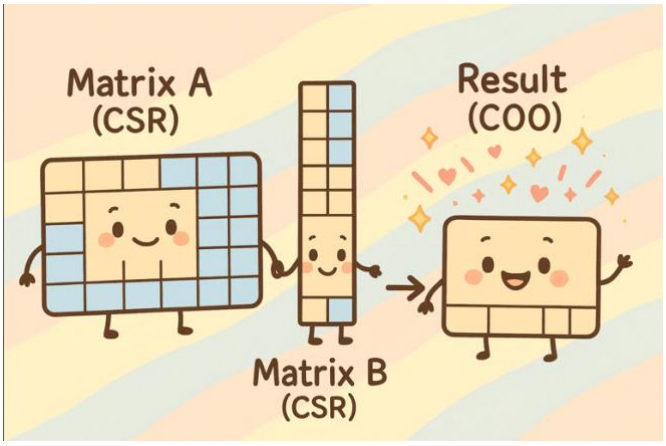
American University of Beirut
School of Arts and Sciences
Department of Computer Science

Sparse Matrix, Very Thin Sparse Matrix



By
Mohammad Khaled Charaf
(mmc51@mail.aub.edu)
Mohammad Ayman Charaf
(mmc50@mail.aub.edu)
Jad Abou Hawili
(jka15@mail.aub.edu)

A Report
submitted to Dr. **Izzat El Hajj** in fulfillment of the project for the course CMPS 224 – GPU Computing

Table of Contents

Milestone 1

What is the problem that the computation solves?

The core problem we aim to solve is the efficient multiplication of two sparse matrices, where one of the matrices is particularly *thin*—i.e., it has significantly fewer columns than rows. Sparse matrix multiplication is a fundamental operation in numerous scientific and engineering applications. However, due to the prevalence of zero entries, traditional dense multiplication algorithms are highly inefficient both in terms of computation and memory usage. Our project focuses on optimizing this computation by leveraging parallelism and sparse matrix storage formats.

Why is the computation important and where is it used in practice?

Sparse matrix multiplication is a cornerstone in many computational domains. Its importance lies in its ability to reduce the computational complexity and memory footprint by avoiding operations on zero elements. This has wide-reaching applications, including:

- **Graph processing**, where adjacency matrices are often sparse.
- **Scientific simulations**, especially those solving systems of partial differential equations (PDEs).
- **Machine learning**, where datasets are often represented as sparse matrices (e.g., in recommendation systems or natural language processing).
- **Computational physics and engineering**, particularly in finite element methods and simulations over large-scale grids.

Efficient multiplication of sparse matrices directly accelerates these applications, making large-scale problems tractable.

How does the computation work? What procedure does it follow?

Sparse matrix multiplication can be implemented in several ways, depending on the chosen data structure. The general idea is to iterate only over non-zero elements, using formats such as:

- **Compressed Sparse Row (CSR)**: Stores row pointers and column indices of non-zero values.
- **Compressed Sparse Column (CSC)**: Similar to CSR but optimized for column access.
- **Coordinate format (COO)**: Stores triplets of (row, column, value).

To multiply a sparse matrix `A` with a thin sparse matrix `B`, we proceed by viewing the multiplication as a series of sparse dot products between the rows of `A` and the columns of `B`. In practice, this may involve:

1. Iterating over each non-zero entry in a row of `A`.

2. For each such entry, accessing the corresponding row in `B` and computing contributions to the resulting matrix.
3. Accumulating the results efficiently, often using intermediate storage like hash maps or index pointers.

This computation is *highly parallelizable*, as each output element can be computed independently (subject to race condition handling). We plan to exploit this parallelism using modern multi-core processors or GPUs.

Artifact

https://github.com/JadAbouHawili/224-project

Overview - Initial Sequential Implementation

Our initial sequential implementation focuses on multiplying two sparse matrices, where:

- **Matrix 1** is stored in **COO (Coordinate Format)**
- **Matrix 2** is stored in **CSR (Compressed Sparse Row Format)**
- The resulting matrix is stored in **COO format**

The goal is to efficiently compute the product while avoiding unnecessary computations on zero entries. To do this, we leverage the strengths of each storage format: COO for its simplicity in iteration, and CSR for its efficient access to all non-zero entries in a given row.

