx[ind]
3: c ← A[ind] × x[colidx[ind]]
4: for i = 0 to nz_per_warp step warpsize do
5:     next_row ← rowidx[next ind]
6:     if any thread has next_row ≠ current_row or final iteration then
7:         Perform segmented scan for the current row
8:         if this thread is first in its segment then
9:             atomicAdd(output[current_row], c)
10:        end if
11:        c ← 0
12:    end if
13:    ind ← next ind
14:    c ← c + A[ind] × x[colidx[ind]]
15:    current_row ← next_row
16: end for
```

5.2 CSR

Similarly to COO, we implemented scalar and vector versions of CSR SpMV. Once again, the scalar version is the easiest to implement because of its similarities to the serial version of CSR. In this case, one thread is assigned to one row in the CSR array and the sum of the product of all values in the row are

added to the result vector by the thread. We divide the number of rows by the number of threads per block to find the number of blocks needed. This version has poor memory coalescence because threads will access non-contiguous entries and can have significant load-imbalance with irregular matrices.

In order to improve on scalar CSR, we created a vector CSR kernel as well. In this case, a warp of 32 threads is assigned to a single row in the matrix and this computes the product of that row. Then, warp reduction is performed to create the sum that is written to the output vector. This version will more effectively utilize memory locality because the threads in the warp are sequentially accessing the same segments of memory. The following pseudo code as provided in the Tsai Y., et al. paper is what we used for our implementation.

Algorithm 2 Vector CSR kernel

```
1: row ← row index for this thread
2: subrow ← stride to next row
3: step_size ← stride to next element
4: c ← 0
5: for row = row to #rows step subrow do
6:     for idx = rowptr[row] to rowptr[row+1] step step_size do
7:         c ← c + val[idx] × b[col[idx]]
8:     end for
9:     Perform warp reduction on c
10:    if thread 0 in subwarp then
11:        Write c to output[row]
12:    end if
13: end for
```

5.3 CSR5

We utilized the already constructed CSR arrays to create CSR5 for this project. The git repository connected to Liu's original work [8] seemed to pull CSR5 directly from a matrix file which could be an interesting attempt in the future.

To determine the optimal σ bounds $\langle$r, s, t, u$\rangle$ for a given GPU, you can perform hardware benchmarks using a set of fabricated sparse matrices with controlled uniform row densities. The process will show CSR5 SpMV performance across a wide range of σ values for each matrix density. You identify r as the point where too-small of a σ degrades performance for sparse rows. Similarly, s is the optimal σ for moderate densities; and t becomes the threshold where a larger σ value no longer is beneficial. U becomes the most optimal and often small σ for extremely dense rows. These hardware-dependent values are fixed once established through this initialization benchmark, and there is no need for parameter tuning for all subsequent SpMV executions.

In implementing CSR5, creating the tile pointer was relatively straightforward, while there were a lot of spaces for interpretation of how to gather together the tile descriptor. As far as memory allocation, writing the value and column arrays to 2D tiles led to the use of `CudaMallocPitch` for tiling, and copying the row pointer was similar to homework 3 and 4 in that the memory from the CPU was copied to global GPU memory.

First implementations of gathering the tile descriptor and the tile pointer took place serially on the CPU. For the tile pointer this meant incrementing through the tiles and finding the initial row index.

For the tile descriptor, naive attempts also stayed on serial on the CPU. This involved using the row pointer to write a comprehensive `bit_flag` array, then using its dimensions to build each tiles `y_offset` array and `seg_offset` array. Constructing the `empty_array` proved to be the most difficult. It was also initially allocated in C++, but each array's length varied based off of the number of rows in the empty tile.

There is pseudo code in Liu et al [8] that finds tile descriptor elements on the GPU. Looking more closely at this implementation could be promising for a faster and more efficient collection of each descriptor variable.

5.4 ELL

The ELLPACK (ELL) format restructures the sparse matrix into two dense `num_rows` $\times L$ arrays, where L is the maximum row length (i.e., the maximum number of non zeros in any row). Each row is padded with zeros so that all rows share the same length, enabling a regular memory access pattern across threads. GPU threads within a warp iterate over the same column index at each step, producing perfectly coalesced memory accesses and minimizing divergence.

