# CUDA Optimization for Sparse Matrix-Matrix Multiplication

**Zepeng Zhao (zepengz)**
Carnegie Mellon University
Pittsburgh, PA 15213
zepengz@andrew.cmu.edu

**Mingxin Li (mingxl)**
Carnegie Mellon University
Pittsburgh, PA 15213
mingxl@andrew.cmu.edu

## Abstract

Sparse matrices are widely used in many computer science applications such as training deep learning models and simulations. It is memory and computationally efficient, but also introduces challenges in parallelization due to the randomness in data access. In this project, we explored sparse matrix multiplication (SpMM) with another dense matrix using 4 different storage formats, analyzed their performance, and compared to existing toolkits. We achieved more than 20x total speedup and average of more than 100x computation speed compared to sequential implementation on large matrices. Our implementation also achieved comparable performance if not better compared to existing libraries, demonstrating the potential and effectiveness of CUDA implementations of sparse matrix multiplication.

Our code is available in this github repository: https://github.com/mli43/Cuda-Optimization-for-SpMM.

## 1 Background

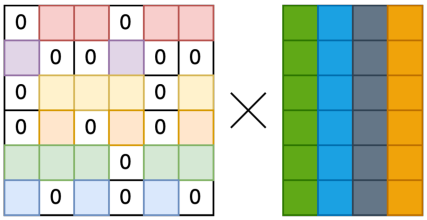

Figure 1: Sparse matrix-matrix multiplication (left matrix is sparse and right one is dense)

Sparse matrix is commonly used in many computer science applications, and sparse matrix multiplication (SpMM) is one of the most commonly used operations on sparse matrices. The objective is to multiply a sparse matrix by a dense matrix to produce a resulting dense matrix. This can be expressed as the formula below. The result matrix C should be a colmoun-major dense matrix. *op* refers to transpose operations. For simplicity, we don't perform any transposition operation. SpMM is much more complex than dense matrix multiplication for several reasons. One one hand, there are many different storage formats for sparse matrices, which means that parallel kernels need to be tailored for specific formats. CUDA kernel implementations for different formats will have different naive algorithms as well as different optimization strategies. On the other hand, because sparse matrices store only the non-zero elements, multiplying sparse matrices result in highly irregular memory access. It is therefore very difficult to exploit locality for parallelization.

$$\text{DenseC} = \text{op}(\text{SparseA}) \,@\, \text{op}(\text{DenseB})$$

Github repository: https://github.com/mli43/Cuda-Optimization-for-SpMM

Although difficult, matrix multiplication is inherently parallelizable, and it applies to sparse matrix multiplication. Similar to dense matrix multiplication, the computation for each non-zero element is the same and does not depend on the computation of others. This is the classic case of data parallelism which allows threads to perform the same operations on different data. SpMM also involves less data overlap because only non-zero elements are processed. Additionally, although locality can't be exploited for the sparse matrix, there is still access locality for the dense matrix. For example, every row in the sparse matrix needs to be multiplied by every column of the dense matrix, which means that all non-zero elements in the sparse matrix will access every column of the dense matrix. This means that the parallel program can process a single column of the dense matrix first before moving on to the next, therefore maximizing data reuse.

There are many storage formats of sparse matrices. The most common forms are Coordinate format (COO) and Compressed Sparse Row (CSR) format. COO is one of the sparse matrix formats that is easiest to understand, and CSR format is widely used in many applications due to its memory and computation efficiency. In addition, we explored two other less common formats, including ELLPACK (ELL) and Block Sparse Row (BSR) formats. The details of each format is explained in the corresponding implementation section.

## 2 Approach

First, we chose to use CUDA toolkit for C++ to parallelize SpMM because the GPU can process thousands of threads concurrently. Its capability for massive parallelism is perfectly suited for SpMM by distributing non-zero elements to different threads. Additionally, GPUs also have high memory bandwidth and can easily handle very large matrices.

We have two sets of test cases for different testing purposes. We browsed Suite Sparse Matrix Collection[2] and downloaded 12 sets of sparse and dense matrices, which are separated into small, medium, and large subsets based on the size of the sparse matrix. At least one set of matrices in each subset is a non-square matrix. These matrices are used to test the correctness as well as the performance of the kernel implementations. The basic information for those matrices are displayed in the table below. After collecting the matrices, we realized that although these test cases cover a good range of matrix sizes, they do not have a good variety of different densities. Therefore, we also developed a python utility script that can randomly generate matrices of fixed size and with different density. We generated 9 sets of matrices with different matrix density that have the same size of 2048x2048, each with density of 0.1-0.9. All test matrices can be converted to the four different formats that we will implement kernels for using a python utility script. (*Note: Matrix ID is the ID of the matrix in the Suite Sparse Matrix Collection[2])

Table 1: Sparse and Dense Matrix Properties by Subset

| Subset | Sparse Matrix Size | Sparse Matrix ID | Num Non-zeros | Density | Dense Matrix Size | Dense Matrix ID |
|---|---|---|---|---|---|---|
| Small | 10x10 | 1524 | 90 | 0.9 | 10x10 | 238 |
| | 32x32 | 1199 | 98 | 0.096 | 32x32 | 168 |
| | 120x210 | 2094 | 840 | 0.083 | 210x21 | 2120 |
| Medium | 1484x1484 | 1626 | 6110 | 0.00074 | 1484x1484 | 2892 |
| | 2048x2048 | 309 | 10114 | 0.0024 | 2048x2048 | 331 |
| | 2880x2880 | 545 | 19635 | 0.0024 | 2880x2880 | 546 |
| | 4000x4000 | 1640 | 8784 | 0.00055 | 4000x4000 | 2827 |
| | 2372x4096 | 2004 | 933343 | 0.096 | 4096x4096 | 2467 |
| Large | 5040x15120 | 2026 | 30240 | 0.00040 | 15120x12600 | 2025 |
| | 20000x20000 | 2828 | 137736 | 0.00034 | 20000x20000 | 2213 |
| | 2798x21074 | 1998 | 81671 | 0.0014 | 21074x105054 | 1997 |
| | 6300x25605 | 2122 | 88200 | 0.00055 | 25605x69235 | 2121 |

The details of the implementation for each kernel are described in the Implementation section. In summary, for each storage format, we first implemented a simple sequential matrix multiplication program on CPU to be used as baseline for performance analysis. We then implemented and tested our kernels using the above test cases, then compared their performance against the sequential program as well as cuSparse[3] for supported formats. Our analysis results are detailed in the section 4.

## 3 Implementation and Format-specific Results

We first used the python utility script to convert our test cases into the following storage formats. Then we implemented sequential programs as well as kernels for each format. The details of the sequential and parallel implementation are described below. For the testing, we ran the sequential and parallel programs, and for formats supported by cuSparse, we also ran the cuSparse implementation for comparison.

### 3.1 COO Format

The COOrdinate format is a simple way to store sparse matrices by storing only non-zero elements and their location. It is one of the most easily understood sparse matrix formats and therefore commonly used. It uses three separate arrays to store the row indices, column indices, and values for each non-zero element. The following is an example of a 5x4 matrix represented in COO format. However, because the number of elements in each row is different, accessing elements in a particular row requires traversing through the entire row array, which makes it less efficient than formats such as CSR.