Algorithm Description

The computation proceeds as follows:

1. **Iterate over all non-zero entries** in `Matrix 1` (COO format).
2. For each entry `(i, k, val1)`:
    - Use `k` (the column index in Matrix 1) as a **row index** in Matrix 2 (CSR format).
    - Access all non-zero entries in that row of Matrix 2 using `rowPtrs[k]` and `rowPtrs[k + 1]`.
3. For each non-zero `(k, j, val2)` in that row of Matrix 2:
    - Compute the contribution `val1 * val2` to the resulting matrix at position `(i, j)`.
    - Instead of storing this directly in a 2D structure, we **flatten the 2D index (i, j)** into a 1D key using the formula:

      ```
      index = i * numCols_output + j
      ```

      and accumulate the value in an `unordered_map<int, float>`.

4. After processing all pairs, **invert the flattened indices** to restore row and column information, and store the results in the COO output matrix.

Key Data Structures

- `unordered_map<int, float>` is used to accumulate contributions in the output matrix. This avoids duplicate insertions and enables constant-time updates.
- Index flattening allows us to avoid storing a pair of integers as keys and simplifies hash map operations.

Changes – Optimized Sequential Implementation

In the updated sequential implementation, both matrices are now represented in **CSR (Compressed Sparse Row)** format. This change allows for efficient row-wise access and removes the need for a hash map, resulting in significantly faster execution.

Key Improvements:

- **CSR × CSR multiplication** is now used instead of COO × CSR.
- **Temporary row storage** (an array of size equal to the number of columns in Matrix 2) is used to accumulate results for each output row.
- By iterating over each row of Matrix 1 and multiplying it with the entire Matrix 2, we compute each row of the output matrix directly.
- After computing all contributions to a row, non-zero results are written to the output COO matrix.

Milestone 2

Our initial parallel strategy for sparse matrix multiplication (CSR × CSR) assigns **one GPU thread per row** of the first matrix. That is, for each row `i` in `csrMatrix1`, a unique thread is responsible for computing all the contributions to the `i`-th row in the output matrix. We launch a fixed number of threads per block (e.g., 32), and the total number of blocks is computed as:

```
int numThreadsPerBlock = 32;
cudaMemset(&cooMatrix3->numNonzeros, 0, sizeof(int));
int numBlocks = (numRows1 + numThreadsPerBlock - 1) / numThreadsPerBlock;
mul_kernel<<<numBlocks, numThreadsPerBlock>>>(csrMatrix1, csrMatrix2,
                                              cooMatrix3, numCols2);
```

Thread Computation Logic

Each thread executes the following procedure:

1. It loads the start and end pointers for its assigned row `i` from `csrMatrix1`.
2. For the purpose of managing memory constraints, we divide the output matrix into chunks of columns of size `COL_OUTPUT_SIZE`. Each chunk is processed sequentially in the thread.
3. For each column chunk, the thread:
    - Initializes a local accumulator array `rowOutput[COL_OUTPUT_SIZE]` to zero.
    - Iterates over the non-zero entries of row `i` in `csrMatrix1`. For each entry `(i, k)`, it:
        - Accesses row `k` of `csrMatrix2` using CSR row pointers.
        - Multiplies the corresponding values from both matrices and accumulates partial sums for each column `j` where `(k, j)` is non-zero and falls in the current column chunk.
    - Once the chunk is complete, the thread writes non-zero values from the local accumulator to the output `cooMatrix3` using `atomicAdd` to safely manage concurrent writes.

Here's a simplified view of the loop over column chunks:

```
// Loop over COL_OUTPUT_SIZE-sized chunks of columns
for (int start_col = 0; start_col < numColM2; start_col += COL_OUTPUT_SIZE) {
  int end_col = min(start_col + COL_OUTPUT_SIZE, numColM2);

  // Reset local accumulator
  for (int i = 0; i < COL_OUTPUT_SIZE; ++i) {
    rowOutput[i] = 0.0f;
  }
```

Limitations

While functionally correct and relatively simple to implement, this strategy has a **significant limitation**: assigning a **single thread per row** severely restricts parallelism, especially when:

- The number of rows is small (underutilization of GPU),
- Or some rows have many non-zero entries (load imbalance).

Each thread also performs all computation and memory accesses serially for its row, leading to suboptimal occupancy and execution throughput.

