Accelerating Sparse Matrix Multiplication

Chun-Lin Chen, Pin-Yen Chen, Yvonne Shih, Aaron Lee

December 10, 2025

Project Repository:
`https://github.com/chunlin-chen/Sparse_MatrixMul`

Abstract

Sparse matrix multiplication is a core operation in scientific computing, data analytics, and machine learning, where exploiting sparsity is essential for reducing arithmetic cost and memory traffic. However, irregular nonzero patterns introduce challenges such as unpredictable memory access, load imbalance, and synchronization overhead, which limit the effectiveness of naive parallelism on both CPUs and GPUs. This project presents a systematic evaluation of five implementations that span three execution models: CPU-only, GPU-only, and a hybrid CPU–GPU pipeline. The implementations include a naive dense CPU baseline, a multithreaded CSR-based CPU version, a naive dense GPU kernel, an optimized CSR GPU kernel, and a double-buffered CPU–GPU pipeline that overlaps CSR preprocessing with device computation. We analyze how sparsity level, matrix dimension, and thread-level parallelism affect performance across architectures. Experimental results show that GPU-only CSR achieves 50×–200× speedup over the multithreaded CPU-only CSR baseline, with higher sparsity providing greater acceleration. The CPU–GPU pipeline CSR version offers modest improvements when CPU preprocessing and GPU computation are balanced, but provides limited benefit otherwise due to CSR conversion bottlenecks. Detailed analysis of bottlenecks and optimization risks highlights key factors governing performance and provides guidance for future sparse-kernel design.

1 Introduction

Sparse matrix multiplication is a fundamental operation that appears in scientific computing, machine learning, graph analytics, and many large-scale numerical applications. Real-world matrices often contain a high proportion of zeros, and exploiting this sparsity is essential for reducing memory traffic and computation time. Despite this advantage, sparse workloads present unique challenges because nonzero elements are distributed irregularly. This irregularity leads to unpredictable memory access patterns, poor cache locality, and load imbalance. As a result, achieving high performance across both CPUs and GPUs requires careful design of data structures, work decomposition, and memory management.

Modern computing systems provide multiple levels of parallelism through multicore CPUs and massively parallel GPUs. Each architecture offers different strengths. CPUs excel at control-heavy tasks such as pointer traversal, format conversion, and symbolic analysis. GPUs provide high arithmetic throughput but require regular access patterns and well-structured kernels to reach peak utilization. Designing an approach that leverages both architectures is therefore a key objective. In this project, we explore this idea through a hybrid CPU–GPU framework that targets efficient sparse matrix–matrix multiplication (SpMM/SpGEMM).

Our work is motivated by three central performance questions. The first concerns how much speedup can be gained simply by moving from a dense formulation to a sparse one. The second asks how multithreaded CPU execution compares with GPU acceleration when both use compressed sparse formats. The third considers whether overlapping CPU preprocessing with GPU computation can reduce end-to-end latency. Addressing these questions requires implementations that isolate the effects of algorithmic choices as well as hardware parallelism.

To support this analysis, we implement five versions of the multiplication. A naive dense CPU version provides a correctness reference. A CPU CSR version with OpenMP introduces row-level parallelism and efficient memory usage. On the GPU side, a dense baseline kernel captures the behavior of a straightforward CUDA approach. A CSR-based GPU kernel serves as the main optimized implementation and uses shared memory to reduce global traffic. Finally, a CPU–GPU pipeline attempts to overlap CSR conversion on the host with computation on the device through a double-buffer mechanism.

The introduction of these versions allows us to perform a structured comparison of performance trade-offs among different architectures and data layouts. Dense kernels reveal the cost of unnecessary computation. CSR implementations highlight the benefits of skipping zeros. GPU kernels expose bottlenecks such as synchronization, shared memory pressure, and row-dependent workload imbalance. The pipeline exposes the interaction cost between the CPU and GPU and shows when overlapping execution is effective.

This report organizes the investigation into several components. We begin with background material on sparse formats and overall system workflow. We then describe the methodology behind each implementation and analyze the design choices that shape their performance. After that, we present detailed GPU kernel behavior, followed by a broad experimental evaluation across different sparsity levels and matrix sizes. The report concludes with a discussion of bottlenecks, lessons learned, and potential directions for future optimization.