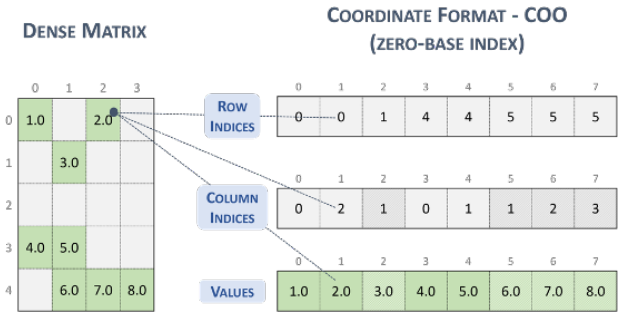


Figure 2: COO Format[3]

The sequential program for COO matrix multiplication roughly follows the following steps:

1. For each non-zero element, find its row i, col k, and value A_ik, and find the kth row of matrix B.
2. For each element in the kth row in B, calculate the product of A_ik with B_kj for each column j in B.
3. Accumulate the values into the result C_ij, and repeat the process for all non-zero elements in A.

This sequential algorithm takes advantage of the COO format's simplicity for iterating through non-zero elements but requires multiple reuse of B elements.

#### 3.1.1 Basic Kernel

The simplest parallel kernel that we designed takes advantage of the simplicity of the storage of non-zero elements. This basic kernel maps each thread to process one non-zero element at location (i, k). Each thread will then access the required kth row of the B matrix and calculate the results of all column values of the ith row of the resulting matrix. Since the stored arrays of COO format are 1-D, the blocks are also 1D and with size of 1024. We then tested the performance of this kernel and compared it with the sequential program. The performance result is displayed in figure 3.

We also compared the performance of COO implementation with the cuSparse implementation. The runtime results are displayed in the graph below. This figure shows that our implementation for COO performs comparable to cuSparse implementation for small and medium test cases, but performs significantly worse for larger test cases.

| testCase | format | cudaKernelTimeMs | cudaTotalTimeMs | sequentialTimeMs | computation_speedup | speedup |
|---|---|---|---|---|---|---|
| small_210 | COO | 0.442 | 0.464 | 0.128 | 0.289593 | 0.275862 |
| small_32x32 | COO | 0.433 | 0.451 | 0.023 | 0.053118 | 0.050998 |
| small_10x10 | COO | 0.435 | 0.455 | 0.007 | 0.016092 | 0.015385 |
| medium_1484 | COO | 1.585 | 5.186 | 68.813 | 43.415142 | 13.268993 |
| medium_2048 | COO | 2.527 | 8.493 | 145.911 | 57.740799 | 17.180148 |
| medium_2880 | COO | 4.161 | 15.394 | 401.393 | 96.465513 | 26.074639 |
| medium_4000 | COO | 6.142 | 25.213 | 248.580 | 40.472159 | 9.859200 |
| medium_4096 | COO | 69.011 | 81.030 | 27113.978 | 392.893568 | 334.616537 |
| large_15120 | COO | 11.135 | 80.269 | 2733.487 | 245.486035 | 34.054081 |
| large_20000 | COO | 58.625 | 477.886 | 19546.453 | 333.414977 | 40.901916 |
| large_21074 | COO | 156.052 | 463.960 | 60900.766 | 390.259439 | 131.262967 |
| large_25605 | COO | 106.102 | 560.486 | 43583.613 | 410.770890 | 77.760395 |

Figure 3: COO Performance List

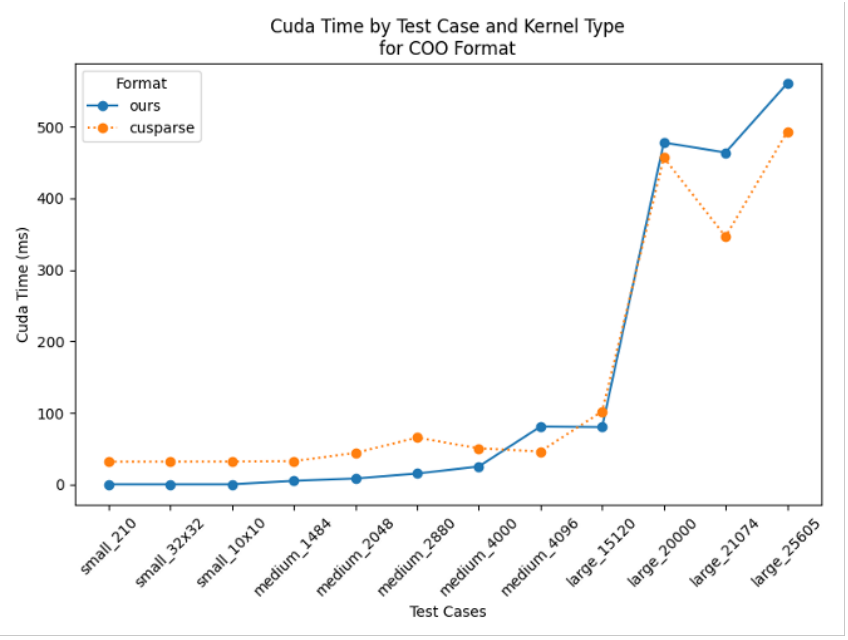


Figure 4: COO kernel test

### 3.2 CSR Format

The CSR format is similar to COO, where the row indices are compressed and replaced by an array of offsets. These offsets represent the starting and position of the values in each row in the value array. A sparse matrix stored in CSR format is represented by the following parameters.

- numRows: The number of rows in the matrix.
- numCols: The number of columns in the matrix.
- numNonZero: The number of non-zero elements (nnz) in the matrix.
- rowPtrs: The pointers to the row offsets array of length number of rows + 1 that represents the starting position of each row in the columns and values arrays.

- colIdxs: The pointers to the column indices array of length nnz that contains the column indices of the corresponding elements in the values array.
- data: The pointers to the values array of length nnz that holds all nonzero values of the matrix in row-major ordering.

The figure 5 shows a 5×4 matrix represented in CSR format.