Running Times For Kernal0.cu

```
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -0 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset1 | grep Duration
srun: job 547880 queued and waiting for resources
srun: job 547880 has been allocated resources
    Duration                                                    usecond                     266.85
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -0 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset2 | grep Duration
srun: job 547881 queued and waiting for resources
srun: job 547881 has been allocated resources
    Duration                                                    usecond                     767.46
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -0 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset3 | grep Duration
srun: job 547882 queued and waiting for resources
srun: job 547882 has been allocated resources
    Duration                                                    msecond                       1.35
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -0 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset4 | grep Duration
srun: job 547883 queued and waiting for resources
srun: job 547883 has been allocated resources
    Duration                                                    msecond                       2.02
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -0 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset5 | grep Duration
srun: job 547884 queued and waiting for resources
srun: job 547884 has been allocated resources
    Duration                                                    msecond                       7.55
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -0 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset6 | grep Duration
srun: job 547885 queued and waiting for resources
srun: job 547885 has been allocated resources
    Duration                                                    msecond                      28.31
```

Motivation for Milestone 3:

In our basic parallel implementation, assigning a single thread per row limited parallelism and led to underutilization of the GPU. To address this, we restructured the computation by assigning an entire thread block to each row, allowing multiple threads within the block to collaboratively process the row's non-zero elements. This approach increases parallel efficiency.

Milestone 3

Optimization Description

In this optimization (`kernel1.cu`), we assign **one thread block per row** of `csrMatrix1`. Within each thread block:

- **Each thread** is responsible for a subset of non-zero elements in the assigned row.
- Threads collaboratively accumulate results for chunks of columns (of size `COL_OUTPUT_SIZE`) using **shared memory** (`__shared__ float rowOutput_s[]`), which reduces global memory traffic.
- To avoid race conditions during accumulation, we use `atomicAdd` on shared memory.
- Once all contributions for a chunk are calculated, threads cooperatively write the non-zero entries to the output COO matrix (`cooMatrix3`), again using `atomicAdd` to safely update the global counter `numNonzeros`.

This strategy allows:

- Greater **parallelism** by involving multiple threads per row.
- **Reduced contention** on global memory due to the use of shared memory for intermediate results.
- Better **scalability** with matrix size, especially for wide and sparse matrices.

Launch Configuration Snippet

```
int numThreadsPerBlock = 64;
cudaMemset(&cooMatrix3->numNonzeros, 0, sizeof(int));

// num rows of first matrix
int numBlocks = numRows1;
mul_kernel_opt_1<<<numBlocks, numThreadsPerBlock>>>(csrMatrix1, csrMatrix2,
                                                    cooMatrix3, numCols2);
```

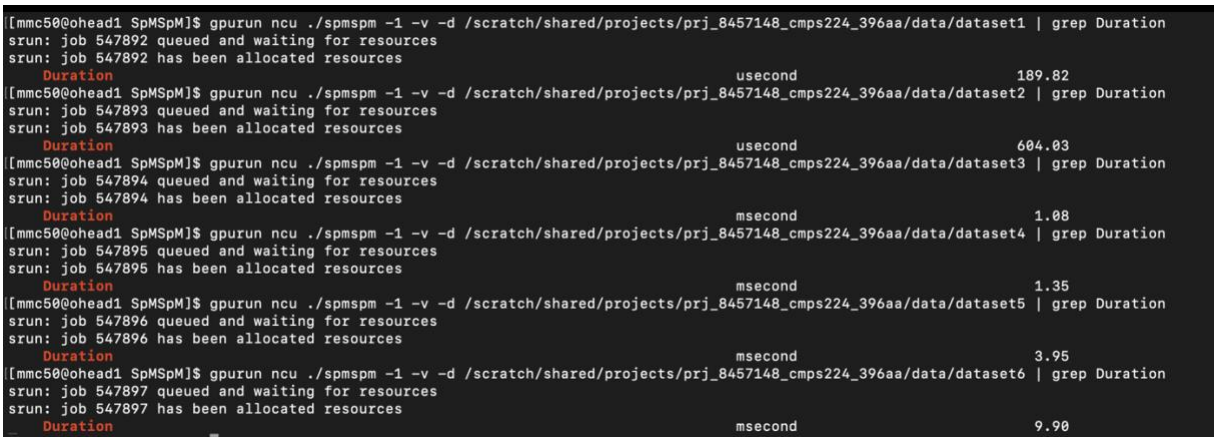
In `mul_kernel_opt_1`, shared memory `rowOutput_s` is used to accumulate values for a fixed chunk of output columns. Each thread processes some of the non-zero elements in its row and performs multiply-accumulate operations with corresponding rows from `csrMatrix2`.