2 Background

Sparse matrix multiplication is a central operation in many scientific and data-driven fields. It appears in numerical simulation, optimization, machine learning, and graph analytics. These applications often work with matrices that contain only a small fraction of nonzero values. Exploiting this sparsity reduces the number of arithmetic operations and significantly lowers memory usage. Although this seems to simplify the problem, sparse computation introduces a different set of difficulties that do not arise in dense matrix multiplication.

Sparse matrices exhibit structure that is highly irregular. The location of nonzero entries varies from row to row, and the pattern often depends on the underlying domain rather than on predictable numerical properties. As a result, the amount of work associated with each row can differ widely. The cost of multiplying two sparse matrices is therefore determined not only by their dimensions but also by the distribution of nonzero elements. This distribution affects how data is accessed in memory, how much parallelism can be extracted, and how reliably threads can share work without unnecessary synchronization.

2.1 Sparse Matrix Computation

The multiplication of sparse matrices requires selective traversal of nonzero elements. Unlike dense multiplication, which processes all entries in a regular grid, sparse multiplication forms products only where both operands contain nonzero values. This reduces arithmetic cost but complicates memory behavior. Each multiplication depends on the structure defined by the nonzero pattern, and accessing that pattern involves jumps to different locations in memory. These jumps disrupt cache locality on CPUs and reduce the chance of coalesced memory access on GPUs. They also cause the execution flow to depend on the data itself rather than on uniform iteration ranges.

The effectiveness of a sparse algorithm is therefore determined by how well it handles these structural variations. A good strategy must manage the uneven work across rows, maintain consistent memory throughput, and accumulate partial results efficiently. These concerns influence the design of both sequential and parallel implementations, regardless of the underlying hardware.

2.2 Computational Challenges in Sparse Multiplication

Sparse multiplication faces several inherent challenges that limit performance across architectures. The first challenge arises from irregular memory access. The position of a nonzero element is rarely predictable, so accesses tend to jump across memory regions. This reduces the benefit of caching on

CPUs and prevents efficient memory transactions on GPUs. Irregular access also makes it hard to prefetch data or reuse values across threads.

A second challenge is load imbalance. Some rows may contain only a few nonzero entries while others contain hundreds or thousands. Parallel threads may finish early or late depending on the structure of the data, which leads to poor utilization of processing resources. Load imbalance is especially visible on massively parallel architectures where uniform workloads are important for maintaining high throughput.

A third challenge involves the accumulation of intermediate products. Sparse multiplication generates partial values that must be combined into final results. These values often map to the same output location, so multiple threads may attempt to update the same entry. This requires synchronization or specialized data structures. Even lightweight synchronization can introduce substantial overhead, and the accumulation process often becomes a limiting factor in high-performance settings.

A final challenge comes from preprocessing and data movement. Sparse multiplication typically requires each matrix to be prepared in a form that exposes only its nonzero structure. This preparation is sometimes more expensive than the multiplication itself. When GPUs are used, data must also be transferred between the host and the device. These transfers introduce delays that must be considered when evaluating the end-to-end cost of a sparse computation.

2.3 Motivation and Project Goal

The purpose of this project is to examine how these challenges influence the performance of sparse matrix multiplication on modern parallel hardware. We seek to understand how sparsity affects the gap between dense and sparse computation, how far multithreaded CPU execution can be scaled, and under which conditions GPU acceleration becomes effective. We also investigate whether overlapping CPU preprocessing with GPU computation can reduce total runtime, and how different design choices shape the behavior of the underlying kernels.

To explore these questions, we implement several alternative approaches that represent different combinations of data layouts, parallelization strategies, and hardware execution models. These implementations allow us to study the interaction between algorithmic structure and architectural features. They also provide a basis for detailed performance comparison under varying matrix sizes and sparsity levels. The goal is not to create a single optimal algorithm but to form a clear experimental framework that reveals how sparse computation behaves across CPUs and GPUs and to identify the factors that determine its efficiency.

3 System Overview

This chapter presents the input representation and high-level data flow used in all implementations. Our system takes two sparse matrices as input, converts them into a compact storage format, and then feeds this representation to different CPU and GPU kernels for multiplication.