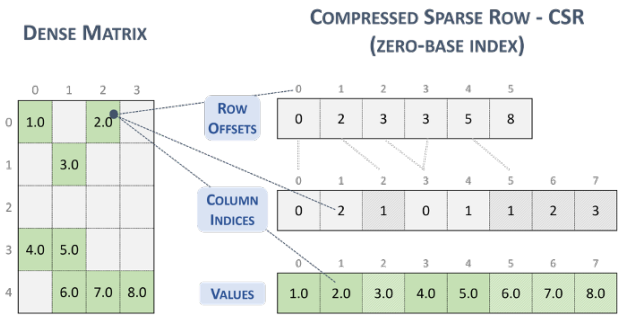


Figure 5: CSR format[3]

### 3.2.1 Basic Kernel

The basic implementation of CSR distributes the computation of 1 element in matrix C to each thread. Therefore, each thread will iterate through the a line of A and a column of B in order to produce the final multiplication-add result. The runtime results are displayed in the following table. The performace is listed in figure 6

| testCase | format | cudaKernelTimeMs | cudaTotalTimeMs | sequentialTimeMs | computation_speedup | speedup |
|---|---|---|---|---|---|---|
| small_210 | CSR | 0.461 | 0.484 | 0.067 | 0.145336 | 0.138430 |
| small_32x32 | CSR | 0.457 | 0.477 | 0.016 | 0.035011 | 0.033543 |
| small_10x10 | CSR | 0.483 | 0.507 | 0.004 | 0.008282 | 0.007890 |
| medium_1484 | CSR | 0.491 | 3.657 | 38.894 | 79.213849 | 10.635494 |
| medium_2048 | CSR | 0.982 | 21.033 | 83.455 | 84.984725 | 3.967812 |
| medium_2880 | CSR | 0.514 | 12.489 | 216.062 | 420.354086 | 17.300184 |
| medium_4000 | CSR | 0.605 | 20.300 | 191.754 | 316.948760 | 9.446010 |
| medium_4096 | CSR | 3.337 | 19.560 | 20864.161 | 6252.370692 | 1066.674898 |
| large_15120 | CSR | 1.189 | 71.508 | 1475.737 | 1241.158116 | 20.637369 |
| large_20000 | CSR | 4.902 | 414.142 | 11089.846 | 2262.310486 | 26.777883 |
| large_21074 | CSR | 13.759 | 392.565 | 30886.603 | 2244.829057 | 78.678953 |
| large_25605 | CSR | 9.290 | 458.807 | 20981.246 | 2258.476426 | 45.730004 |

Figure 6: CSR performance list

### 3.2.2 Optimizations

There are several opportunities to enhance the kernel's efficiency.

First, we can optimize the memory layout of matrix B. Matrix-Matrix multiplication inherently benefits from storing matrix B in a column-major format, as it facilitates better-coalesced memory access in global memory. When computing an element in matrix C, we need to iterate through a column in matrix B. By making matrix B column-major, each thread can access memory in a more sequential pattern, thereby reducing access randomness. However, due to the sparsity of matrix A, a minimal level of randomness in memory access still persists.

Second, we leverage intra-warp communication techniques, such as the __shfl_sync API provided by CUDA. This API enables efficient register-to-register communication across threads within a warp, significantly faster than accessing shared or global memory. The only overhead introduced is the requirement that all threads within the warp participate in the operation.

Third, we utilize shared memory to cache frequently accessed data, reducing unnecessary global memory accesses. Specifically, we cache row pointers and column indices of matrix A, as well as values from matrix B, organized by columns. The feasibility of caching depends on the matrix size; for extremely large matrices, even a single row or column may exceed shared memory limits, making this approach unsuitable in such cases.

Lastly, we exploit the read-only cache to store data that remains immutable during execution. For instance, values from matrix B can be loaded using the __ldg API, which attempts to cache them in the read-only cache. However, this optimization yields limited performance improvement in practice.

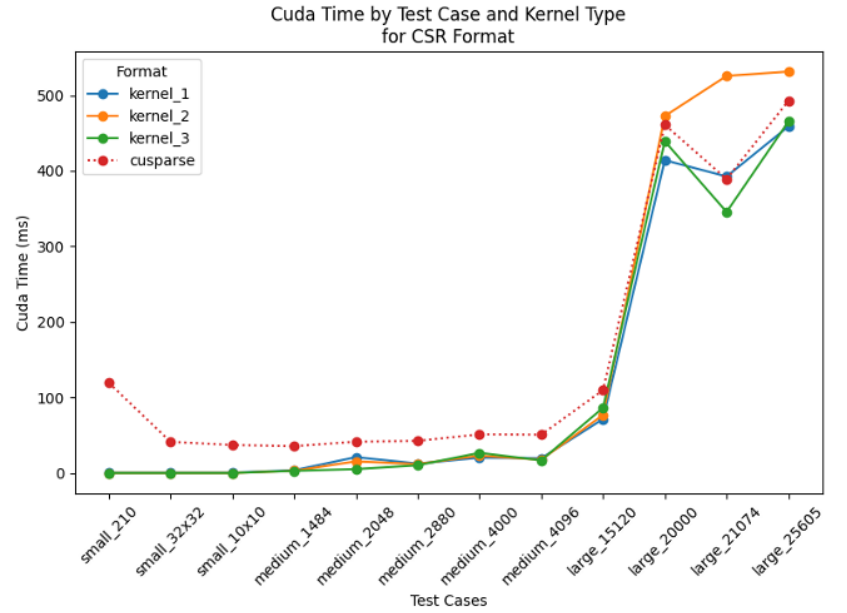


Figure 7: CSR kernel test

### 3.3 BSR Format

The BSR(Block Sparse Row) format is similar to CSR, where the column indices represent two-dimensional blocks instead of a single matrix entry. A sparse matrix stored in BSR format is represented by the following parameters.

- blockRowSize: The block size in row dimension.
- blockColSize: The block size in column dimension.
- numBlockRows: The number of row blocks in the matrix.
- numBlocks: The number of non-zero blocks in the matrix.
- blockRowPtrs: The pointers to the row block offsets array of length number of row blocks + 1 that represents the starting position of each row block in the columns and values arrays.
- blockColIdxs: The pointers to the column block indices array of length numBlocks that contains the location of the corresponding elements in the values array.
- data: The pointers to the values array of length numBlocks that holds all nonzero values of the matrix.

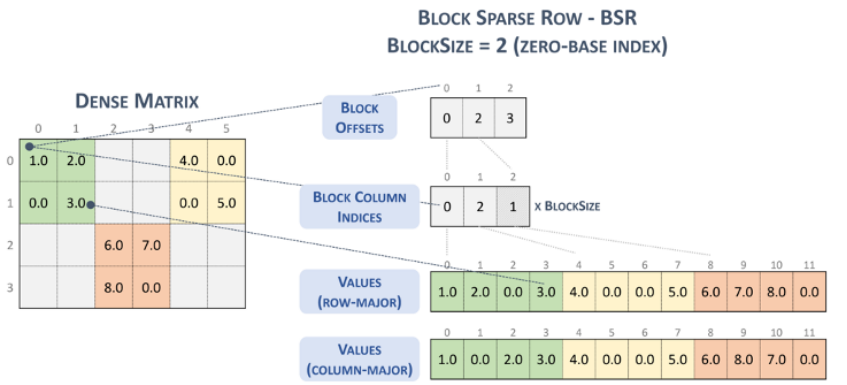


Figure 8: BSR format[3]

### 3.3.1 Basic Kernel

The basic implementation maps each storage block onto one thread block. That is, each thread block will have the shape of (blockRowsize, blockColSize), and will be responsible for processing all computations relative to the data inside this block. For example, if the data is stored with size (4x4), and there are 16 blocks in total, then we will start 16 threads blocks with shape (4x4). To ensure the correctness of the final result, we used atomicAdd to avoid race conditions.