After computing the contributions, each thread writes valid results to the output matrix using `atomicAdd` to safely increment the global counter and avoid write conflicts.

Limitations

While this optimization significantly increases parallelism and reduces global memory access during computation, it introduces a new bottleneck: **all thread blocks write to the same global counter** (`cooMatrix3->numNonzeros`) using `atomicAdd`. This contention can limit scalability as the number of non-zero output elements grows.

Running Times For Kernal1.cu



Motivation for Milestone #4

To reduce contention on the global `atomicAdd` used for writing output entries, we introduce a shared memory counter within each thread block. This allows threads to locally compute their contributions, and then a single thread performs one `atomicAdd` to reserve space in the output matrix. The rest of the threads then write directly to this allocated region. Full details will be provided in the next milestone.

Milestone 4

Optimization Description

In the previous implementation (`kernel1.cu`), every thread that wrote a non-zero element to the output matrix performed an `atomicAdd` on the global counter `cooMatrix3->numNonzeros`. While functionally correct, this approach introduced significant contention on global memory, especially with a large number of non-zero elements.

To address this, our fourth milestone introduces **privatization using shared memory** within each thread block:

1. Each thread block declares a **shared counter** `local_count` that keeps track of how many non-zero elements need to be written for the current column chunk.
2. All threads in the block contribute to `local_count` using `atomicAdd` in shared memory.
3. Once counting is complete, a **single thread** performs one `atomicAdd` to the global counter to reserve a contiguous region in the output matrix and stores the result in a shared variable `start_index`.
4. Threads then write their contributions directly into the output matrix starting from `start_index`, using another shared counter `current_pos` to track write offsets.

This reduces the number of global `atomicAdd` operations from **one per non-zero element** to **one per block**, significantly improving scalability.

Limitations

We notice that threads within a warp frequently access different rows of the second matrix (`csrMatrix2`), leading to non-coalesced memory accesses and degraded performance.

Running Times For Kernal2.cu

```
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -2 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset1 | grep Duration
srun: job 547900 queued and waiting for resources
srun: job 547900 has been allocated resources
    Duration                                                    usecond                       136.80
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -2 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset2 | grep Duration
srun: job 547901 queued and waiting for resources
srun: job 547901 has been allocated resources
    Duration                                                    usecond                       533.95
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -2 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset3 | grep Duration
srun: job 547902 queued and waiting for resources
srun: job 547902 has been allocated resources
    Duration                                                    msecond                         1.63
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -2 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset4 | grep Duration
srun: job 547903 queued and waiting for resources
srun: job 547903 has been allocated resources
    Duration                                                    msecond                         2.20
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -2 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset5 | grep Duration
srun: job 547904 queued and waiting for resources
srun: job 547904 has been allocated resources
    Duration                                                    msecond                         6.84
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -2 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset6 | grep Duration
srun: job 547905 queued and waiting for resources
srun: job 547905 has been allocated resources
    Duration                                                    msecond                        20.41
```

Motivation for Milestone 5

To address memory divergence, our next optimization assigns **one warp per element** in the current row of the first matrix. This allows the warp to cooperatively and coalescently process the corresponding row in `csrMatrix2`, improving memory access efficiency and reducing divergence.

Milestone 5

For this milestone, please consider both kernal3.cu and kernal4.cu for us as part of this milestone, just to see our incremental work of optimization more clearly.

Objective:

Eliminate memory divergence and further reduce contention when writing output by using warp-level coordination.

Key Ideas:

1. **Warp-per-element processing**
   - Instead of one thread handling a single non-zero, we assign **one warp** (32 threads) to each non-zero element in the current row of `csrMatrix1`.
   - All 32 lanes in the warp collaboratively traverse the corresponding row of `csrMatrix2` in **coalesced** chunks of size 32, ensuring that memory accesses to `csrMatrix2->colIdxs` and `csrMatrix2->values` are aligned and coalesced.
2. **Warp-level primitives for output writes**
   - After accumulation into shared memory `rowOutput_s`, we let **only the first warp** handle the write-back.
   - Each lane in that warp checks whether its assigned column slot is non-zero (`is_nonzero`).
   - Using `__ballot_sync(…)`, the warp builds a 32-bit mask of active lanes.
   - `__popc(mask & ((1 << laneId) - 1))` computes each lane's **prefix sum** (its position among active lanes).
   - A **single lane** (e.g. lane 0) does one `atomicAdd` to reserve `popc(mask)` slots in the global COO counter.