However, the primary drawback of ELL is its sensitivity to irregular sparsity. If a matrix contains even a small number of long rows, the padding overhead increases substantially, forcing the GPU to load values that correspond to structural zeros. This incurs wasted bandwidth and unnecessary computation. In

practice, ELL performs well for highly structured matrices with narrow row-length distributions but degrades quickly for matrices with large variance in row densities.

ELL serves as a conceptual predecessor to SELL-C-σ, which attempts to retain ELL's benefits while addressing its pathological sensitivity to outlier rows [1].

5.5 SELL-C-σ

The SELL-C-σ format extends ELLPACK by partitioning the matrix into fixed-height slices of C rows, where C is chosen to match the GPU warp size ($C = 32$ on NVIDIA architectures). Within each slice, rows are sorted by decreasing nonzero count (controlled by the parameter σ), which reduces local padding overhead while preserving the global structure of the matrix. This design balances ELL's regular memory access pattern with CSR's flexibility, yielding a hybrid format well aligned with GPU execution characteristics [1].

During preprocessing, each slice's longest row determines the padded row length for that slice alone—unlike ELL, which pads according to the single global maximum. This significantly reduces wasted storage and memory traffic for matrices with moderate irregularity. The values and column indices inside each slice are then stored in column-major order to ensure that threads in a warp read contiguous memory locations at each iteration.

The SELL-C-σ kernel launches one warp per slice, with each thread assigned to a single row. Because all rows within a slice share identical padded lengths, the warp executes in lockstep without branch divergence. Each iteration of the kernel performs a multiply–add using coalesced memory transactions, leading to high effective memory bandwidth. After slice-level accumulation, the results are optionally reordered to match the original row layout.

As demonstrated by Anzt et al. [1], this structure provides substantial performance improvements for matrices with controlled irregularity and approaches the throughput of memory-bound kernels on modern GPUs. In our implementation, SELL-C-σ consistently outperformed CSR and ELL on the Talapas A100 for structured matrices, and remained competitive even on those with moderate row-length variance.

Results

6 Benchmarking and Evaluation Methodology

To benchmark, we attempted to use cusparse, the CUDA library for sparse matrix operations, as a baseline for comparison. However, at our smaller matrix sizes, cusparse's execution times were significantly worse than all of our kernels, making it a poor baseline. We hypothesize that this could be due to cusparse being optimal for much larger matrices, and perhaps therefore having higher launch/configuration latency for these smaller ones that gets amortized while doing larger operations. As such, we instead chose to cross-compare between our kernels to find the tradeoffs of each.

For initial timing, we used CUDA events, which allowed us to measure the time it took for 100 executions of a kernel. We then divided this result by the number of executions (100), achieving the average execution time. For further analysis, we used Nvidia Nsight's profiler nci, obtaining the Speed Of Light metrics (described as the "High-level overview of the throughput for compute and memory resources of the GPU") [9]. This allowed us to obtain more accurate results for each kernel.

Most of the Nsight-based profiling occurred on ORCA's L40S.

7 Results

Our profiling began with a set of five matrices, described in table 1. To debug and iterate without using too many computing resources, we chose a set of relatively small matrices.

Figures 2a and 2b show the minimum execution time across 100 runs for each kernel. Figure 2a shows the data gathered on Talapas, while Figure 2b shows the data gathered on ORCA. Note that the CSR5 figure is a measure of the sub-optimal CSR5 implementation. It can clearly be observed that the patterns are generally consistent across both clusters, although interesting patterns emerge. The smallest execution time on ORCA was 6.1 microseconds, while on Talapas it was 3.2. However, the largest execution time on ORCA was around six times faster than on Talapas. This can perhaps be explained by the trivial size of these test matrices; the slower execution time for the tiny matrices on ORCA could show that the kernel launch latency is higher on the L40S than on the A100.

In terms of patterns between the different kernels, we see that on the L40S, CSR5 is the clear winner, being faster than every other kernel on every input. In contrast, SELL-C seems to outperform its competitors on the A100.

Across the benchmarks, SELL-C-σ demonstrated consistently strong performance on the matrices with a more regular sparsity structure. On the A100 profiling, SELL-C-σ achieved the highest effective bandwidths observed in our testing. The most consistent pattern was SELL-C-σ outperforming CSR and ELL, which comes from the intentional additions to this algorithm (slicing sorting). On the `cantilever` matrix, the kernel approached memory bandwidth saturation. Performance degraded, however, on more irregular matrices like `bcsstk`. The decrease in performance can be explained by the fact that local padding still wastes significant memory, undermining one of our primary optimizations. One notable observation is that, on the smaller matrices we tested, the conversion overhead is relatively small compared to kernel execution time for repeated multiplications, increasing this method's appeal for iterative solvers. These amortized benefits, despite not fully optimizing the preprocessing, demonstrate the strength of this algorithm.