One point worth mentioning is that the number of rows and columns must be divisible by the size of block in row and column dimension, respectively. Otherwise a complicated padding strategy would be required when generating data. Runtime adjustment is extremely expensive.

The performance results of the BSR multiplication is shown in the table below.

| testCase | format | cudaKernelTimeMs | cudaTotalTimeMs | sequentialTimeMs | computation_speedup | speedup |
|---|---|---|---|---|---|---|
| small_210 | BSR | 0.443 | 0.465 | 0.145 | 0.327314 | 0.311828 |
| small_32x32 | BSR | 0.451 | 0.469 | 0.114 | 0.252772 | 0.243070 |
| small_10x10 | BSR | 0.445 | 0.466 | 0.009 | 0.020225 | 0.019313 |
| medium_1484 | BSR | 2.263 | 7.463 | 331.596 | 146.529386 | 44.431998 |
| medium_2048 | BSR | 1.990 | 7.865 | 656.351 | 329.824623 | 83.452130 |
| medium_2880 | BSR | 7.693 | 18.947 | 2821.870 | 366.810087 | 148.934924 |
| medium_4000 | BSR | 9.155 | 34.149 | 976.608 | 106.674823 | 28.598436 |
| medium_4096 | BSR | 188.113 | 201.054 | 83883.636 | 445.921526 | 417.219434 |
| large_15120 | BSR | 50.037 | 117.626 | 27722.060 | 554.031217 | 235.679697 |
| large_20000 | BSR | 312.407 | 750.872 | 176015.051 | 563.415836 | 234.414189 |
| large_21074 | BSR | 2423.662 | 2735.643 | 68571.315 | 28.292441 | 25.065886 |
| large_25605 | BSR | 134.343 | 600.564 | 48764.435 | 362.984562 | 81.197732 |

Figure 9: BSR performance list

### 3.3.2 Optimizations

The BSR format presents one of the most challenging optimization scenarios due to the intricate balance required between the size of the storage blocks and the size of the thread blocks.

If we choose a smaller storage block size, it can reduce the redundancy of zero data, improving both storage efficiency and memory loading efficiency. Storing less data increases the compression ratio and reduces the data volume needed for computation, thereby enhancing the prologue, execution,

and epilogue times. However, smaller storage blocks necessitate more blocks to represent the same matrix, making it inefficient to assign one thread block per storage block. This can result in many small thread blocks with fewer than 32 threads, failing to form a complete warp and underutilizing computational resources.

Conversely, opting for a larger thread block size requires each thread to process multiple storage blocks rather than one, increasing the complexity of workload distribution. To achieve efficient shared memory utilization or implement intra-warp communication techniques, it becomes necessary to design an intricate mechanism to distribute the workload effectively across threads within a warp.

An optimization we conceived:

1. Optimize Storage Block Size: To strike a balance, the storage block size should ideally be no larger than 4×4, maximizing the compression ratio and minimizing memory loading overhead. Smaller storage blocks keep the data sparse and efficient.
2. Workload Mapping with Tile-Based Computation: Adopt a workload mapping strategy where each warp collectively handles a tile of the output matrix C, similar to standard Matrix-Matrix multiplication techniques but with smaller tile sizes. This approach ensures better resource utilization and allows for improved coalesced memory access patterns. The number of elements each thread is responsible for should be further tuned with experiments.
3. Warp-Level Accumulation for Partial Sums: Enable warp-level collaboration, where threads collectively compute local partial sums of matrix products. These sums are then reduced across the warp. This strategy reduces shared memory usage, freeing up space to cache frequently accessed data, such as block pointers or values.
4. Data reuse by caching: Similarly, we can try to cache data in A. But when the number of rows in B is relatively large, it's impossible to cache multiple columns of B in shared memory. It will either use an iteratively loading fashion, or simply use read-only cache if it's too big. We should implement many different kernels to use for different shapes of the input.

Therefore, we need to implement many different kernels for different input sizes. Due to time constraints, further exploration of these optimizations is deferred to future work. These enhancements could potentially yield significant performance improvements by achieving a better balance between computation and memory access efficiency.

### 3.4 ELL Format

The ELLPACK format is another less common sparse matrix storage format. The conventional format stores the non-zero elements in a fixed number of columns for each row regardless of the number of non-zero elements in that row. The values are stored as a 2-D array where each row represents a matrix row and columns represent non-zero elements in that row, padded with zero if needed. The figure below shows an example of an ELL packed matrix in one-based index. ELL format is efficient for matrices that have roughly uniform distribution of non-zero elements, which makes the data access more structured compared to other storage formats. However, this format will use much more space if one row has much more non-zero elements than the other rows because of the need for padding since the number of columns stored depends on the longest row in the matrix. There are also other variants of ELL format such as Sliced ELL and Blocked ELL.

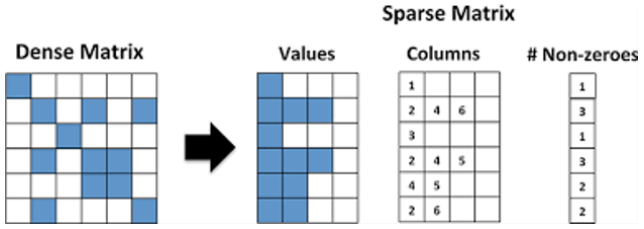


Figure 10: ELL format

The sequential program for implementing ELL sparse matrix multiplication is very similar to regular dense matrix multiplication. The only difference is that instead of iterating through all values in a

row in the sparse matrix, the program only processes non-zero elements in a row and moves on to the next once there is no more non-zero elements row, which can be signaled by invalid column indices.

### 3.4.1 Basic Kernel

| testCase | format | cudaKernelTimeMs | cudaTotalTimeMs | sequentialTimeMs | computation_speedup | speedup |
|---|---|---|---|---|---|---|
| small_10x10 | ELL | 0.281 | 0.478 | 0.010 | 0.035587 | 0.020921 |
| small_32x32 | ELL | 0.287 | 0.496 | 0.026 | 0.090592 | 0.052419 |
| small_210 | ELL | 0.275 | 0.487 | 0.133 | 0.483636 | 0.273101 |
| medium_1484 | ELL | 0.783 | 4.194 | 68.129 | 87.010217 | 16.244397 |
| medium_2048 | ELL | 1.238 | 7.258 | 152.098 | 122.857835 | 20.955911 |
| medium_2880 | ELL | 1.496 | 14.961 | 410.178 | 274.183155 | 27.416483 |
| medium_4000 | ELL | 1.333 | 28.284 | 273.604 | 205.254314 | 9.673455 |
| medium_4096 | ELL | 45.605 | 62.743 | 27212.460 | 596.699046 | 433.713084 |
| large_15120 | ELL | 6.351 | 94.685 | 2780.944 | 437.874980 | 29.370481 |
| large_20000 | ELL | 40.565 | 491.838 | 20033.864 | 493.870677 | 40.732648 |
| large_21074 | ELL | 159.827 | 648.826 | 61229.874 | 383.100940 | 94.370253 |
| large_25605 | ELL | 90.262 | 683.211 | 43800.033 | 485.254404 | 64.109086 |

Figure 11: ELL performance list

The basic kernel maps the threads in each block to one element in the values array, which is represented as 2-D but stored as 1-D array in memory. For threads that are mapped to a valid non-zero element, it performs matrix multiplication for that element by accessing the row that corresponds to the column index in the sparse matrix columns array and calculating the results for all columns in that row of the B matrix. This basic kernel is essentially a modified version of the COO kernel, where some zero-elements are mapped to threads. This can lead to more warp divergence because all threads will now need to check for whether the input element is a non-zero element. As a result, we expected that this kernel performance would perform slightly worse than the basic COO kernel due to increased warp divergence. The performance result is displayed in the table below, and the figure below shows that the runtime of ELL format is indeed slightly slower than that of COO format for the majority of the test cases, confirming our expectation.