- `__shfl_sync` broadcasts that starting offset to all lanes, so each active lane can compute its unique global write-position without further atomics.

This approach reduces global atomic operations from one per non-zero to **one per warp per column chunk**, and ensures fully coalesced reads from `csrMatrix2`.

Limitation of **`kernel3.cu`**:

- We hard-code that only **one warp** (the first) writes back, so `COL_OUTPUT_SIZE` is effectively capped at 32. Additional warps in the block cannot participate in output writes, leaving potential parallelism unused.

Running Times For Kernal3.cu

```
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -3 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset1 | grep Duration
srun: job 547907 queued and waiting for resources
srun: job 547907 has been allocated resources
    Duration                                                    usecond                       195.78
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -3 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset2 | grep Duration
srun: job 547908 queued and waiting for resources
srun: job 547908 has been allocated resources
    Duration                                                    usecond                       600.22
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -3 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset3 | grep Duration
srun: job 547909 queued and waiting for resources
srun: job 547909 has been allocated resources
    Duration                                                    msecond                         1.25
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -3 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset4 | grep Duration
srun: job 547910 queued and waiting for resources
srun: job 547910 has been allocated resources
    Duration                                                    msecond                         1.87
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -3 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset5 | grep Duration
srun: job 547911 queued and waiting for resources
srun: job 547911 has been allocated resources
    Duration                                                    msecond                         6.02
[mmc50@ohead1 SpMSpM]$ gpurun ncu ./spmspm -3 -v -d /scratch/shared/projects/prj_8457148_cmps224_396aa/data/dataset6 | grep Duration
srun: job 547912 queued and waiting for resources
srun: job 547912 has been allocated resources
    Duration                                                    msecond                        22.00
```

Extension in `kernel4.cu`

Objective:

Scale the warp-level writing scheme to **multiple warps**, removing the 32-column cap and exploiting all threads in the block for output.

Key Changes:

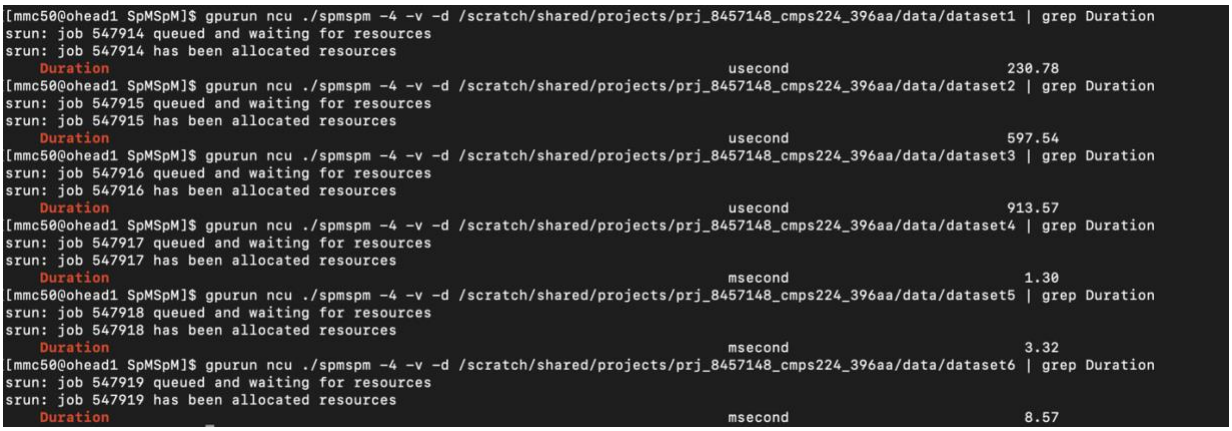
1. **Larger column chunks**
   - Increase `COL_OUTPUT_SIZE` (e.g. to 512), so each block accumulates many more columns in shared memory.
2. **Multi-warp write loops**
   - After accumulation, we iterate over the shared buffer in **strides of** `WARP_SIZE × WARPS_PER_BLOCK`.
   - In each iteration, **every warp** in the block repeats the same **ballot/popc/shuffle** pattern over its own 32-column sub-chunk.

- Each warp performs exactly one `atomicAdd` per sub-chunk to reserve space for its non-zero entries, and then writes its active lanes directly into that region.