3.1 Problem Setting

We consider the multiplication of two square sparse matrices A and B of size $N \times N$. The goal is to compute $C = A \times B$ while exploiting sparsity to reduce work and memory traffic. The matrices may originate from dense generators or external data, but all computation in our system is performed on a compressed representation.

3.2 Compressed Sparse Row (CSR) Representation

We use the Compressed Sparse Row (CSR) format as the standard input representation. CSR is widely used in numerical computing because it stores only the nonzero structure of a matrix while preserving direct access to row boundaries. It consists of three arrays:

- **rowPtr**: an array of length $n + 1$ where n is the number of rows. rowPtr[i] marks the starting index of row i in the data arrays.
- **colIdx**: an array storing the column indices of the nonzero elements.
- **values**: an array storing the numerical values of the nonzero elements.

These arrays together describe the sparsity structure and the data needed for multiplication without requiring a dense matrix.

3.3 Dense-to-CSR Conversion

When the matrix starts in dense form, we convert it to CSR using a three-step algorithm. The procedure follows the structure summarized in our slides.

3.3.1 Step 1: Count Nonzero Elements Per Row

The system scans each row of the dense matrix and counts how many nonzero entries it contains. This produces a temporary array that records the number of nonzeros for every row. The result determines the final size of the CSR arrays and defines how much space each row will occupy.

This loop is independent across rows and can be parallelized on the CPU with OpenMP:

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    int count = 0;
    for (int j = 0; j < N; ++j) {
        if (dense[i][j] != 0) {
            count++;
        }
    }
    row_nnz[i] = count;
}
```

For example, consider

$$A = \begin{bmatrix} 0 & 2 & 0 & 0 & 5 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 3 & 4 & 0 & 0 \end{bmatrix}.$$

Row 0 has two nonzeros, row 1 has one, and row 2 has two. This produces row_nnz = [2, 1, 2].

3.3.2 Step 2: Build the Row Pointer Array

The row pointer array is generated by taking the prefix sum of the nonzero counts. The process begins by setting rowPtr[0] to zero. Each subsequent entry accumulates the total number of nonzeros up to that row. After processing all rows, rowPtr[n] equals the total number of nonzero elements in the matrix. This prefix-based structure gives every row a unique and contiguous segment inside colIdx and values.

```
rowPtr[0] = 0;
for (int i = 0; i < N; ++i) {
    rowPtr[i + 1] = rowPtr[i] + row_nnz[i];
}
int total_nnz = rowPtr[N];
```

Using the previous example, we obtain

rowPtr = [0, 2, 3, 5] and total_nnz = 5.

3.3.3 Step 3: Fill Column Indices and Values

The system scans each row again and writes the nonzero entries into the appropriate positions defined by rowPtr. For every nonzero value encountered, its column index is stored in colIdx and its numerical value is stored in values. The write position increments as each nonzero is inserted. Because each row writes only within its designated segment, the procedure does not require synchronization and is deterministic.