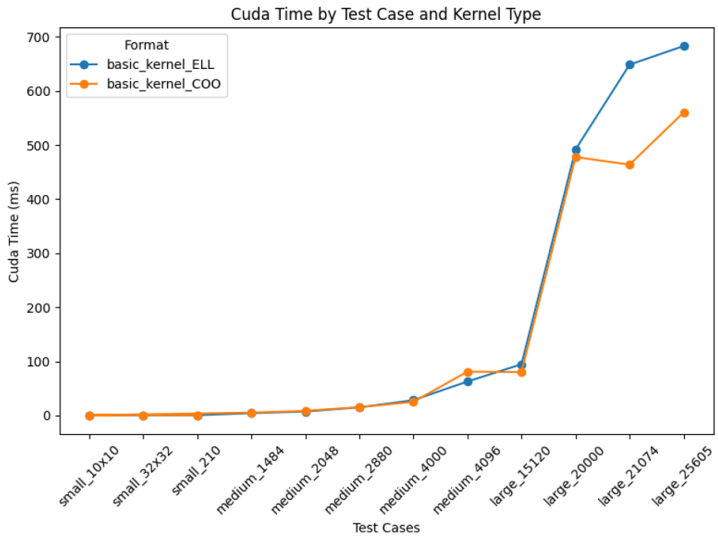


Figure 12: ELL kernel test

### 3.4.2 Optimizations

An observation of sparse matrix multiplication is that each non-zero element will need to access all columns of a specific row, which means that each column of B matrix will be used by all non-zero elements. This gives the possibility of using shared memory because each thread will need to access a different row value when processing the same column of the B matrix, and threads in the thread block can collaborate to load one column into shared memory.

This means that there are two changes that could lower runtime. First, the dense matrix can be changed from row-major storage into column-major storage for easy loading of columns. Second, Instead of each thread iterating through the columns of B matrix independently, they now need to first collaborate to load one element from one column of B, synchronize, then perform necessary multiplication for threads that are mapped to a non-zero element. The performance results of the kernel using shared memory is shown in the table below.

| testCase | format | kernelType | cudaKernelTimeMs | cudaTotalTimeMs | sequentialTimeMs | speedup | computation_speedup |
|---|---|---|---|---|---|---|---|
| small_10x10 | ELL | 2 | 0.021 | 0.078 | 0.009 | 0.115385 | 0.428571 |
| small_32x32 | ELL | 2 | 0.031 | 0.087 | 0.026 | 0.298851 | 0.838710 |
| small_210 | ELL | 2 | 0.028 | 0.101 | 0.136 | 1.346535 | 4.857143 |
| medium_1484 | ELL | 2 | 1.627 | 4.875 | 69.328 | 14.221128 | 42.610940 |
| medium_2048 | ELL | 2 | 6.837 | 29.435 | 161.784 | 5.496314 | 23.663010 |
| medium_2880 | ELL | 2 | 11.002 | 25.980 | 421.527 | 16.225058 | 38.313670 |
| medium_4000 | ELL | 2 | 20.207 | 38.623 | 278.197 | 7.202884 | 13.767358 |
| medium_4096 | ELL | 2 | 214.913 | 229.269 | 28336.967 | 123.597028 | 131.853201 |
| large_15120 | ELL | 2 | 119.003 | 196.476 | 2987.478 | 15.205308 | 25.104224 |
| large_20000 | ELL | 2 | 743.253 | 1181.178 | 21414.494 | 18.129777 | 28.811850 |
| large_21074 | ELL | 2 | 45272.783 | 45727.571 | 72305.498 | 1.581223 | 1.597107 |
| large_25605 | ELL | 2 | 2154.597 | 2715.387 | 49115.510 | 18.087849 | 22.795683 |

Figure 13: ELL performance list for optimizations

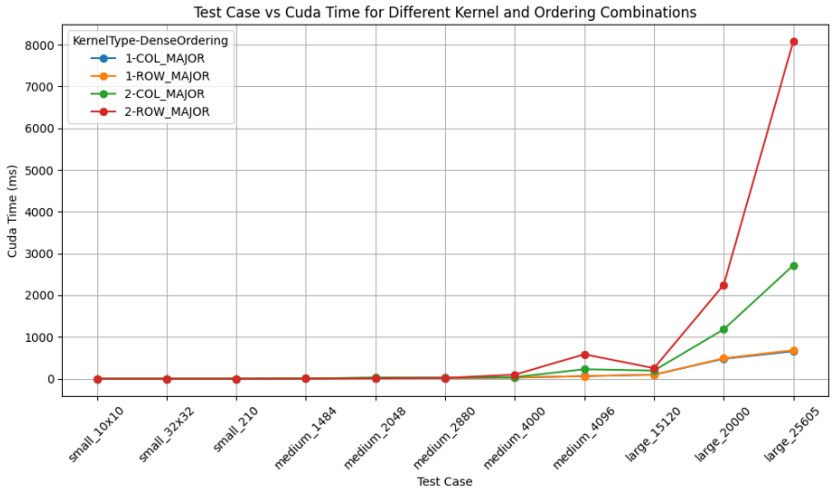


Figure 14: ELL kernel tests 2

The runtime results indicate that the shared memory kernel performs worse than the basic kernel. We also tried to change the dense matrix into column major and used the basic kernel instead of shared memory, and the following graph indicates that regardless of the dense matrix ordering, the basic kernel performs better than the advanced kernel.The graph excludes outlier test case large_21074 for better visual. We believe that there are three reasons that explain the performance drop. First,

each thread will load one row element when processing every column of the B matrix, regardless of whether any non-zero element will need the row elements. This may increase the number of memory operations because unnecessary elements from the B matrix are loaded. Secondly, synchronization is now needed after loading and after calculation, which could increase the amount of unnecessary idle time. Lastly, in order to account for the situation that the shared memory may not be large enough to hold one column of B, multiple loads into shared memory are needed regardless of whether the loaded values will be used. This complicates the shared memory loading and adds even more runtime on wasted memory access on not-used elements and synchronization.