Benefits:

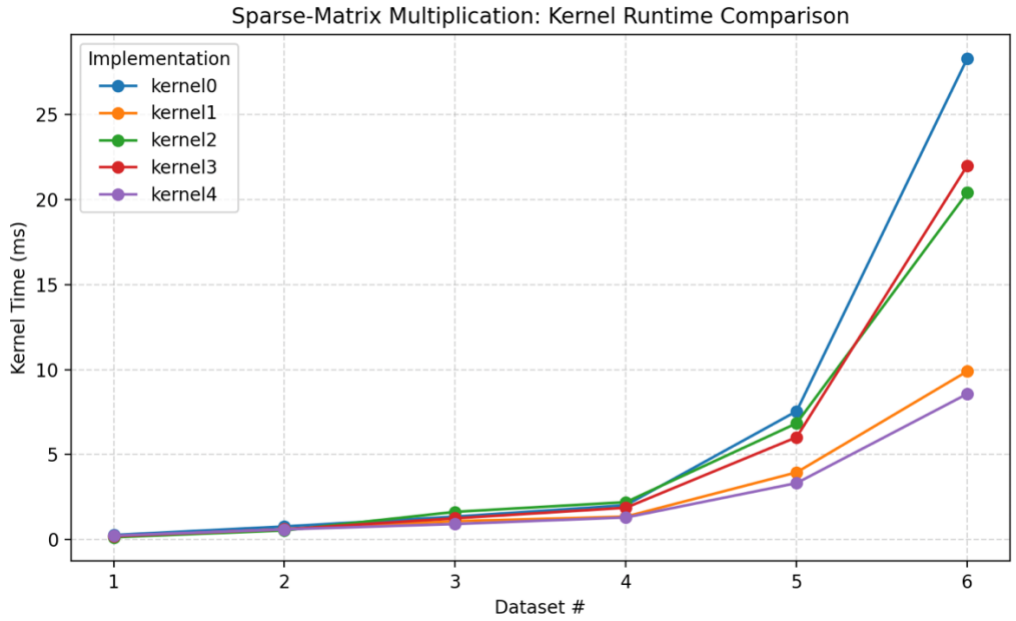
- **Full block utilization**: All warps can now write concurrently, rather than only one.
- **Flexible chunk size**: `COL_OUTPUT_SIZE` no longer tied to warp width.