We also compared the performance of the scalar/vector CSR/COO kernels on the Cantilever matrix using Nvidia Nsight and achieved some strange results. As can be seen in figure 3a, the scalar coo kernel was by far the fastest, despite being, in theory, the least optimized kernel. The results of the low optimization can be seen in its relatively low achieved memory throughput and occupancy. How, then, does this suboptimal kernel achieve incredible runtimes? This is as of yet unclear; we hypothesize it could be due to mis-selecting block sizes for the other kernels, improper collection of data, or a real result of the way that the kernel runs. This should be evaluated by using other methods to time the kernels and testing using variant matrices.

Despite the relatively high runtime, the memory/compute throughput and achieved occupancy of the vector kernels is promising, and seems to bode well for future iterations of these kernels. More work is needed to identify the exact source of this discrepancy, but there may be some memory mismanagement within the kernels that produces more work than necessary and so lengthens the runtime. However, the higher bandwidth points to an obvious potential for speedup compared to their scalar counterparts. Additionally, more tests are needed on different matrices to understand how COO and CSR kernels compute them.

8 Conclusion and Reflection

While both the SELL-C-σ and CSR5 formats have demonstrated significant performance improvements over traditional sparse storage formats in their respective publications, there exists a notable gap in the literature regarding direct, systematic comparisons between these two state-of-the-art approaches. Most existing studies evaluate either SELL-C-σ or CSR5 against legacy formats like CSR, ELL, or HYB, but few provide head-to-head comparisons using consistent experimental frameworks and implementation baselines.

This absence of comparative analysis is compounded by the lack of unified code bases that implement both formats with equivalent optimization effort, making fair performance evaluation challenging. Furthermore, current research offers limited insight into practical format selection heuristics. While both formats introduce tunable parameters like C and σ for SELL-C-σ, as well as ω and σ for CSR5. These comprehensive guidelines for parameter optimization across diverse matrix characteristics remain underdeveloped. This left us as interpreters without clear guidance for implementation decisions.

Despite having somewhat confusing or inconsistent results while profiling vector-based COO and CSR kernels, our experiments nonetheless show that there is substantial untapped performance potential in optimizing even the most basic implementations of these classical formats. This observation is consistent with prior research, which repeatedly demonstrates that thoughtful warp-level cooperation, load-balancing, and memory-traffic reduction can transform otherwise naive kernels into highly competitive baselines. This reinforces the importance of methodological rigor and careful design when comparing more advanced formats.

The community lacks robust automated tuning methodologies specifically designed for these "hybrid" formats, particularly for handling the increasingly irregular sparse matrices prevalent in modern scientific applications. Finally, benchmark suites often emphasize regular or moderately irregular matrices, leaving a need for more extensive evaluation on highly irregular and extreme-scale sparse patterns that better represent real-world computational chal-

lenges.

References

[1] Anzt, H., Tomov, S., & Dongarra, J. (2014, November). Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C- formats on NVIDIA GPUs. *In 2014 IEEE International Conference on Cluster Computing (CLUSTER)* (pp. 61–66). https://library.eecs.utk.edu/files/ut-eecs-14-727.pdf

[2] Bell, N. & Garland, M. (2008). Efficient Sparse Matrix-Vector Multiplication on CUDA. *Nvidia Technical Report NVR-2008-004* Vol. 2. No. 5.

[3] Benatia, A., Ji, W., Wang, Y., & Shi, F. (2018). BestSF: A Sparse Meta-Format for Optimizing SpMV on GPU. *ACM Transactions on Architecture and Code Optimization*, 15(3), 29. https://doi.org/10.1145/3226228

[4] Flegar, G., & Anzt, H. (2017). Overcoming Load Imbalance for Irregular Sparse Matrices. *In Proceedings of the International Conference on Supercomputing (ICS)*, Chicago, IL, USA. https://publikationen.bibliothek.kit.edu/1000079215

[5] Gao, J., Wang, Y., & Wang, J. (2016). A novel multi–graphics processing unit parallel optimization framework for the sparse matrix-vector multiplication. *Concurrency and Computation: Practice and Experience*, 29(17), e3936. https://doi.org/10.1002/cpe.3936

[6] Kolodziej, S. P., Aznaveh, M., Bullock, M., David, J., Davis, T. A., Henderson, M., Hu, Y. & Sandstrom, R. (2019). The SuiteSparse Matrix Collection Website Interface. *Journal of Open Source Software* Vol. 4. No. 35. pages 1244.