# 4 Results

## 4.1 Evaluation Setup

Experiment platform setup:

1. CPU: AMD EPYC 9534, 256 Cores, 1.5TB Mem
2. GPU: H100 SXM, 80 GB Mem
3. CUDA: CUDA Toolkit 12.4

Result overview:

1. We achieved our 100% goal of performance on all test cases with CSR format, outperforming NVIDIA library cuSparse and even exceeding it by a large margin.
2. We successfully implemented CUDA kernels for all types of storage format, and fine-tuned 2 of them.
3. Beyond the 100% goal, we additionally analyzed sensitivity of our implementation to data density, providing a comprehensive evaluation.

Metrics:

1. Overall runtime: prologue + execution time + epilogue
   (a) Prologue time mainly consists of allocating space for result, searching for suitable thread block shape.
   (b) Execution time includes only kernel execution time.
   (c) Epilogue time includes time copying the data back to cpu.
2. Speedup: speedup is computed against the CPU version of the corresponding storage format. (NOTE: Each storage format has its own CPU implementation.) Regarding our focus on comparing with cusparse, the CPU version is implemented using only 1 core of CPU without unnecessary further optimization.
   (a) Computation speedup considers only kernel time
   (b) Overall speedup considers prologue, execution time and epilogue.
3. Due to hardware limit and restricted permission access, we cannot use Nsight System to profile our kernel to collect low-level metrics for analysis. (This feature is turned off by admin to prevent side channel attacks.)

Baseline selection:

1. Sequential: Our sequential implementations for each format are used to calculate the speedup of the kernels.
2. cuSPARSE: The cuSPARSE library from NVIDIA contains a set of GPU-accelerated basic linear algebra subroutines used for handling sparse matrices that perform significantly faster than CPU-only alternatives. It serves as a good baseline considering because it's widely used among many other applications. Since cusparse supports only COO and CSR format in our version (12.4), we only use it to compare these two formats.

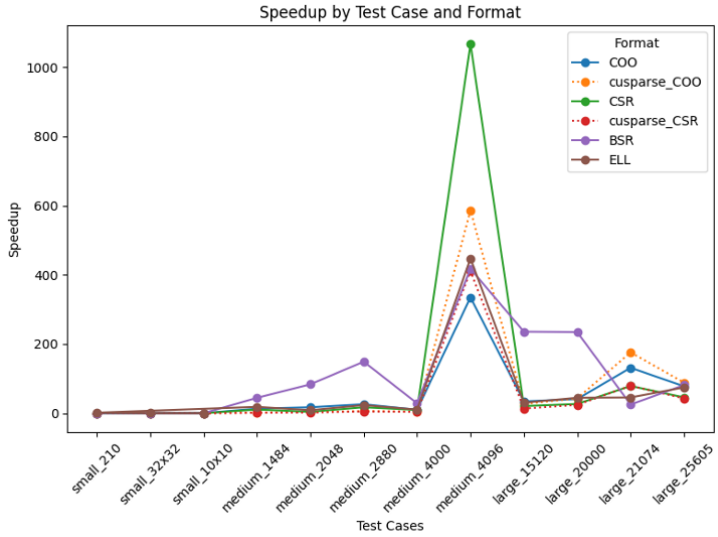


Figure 15: Total speedup by test cases and formats

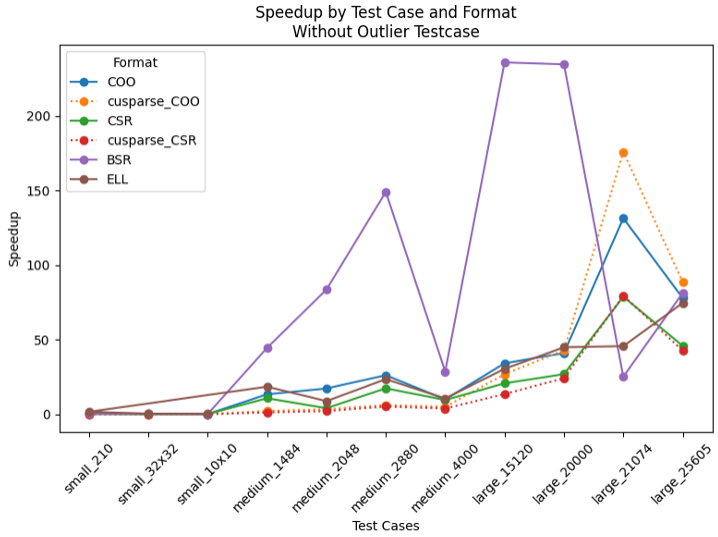


Figure 16: Total speedup by test cases and formats without outlier

### 4.2 Total Speedup By Test Cases and Formats

The figure 15 and figure 16 show the overall speedup of our kernel implementations over the CPU version. Based on the data we collected, we have the following analysis:

1. The speedup shows an overall growing trend as the size of the matrix grows. Because there are more elements to process in matrix C, CPUs take much longer time to compute sequentially, while CUDA kernels can efficiently use parallelism to distribute work to more threads.
2. BSR format is very suitable for CUDA optimization. From the graph, we can see that BSR format has the greatest speedup for many different test cases against CPU. This can be explained by the storage structure of BSR format. Since data is structured and stored as units of blocks, CUDA can easily data to different threads with similar shape, and threads can share the same cache. Otherwise, when the CPU wants to compute an element in matrix C, it has to access data of A in a strided fashion, significantly damaging the cache performance.

3. The special test case medium_4096 has the largest number of non-zero elements among all test cases, around 100 times more, thus resulting in an unnatural curve. The computation overhead grows as the number of non-zero elements increases.

4. With small size test cases, all formats demonstrate the similar performance, while with larger test cases, the variance is shown more obviously. Test case large_21074 has the biggest matrix B among all test cases while having moderately more non-zero elements in sparse matrix A.