Running Times For Kernal4.cu

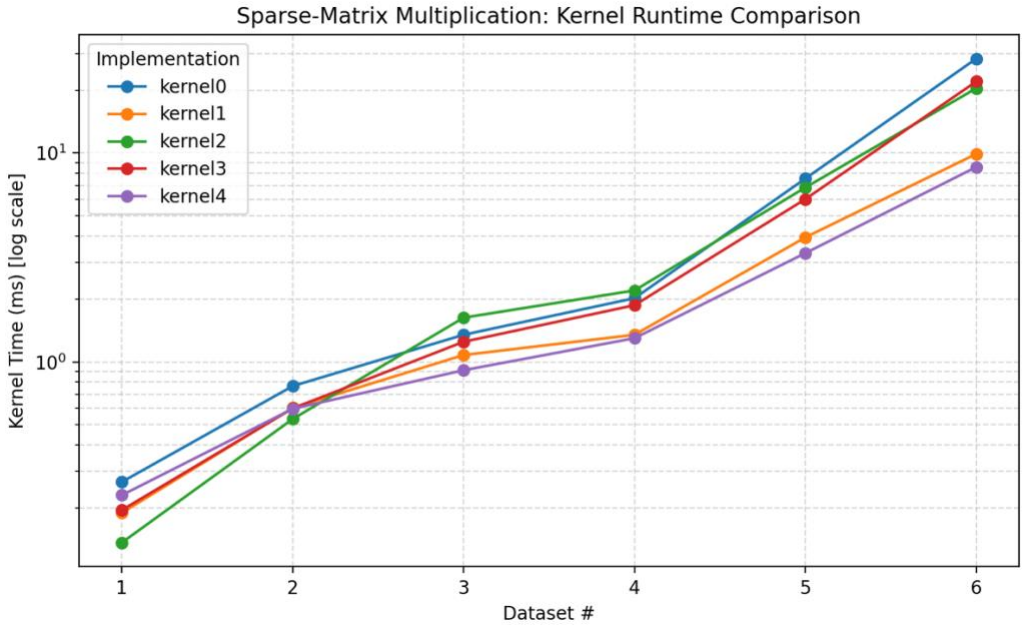


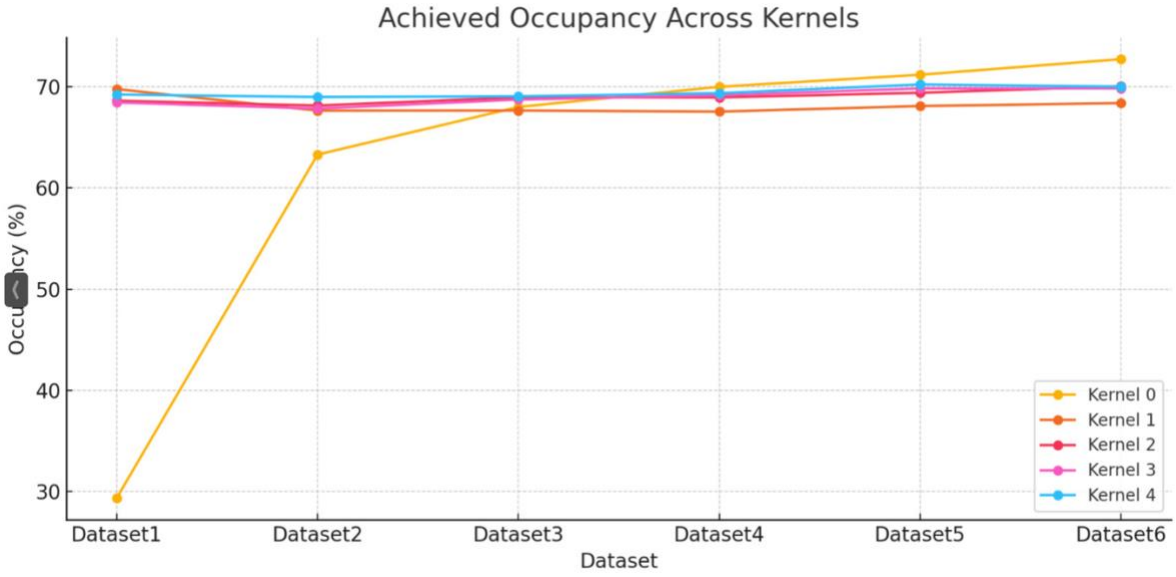
Evaluation

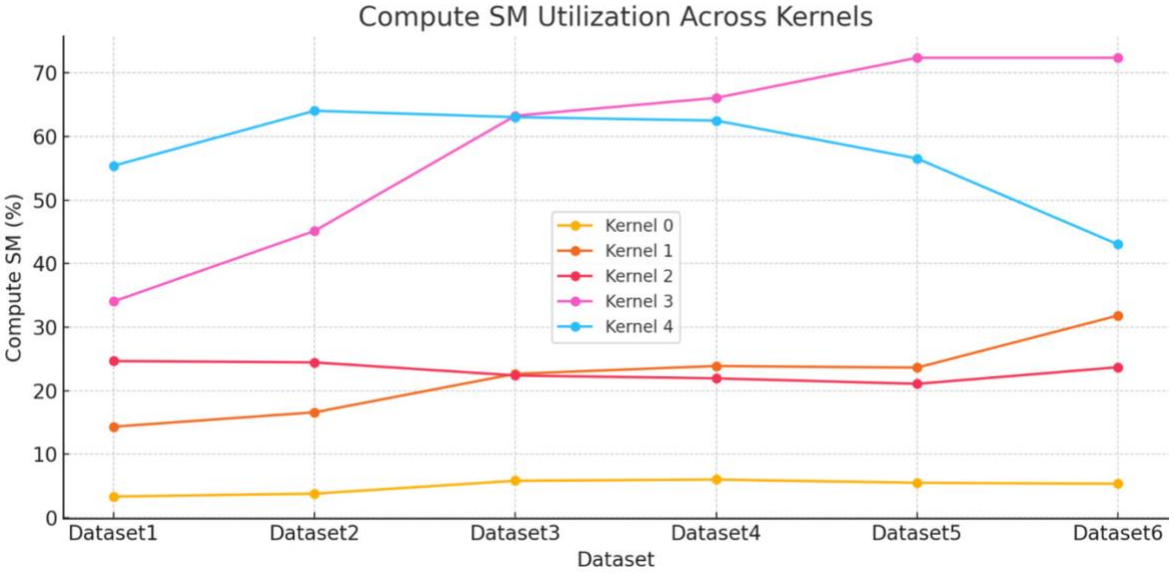
Using a python script, I plot the running times for each kernal as a function of the dataset.