[7] Li, K., Yang, W., & Li, K. (2015). Performance Analysis and Optimization for SpMV on GPU Using Probabilistic Modeling. *IEEE Transactions on Parallel and Distributed Systems*, 26(1), 196-205. https://doi.org/10.1109/TPDS.2014.2308221

[8] Liu, W., Vinter, B. (2015). CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. *arXiv preprint arXiv:1503.05032v2 [cs.MS]*. https://arxiv.org/pdf/1503.05032v2

[9] NVIDIA Corporation. (2023). *Nsight Compute Profiling Guide*. Retrieved from https://docs.nvidia.com/nsight

[10] NVIDIA. (2020). *NVIDIA A100 Tensor Core GPU Datasheet*. Santa Clara, CA: NVIDIA Corporation. https://www.nvidia.com/en-us/data-center/l40s/

[11] Portland State University Research Computing. (2024). *Orca High Performance Computing Cluster Documentation*. Describes compute nodes with 6× L40S GPU nodes (24 total GPUs) and 19× A30 GPU nodes (76 total GPUs), each with AMD EPYC 9534 CPUs and 576 GB RAM. Retrieved from https://orca.pdx.edu/about/compute-resources/

[12] Tsai, Y. M., Cojean, T., & Anzt, H. (2021). Sparse Linear Algebra on AMD and NVIDIA GPUs – The Race Is On. *In Proceedings of the International Conference on High Performance Computing (ISC)*, Virtual Event. https://doi.org/10.1007/978-3-030-78713-4_11

[13] University of Oregon Research Computing. (2024). *Talapas GPU-Enabled Compute Nodes*. Documentation for NVIDIA A100 GPU nodes on the Talapas HPC cluster. Retrieved from https://racs.uoregon.edu/services

Table 1: The matrices used for testing

Matrix	Dimensions	NNZ	Properties
$\text{lns}_1 31$	131 x 131	536	unsymmetric