5. Compared with cusparse, our implementations of COO and CSR show relatively similar or better results for most cases.

### 4.3 Computation Speedup By Test Cases and Formats

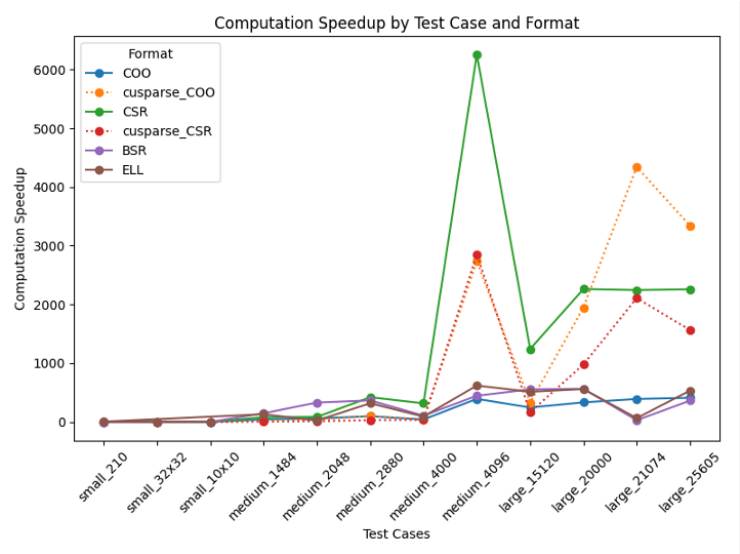


Figure 17: Computation speedup by test cases and formats

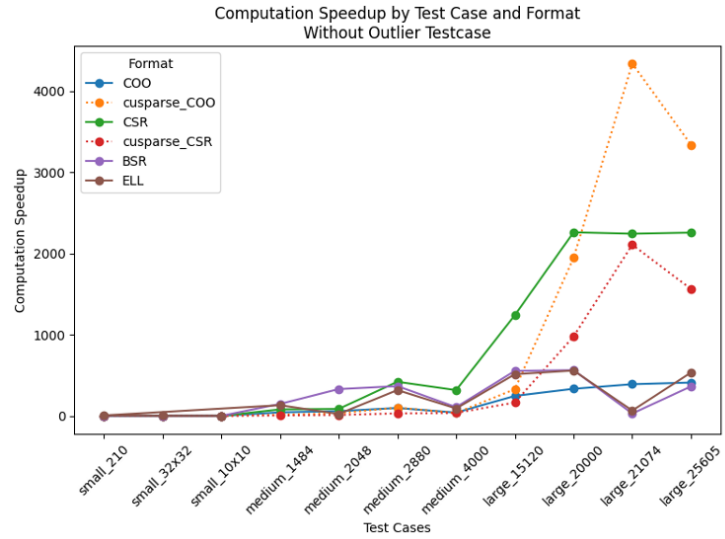


Figure 18: Computation speedup by test cases and formats

The figure 17 and 18 show the speedup, considering only kernel execution time, of our kernel implementations over the CPU version. Based on the data we collected, we have the following analysis:

1. CSR has the best performance for a relatively dense matrix among all formats, as the outlier shows. This can be explained by its highest compression ratio, which would benefit kernel execution by reducing data loading overhead and cache efficiency.
2. Compared with overall speedup, our CSR and COO format have much better kernel execution speedup, showing better execution performance. This indicates that these two formats have much smaller standalone computation time, while having relatively heavy prologue and epilogue.
3. Compared to overall speedup, BSR format is not showing the same outstanding computation speedup. This implies that the computation takes most part of the whole computation of BSR implementation. Since BSR has many redundant zero values, it introduces many unnecessary computations, which downperforms the kernel performance.
4. With a large matrix, cusparse is showing greater computation speedup. This is in line with our expectation, since it is more suitable for big matrices. But our CSR implementation is still better than cusparse, showing even better scalability. While on the other hand, cusparse is more proficient in COO format, indicating more space for improvement of our implementation.

### 4.4 Runtime by Test Cases and Formats

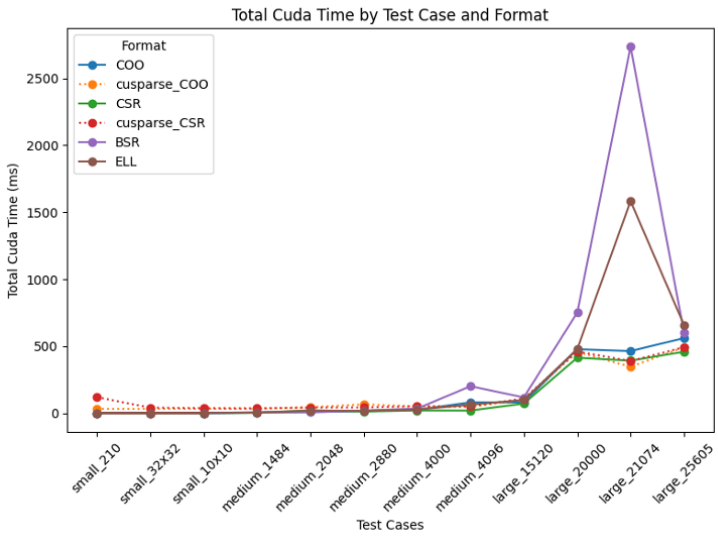


Figure 19: Runtime

1. Among all formats, CSR shows the lowest runtime for almost all test cases. This is just what we have expected. Among CSR, BSR, COO, and ELL, CSR has the greatest compression ratio, giving the lowest memory access overhead and data loading time, therefore having the best memory efficiency. And from an algorithmic perspective, CSR reuses data as much as possible.
2. CSR is dominant for large test cases, showing great scalability. This can be explained by an efficient thread mapping strategy. When there are larger matrices to handle, no race condition and synchronization would need to be resolved, giving the minimal extra overhead.
3. The total runtime of the large_21074 test case for BSR and ELL formats are significantly higher than the other formats, but the Cuda time of both formats are not significantly higher than other formats' Cuda runtime. This indicates that for the specific test case, the runtime for BSR and ELL formats are dominated by copying from and to device instead of the kernel time.

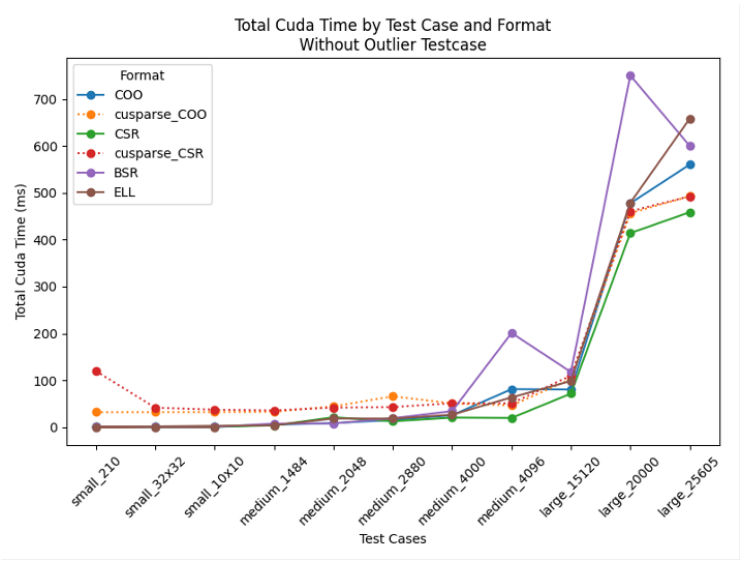


Figure 20: Runtime without outlier

4. Compared to cusparse, our implementation is more good at handling small matrices, because we have relatively low prologue and epilogue time. We are using a deterministic partitioning strategy, which requires only simple arithmetic operations to complete, but cusparse requires setting a lot of attributes in order to perform its algorithm. Its complicated setting gives it more potential when handling extremely large matrices.