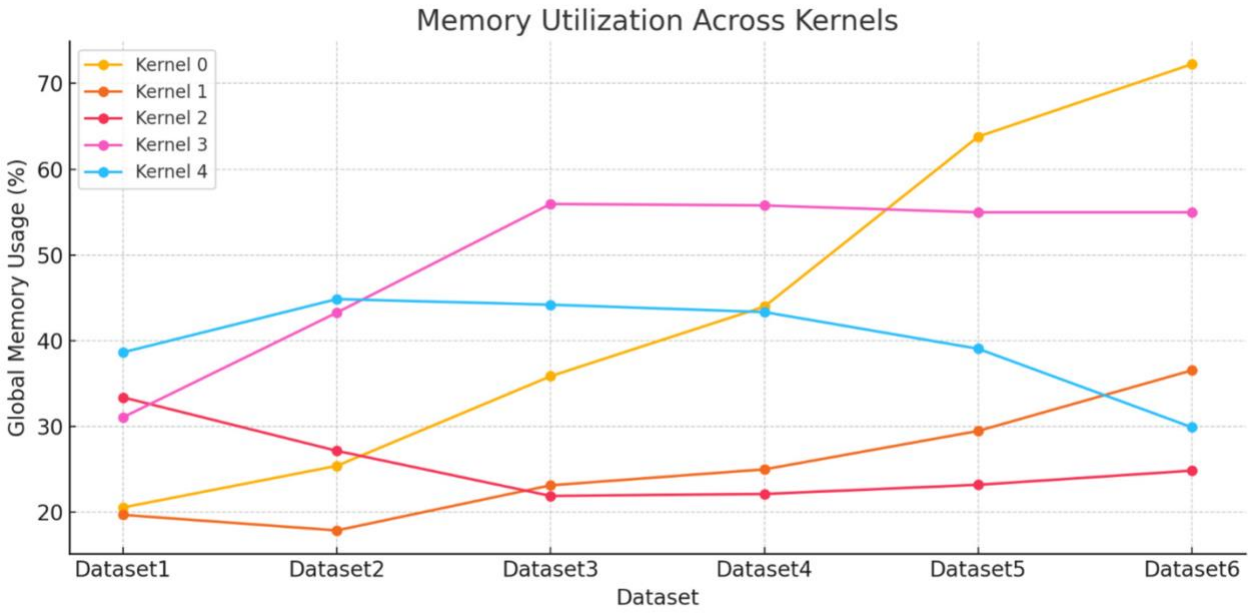


Comparison is not very clear, for smaller datasets, so we apply a log-scale.




Achieved Occupancy Across Kernels
Occupancy (%)
70
60
50
40
30
Dataset1
Dataset2
Dataset3
Dataset4
Dataset5
Dataset6
Dataset
Kernel 0
Kernel 1
Kernel 2
Kernel 3
Kernel 4


Compute SM Utilization Across Kernels
Compute SM (%)
70
60
50
40
30
20
10
0
Dataset1
Dataset2
Dataset3
Dataset4
Dataset5
Dataset6
Dataset
Kernel 0
Kernel 1
Kernel 2
Kernel 3
Kernel 4



Analysis:

The evaluation compares five GPU kernel implementations (Kernel 0–4), each incrementally optimized, using six sparse matrix datasets of increasing size. Execution was performed on the university HPC cluster with an NVIDIA GPU and 16 GB of memory. The table and plot below show kernel execution times and speedups relative to the baseline (Kernel 0). Kernel 0 exhibited poor performance and low occupancy (29% on Dataset1), while Kernel 1 significantly improved occupancy (~68%) and compute utilization. Kernel 2 optimized memory usage but showed marginal compute gains. Kernel 3 achieved peak compute utilization (~72%) and maintained high memory throughput, making it the most balanced in terms of resource use. Kernel 4 introduced advanced optimizations, improving performance further on smaller datasets, but its compute percentage dropped on larger datasets, suggesting a memory bottleneck. Although sparse matrix multiplication is inherently compute-bound, the excessive memory accesses—especially to the second input matrix—diminish performance and prevent full exploitation of compute resources. As a direction for future work, performance could be further improved by reducing these memory accesses, potentially through pre-sorting the second matrix to improve locality, using shared memory tiling strategies, or exploring data layout transformations that enable coalesced accesses and better cache utilization.