plsk1919	1919 x 1919	4831	skew symmetric
e40r5000	11948 x 11948	80519	symmetric positive definite
cant	62451 x 62451	4007383	symmetric

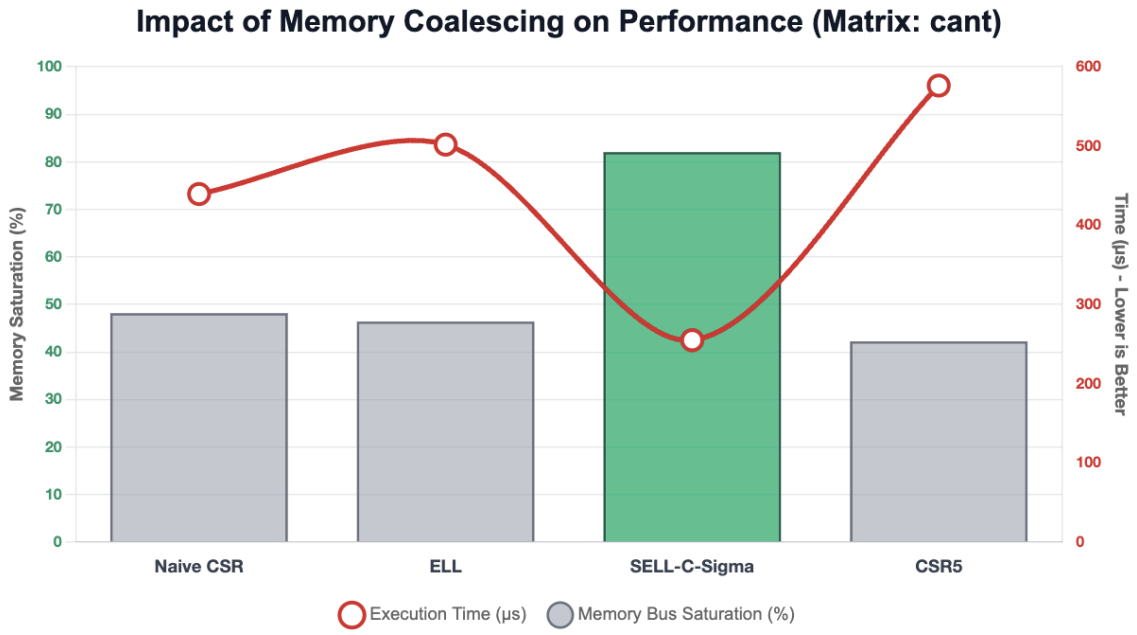
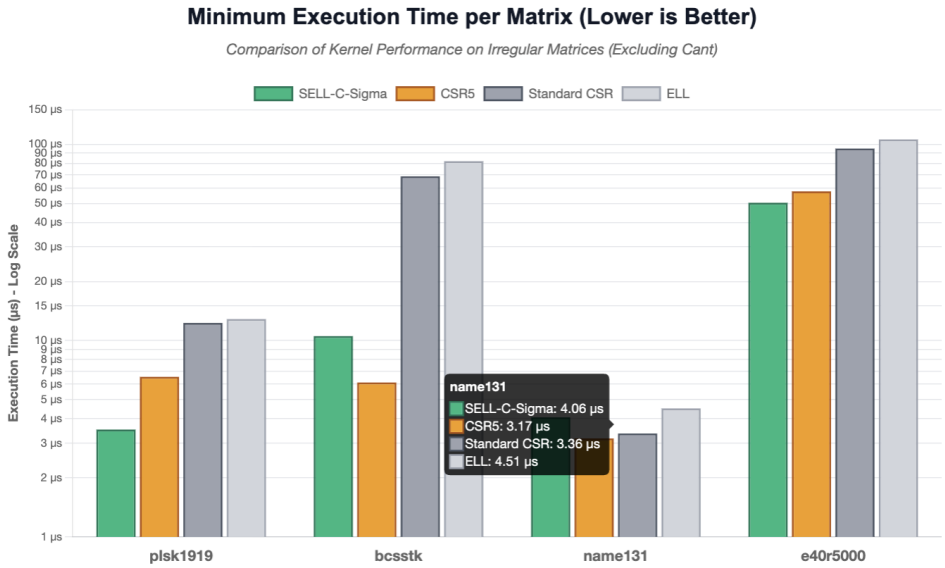
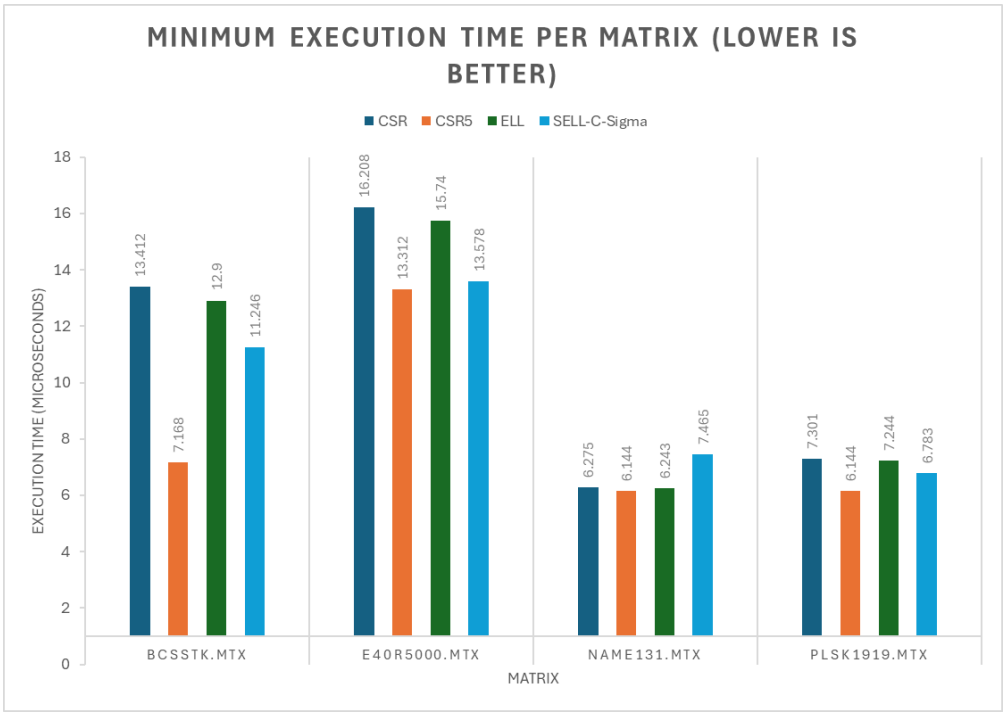


Figure 1: Impact of Memory Coalescence on SpMV Times on Talapas

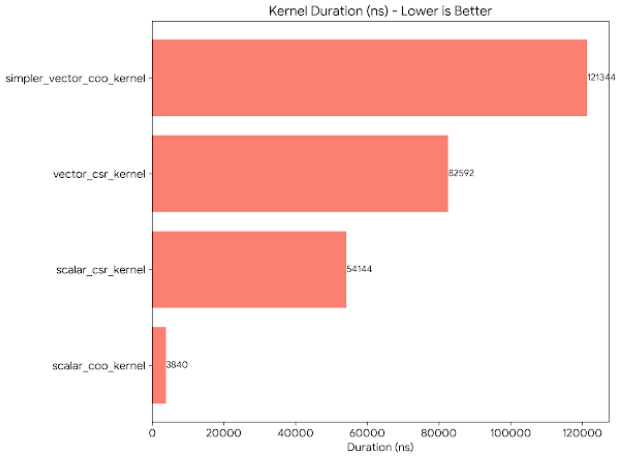


(a) Minimum Execution on Talapas

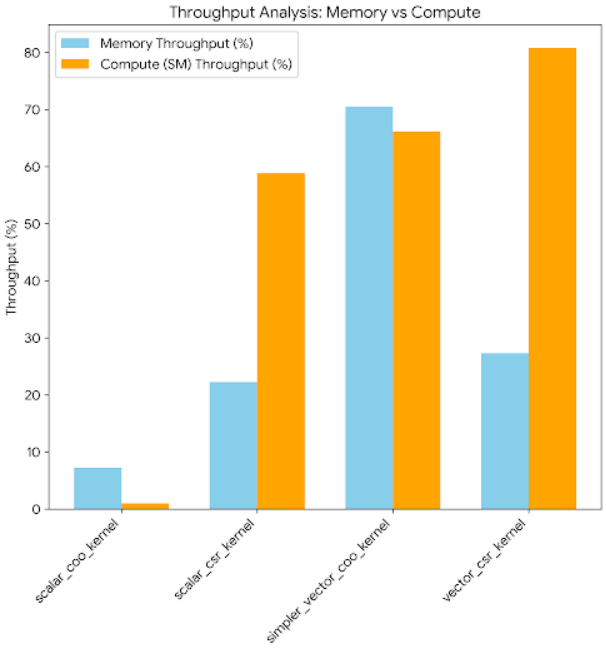


(b) Minimum Execution on ORCA

Figure 2: Execution Times on Talapas and ORCA



(a) Duration of COO and CSR Kernels



(b) Memory throughput of COO and CSR Kernels

Figure 3: Duration and memory throughput of COO and CSR Kernels

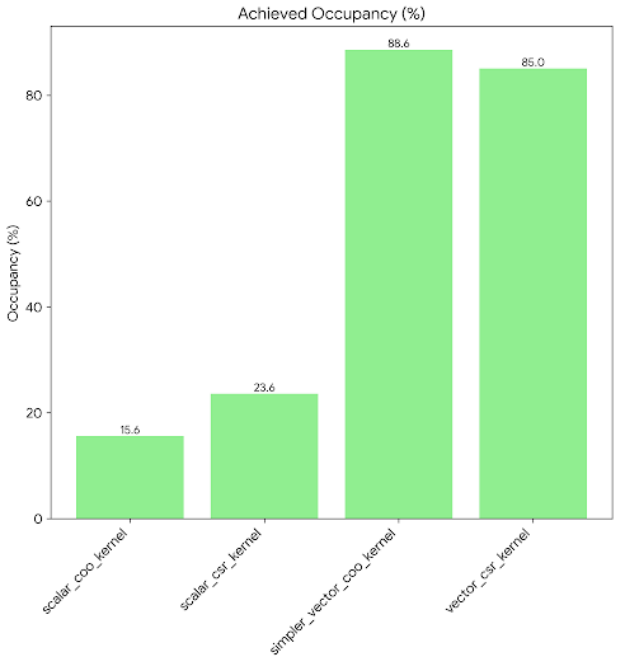


Figure 4: Maximum Occupancy achieved by COO and CSR Kernels