### 4.5 Density Sensitivity

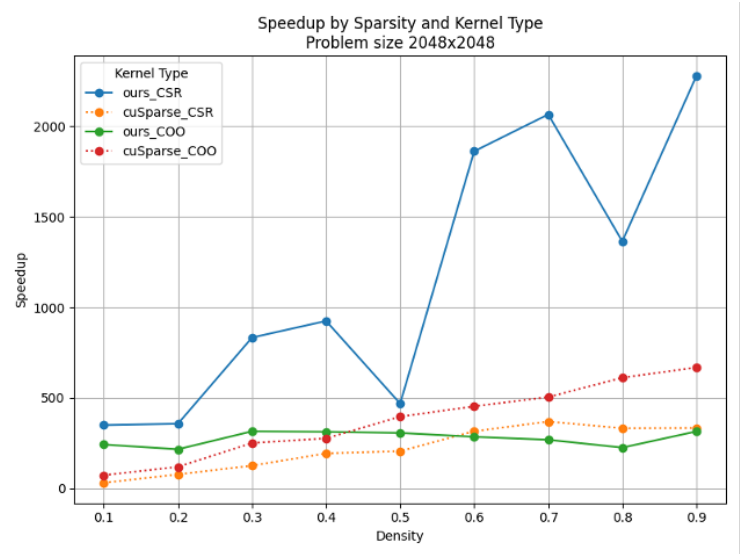


Figure 21: Density test

For the additional experiments of sensitivity to data density, we use CSR format and cusparse-CSR for comparison and analysis. The figure shows the overall speed up against CPU baseline and the overall CUDA runtime. It's obvious that our CSR version defeats the cusparse CSR format on every test case, while outperformed by cusparse on COO format.

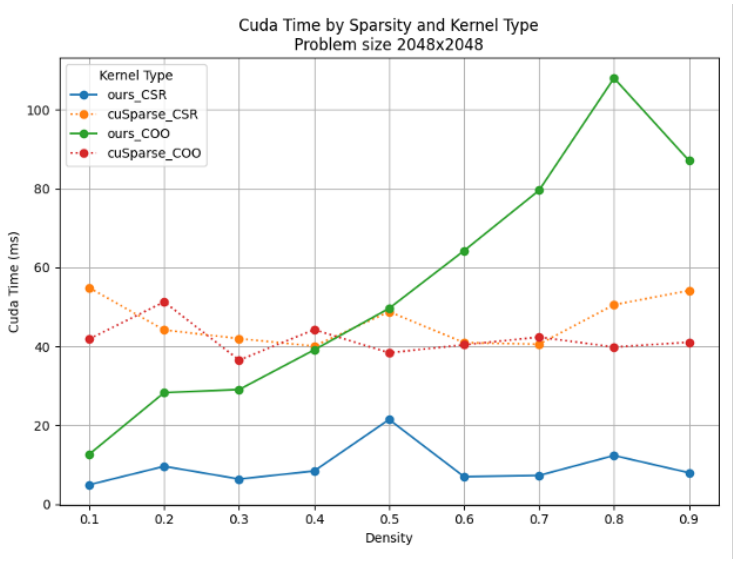


Figure 22: Density test without outlier

As the data density increases, the advantage of our implementation grows. With density=0.2, we have around 2x speedup over cusparse, while with density=0.9, we have nearly 6x speedup. It proves the generality of our implementation.

Based on CUDA runtime:

1. For CSR format, our implementation always launches the same amount of threads, because the matrices have the same size across all different test cases. Therefore, the real computation workload should just increase linearly as data density increases. However, this trend is covered by the long time consumption of prologue and epilogue. For each test case, the prologue and epilogue are copying the same amount of data, and the time is more than kernel execution time, taking a big part of the overall time. Omitting the fluctuation is hardware, we do observe a growing trend in the time consumption, proving our analysis.
2. For cusparse CSR format and COO format, we observed the same effect. The prologue and epilogue are taking the most part, hiding the growing trend of kernel execution time. Since cusparse is not open-source, we cannot peek into its implementation and provide further analysis.
3. For our COO format, the growing trend is much more obvious. Because COO implementation has the worst data compression ratio, it suffers from a longer time of prologue copying data from host to device if there are more non zero elements. Furthermore, COO format launches threads according to the number of non zero elements. When there are more elements to deal with, there are more totally random memory accesses to handle, causing great memory overhead. Lastly, since it uses atomicAdd for forcing consistency, as more threads compete to update the same data, the more synchronization overhead occurs. Therefore, the computation time shows a linear growing trend.

Based on speedup:

1. Our CSR format has the best performance and can exploit CUDA parallelism the most effectively. As the density increases, the computation overhead on CPU grows linearly as well. With just minimally growing CUDA runtime, we observed a growing trend in speedup. It also proves our generality.
2. For cusparse CSR format and COO format, they have much slower growth in speedup, show less capability for fully exploiting parallelism on CUDA.
3. For COO implementation, we observed no speedup growth. This can be explained by the analysis in CUDA runtime.

In conclusion, density sensitivity is mostly determined by two factors: workload distribution and synchronization methods. CSR implementation has a workload distribution that incurs less synchronization overhead across threads, therefore not very sensitive to different density. On the other hand, it demonstrates a better ability to exploit massive parallelism provided by CUDA. In the range of 10%-90%, only COO format shows great sensitivity to data density.

## 5 Conclusion

In this project, we implemented sequential program and CUDA kernel implementations of sparse matrix multiplication using four different formats. We performed analysis on the performance of our kernels and discovered that CSR format performs the best in terms of overall runtime as well as speedup. Even compared to library implementation, our naive kernels for the CSR still managed to perform comparably well or better in most test cases. We found that both the runtime and speedup of the kernels implementations roughly increases as test case increases, which can be attributed to a number of factors such as increased number of operations that lead to increased runtime and significant increase to sequential runtime that lead to increased speedup. We also found that our CSR kernel and the cuSparse implementation of CSR and COO formats does not strongly correlate with the density of the matrix, whereas our COO kernel is positively correlated with the density of the sparse matrix.

## 6 Future Work

1. We will conduct more experiments with more different test cases, to fully disclose the characteristics of these different formats from different perspectives, like data load overhead, memory access efficiency, and cache misses.
2. We will collect more data, conduct experiments along different dimensions, like the number of rows in A, the number of columns in B, the hiddien dimension size, etc.
3. Due to hardware limit and restricted permission access, we cannot use Nsight System to profile our kernel to collect low-level metrics for analysis. We will try to find a same machine with such permission and profile our kernels with professional tools, to deeply reveal their behavior patterns.
4. Due to limited time for project period, we don't have enough time to finish our stretch goals. Initially, we wanted to use advanced GPU features like Tensor Cores, Sparse Tensor Cores[1], or Textures to further accelerate our kernels. However, to write them by ourselves, instead of using libraries, we should program with low-level instructions, even assembly level, which requires a long time to read documentation and conduct experiments. We are still working on it, and may be we will finish it in a few weeks. Welcome to follow our github repository for any updates!

## 7 Code Availability

Our code is all open-source and available at https://github.com/mli43/Cuda-Optimization-for-SpMM. Please refer to README file for environment setup and how to run the code.

## 8 Division of work

The work of this project was contributed equally among teammates.

## References


[1] Roberto L. Castro, Andrei Ivanov, Diego Andrade, Tal Ben-Nun, Basilio B. Fraguela, and Torsten Hoefler. Venom: A vectorized n:m format for unleashing the power of sparse tensor cores, 2023.

[2] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011.

[3] NVIDIA. cusparse, 2024.