```
#pragma omp parallel for
for (int i = 0; i < N; ++i) {
    int offset = rowPtr[i];
    for (int j = 0; j < N; ++j) {
        float v = dense[i][j];
        if (v != 0.0f) {
            colIdx[offset] = j;
            values[offset] = v;
            offset++;
        }
    }
}
```

In the running example, the resulting CSR representation is:

rowPtr = [0, 2, 3, 5] colIdx = [1, 4, 0, 1, 2] values = [2, 5, 1, 3, 4].

3.4 Why CSR Is Efficient

The dense-to-CSR conversion has two main phases: counting nonzeros and filling the CSR arrays. Both phases are simple loops over rows and columns and can be parallelized over rows. Conceptually, they form a map stage (counting nnz) followed by a scan stage (writing CSR arrays).

For an $N \times N$ matrix, the total work of the conversion is $O(N^2)$ operations per thread when fully parallelized over rows. Once the matrix is in CSR form, later kernels operate in time proportional to the number of nonzero entries rather than N^2.

The memory footprint is also compact. The rowPtr array uses $O(N)$ space. The colIdx and values arrays together use $O(\text{nnz})$ space, where nnz is the number of nonzero entries. This is significantly smaller than storing all N^2 elements when the matrix is highly sparse.

3.5 Why CSR Is Easy to Parallelize

CSR exposes a natural unit of parallel work: the matrix row. Rows are independent, and their nonzeros occupy separate segments of colIdx and values. This structure has several consequences that are beneficial for parallel implementations.

First, rows can be processed concurrently without conflicts. A thread responsible for row i reads the range $[\text{rowPtr}[i], \text{rowPtr}[i+1])$ and does not touch the ranges of other rows. Second, the prefix-summed rowPtr array gives each row a fixed memory slice, which eliminates the need for dynamic allocation or locking during traversal. Third, all CSR arrays are read-only once constructed, so they can be safely shared among threads on both CPUs and GPUs. Finally, the predictable pattern of row access improves cache behavior on the CPU and simplifies work distribution in GPU kernels, where a block or warp can be mapped to one or more rows.

These properties make CSR a convenient and efficient representation for the sparse matrix multiplication methods that we describe in the next chapter.

4 Methodology and Implementation Approaches

This chapter presents the five implementations developed for sparse matrix multiplication in this project. Each implementation targets a different execution model and provides a distinct viewpoint on algorithmic behavior, memory patterns, and parallel performance. Together, they form a complete experimental framework that allows us to evaluate the effect of sparsity, thread-level parallelism, and heterogeneous CPU–GPU execution.

The five implementations are grouped into three categories:

- **CPU-only Implementations**
  - **CPU-only Dense Baseline (Naive)** — correctness baseline
  - **CPU-only CSR Parallel Version (OpenMP)** — main CPU-only baseline
- **GPU-only Implementations**
  - **GPU-only Dense Baseline (Naive)** — GPU correctness baseline
  - **GPU-only CSR Optimized Kernel** — main GPU-only baseline
- **Hybrid CPU–GPU Implementation**
  - **CPU–GPU Pipeline CSR** — specialized optimization with overlap

Each subsection below describes the design motivation, key algorithmic structure, major optimization techniques, and a code-informed explanation of how each version operates.

4.1 CPU-Only Implementations

4.1.1 CPU-only Dense Baseline (Naive)

The most direct way to compute the product $C = A \times B$ is to evaluate the classical triple loop over all matrix indices. Although this approach does not exploit sparsity, it provides a simple and reliable correctness reference for later comparisons.

The algorithm computes each entry

$$C[i][j] = \sum_{k=0}^{N-1} A[i][k] \cdot B[k][j]$$

by scanning all columns of A and all rows of B. Because every element is processed, the runtime is $O(N^3)$.

This dense baseline is used for:

- verifying the correctness of all CPU sparse and GPU sparse methods
- measuring relative speedup
- ensuring consistency across different data layouts

This version is implemented inside `main.cpp` as the part that computes the dense baseline for correctness checking. Although slow, it plays an essential role by serving as the ground truth for numerical validation.

4.1.2 CPU-only CSR Parallel Version (OpenMP)

This implementation performs sparse–sparse multiplication using CSR-formatted matrices. To accelerate CSR-CSR sparse matrix multiplication, row-level parallelism combined with OpenMP dynamic scheduling is adopted. Since each output row of matrix C is computed independently, the computation can be parallelized without requiring synchronization among threads.

Row-Level Parallelism

If input matrices A and B are stored in CSR form, then for each row i:

- The nonzeros of row i lie in the slice $A.\mathrm{rowPtr}[i] : A.\mathrm{rowPtr}[i+1]-1$.
- A single thread can read all nonzeros of that row without conflict.
- Matrix B is read-only during multiplication and is shared safely across threads.
- The thread produces all contributions to row $C[i,:]$.

This eliminates locking or atomics because no two threads write to the same row of C.

Dynamic Scheduling for Balanced Work

Sparse matrices often contain rows with highly unbalanced numbers of nonzeros. Fixed scheduling would leave threads idle while others process long rows. Using OpenMP:

```
#pragma omp parallel for schedule(dynamic)
```

threads dynamically pull new rows when they finish previous ones, resulting in better utilization and reduced load imbalance.

Local Accumulation Without False Sharing

Each thread stores its partial results in a thread-local hash map, implemented as:

```
unordered_map<int, double> rowMap;
```

Then for every nonzero pair $(A[i,k], B[k,j])$, the thread performs:

```
rowMap[j] += A_ik * B_kj;
```

When a row is finished, the thread sorts the entries and writes the result back to `rowResults[i]`, eliminating the need for synchronization since each thread writes exclusively to its own row index. Because each thread writes to a unique index, false sharing is avoided.

This design appears directly in the CPU code. The CPU CSR implementation serves as the strongest non-GPU baseline and reflects an optimized, realistic multi-core sparse multiplication strategy.

4.2 GPU-Only Implementations

4.2.1 GPU-only Dense Baseline (Naive)

To provide a GPU-side correctness reference, we implemented a straightforward dense matrix multiplication kernel. This approach mirrors the CPU naive version but executes in parallel using CUDA.

Each thread computes one output element $C[i,j]$. The kernel evaluates:

```
for (int k = 0; k < N; ++k)
    C[i*N + j] += A[i*N + k] * B[k*N + j];
```

This version does not exploit sparsity or advanced GPU optimizations. Instead, it serves two critical purposes:

- Correctness Check — all sparse GPU kernels are compared against this dense baseline.
- Speedup Comparison — allows measurement of sparsity-induced acceleration.

The implementation resides in `matmul_base.cu`, invoked through `matmul_base()` in the main program.

4.2.2 GPU-only CSR Optimized Kernel

This method represents the core GPU sparse implementation. Its kernel processes one row of A at a time but incorporates GPU-oriented optimizations to overcome irregular memory access and load imbalance. The kernel adopts several GPU optimization techniques, including persistent threads, read-only cache loads, and shared-memory accumulation. These techniques collectively improve load balancing, memory efficiency, and overall throughput.

Persistent Threads for Dynamic Row Assignment

Sparse matrices often exhibit significantly different numbers of nonzeros per row (nnz), leading to severe load imbalance if each thread block is statically assigned a fixed row. To mitigate this, the kernel employs the persistent threads model.

Rather than assigning each row to a fixed block, the kernel uses a global counter:

```
int row = atomicAdd(&next_row, 1);
```

Each block repeatedly fetches a new row until all rows are processed. This improves occupancy and ensures that blocks handling shorter rows do not sit idle. This dynamic assignment provides effective row-level load balancing, especially for matrices with irregular sparsity patterns

Read-Only Cache Access Using __ldg()

Accessing CSR structures (rowPtr, colIdx, values) is a major performance bottleneck due to their non-coalesced and read-only nature. Therefore, the kernel uses `__ldg()` to load these arrays through the read-only data cache The kernel loads them through:

```
int colA = __ldg(&colIdx[idx]);
float valA = __ldg(&values[idx]);
```

Using the GPU's read-only cache reduces memory access latency, increases cache reuse of frequently accessed rows, and improves memory bandwidth efficiency.

Shared-Memory Accumulation

Each block allocates a shared-memory array sized to the number of columns:

```
extern __shared__ float s_row[];
```

This shared-memory array holds the entire output row $C(i,:)$ during computation, and all partial products generated within the block are accumulated on-chip, which drastically reduces the number of slow global memory updates. Thread-level reductions occur entirely in shared memory, and writes to global memory happen only once per row, improving overall bandwidth efficiency.

Synchronization

Throughout the computation, `__syncthreads()` is used to maintain correctness among threads within the block, particularly after processing each nonzero element of the current row of A. Because all partial sums are accumulated in shared memory instead of using `atomicAdd()`, the reduction is significantly faster and incurs far lower synchronization overhead.

Illustrative Per-Row Processing Example

The following example illustrates how the GPU-only CSR kernel processes one row at a time. For clarity, the diagrams use a dense matrix representation; however, the actual implementation operates on CSR-format matrices, and the conceptual procedure is identical. Each row is processed independently by a block, which loads the nonzero entries of A, fetches the corresponding rows of B, and accumulates all partial results in shared memory before writing the final values to global memory.

When $i = 0$ The threads assigned to row 0 load the nonzero entries of $A[0,:]$ and gather the necessary data from matrix B. All contributions to row $C[0,:]$ are accumulated into the shared-memory buffer.

When $i = 1$ The block clears its shared-memory buffer and repeats the same procedure for row 1. The nonzero structure of the row determines how many threads remain active and how many products are generated.

Eventually, when $i = 3$ Processing continues row by row. When row 3 is processed, the block again accumulates all partial results and writes them back to global memory. At this moment, we obtain the completed first rows of the output matrix C for this example.

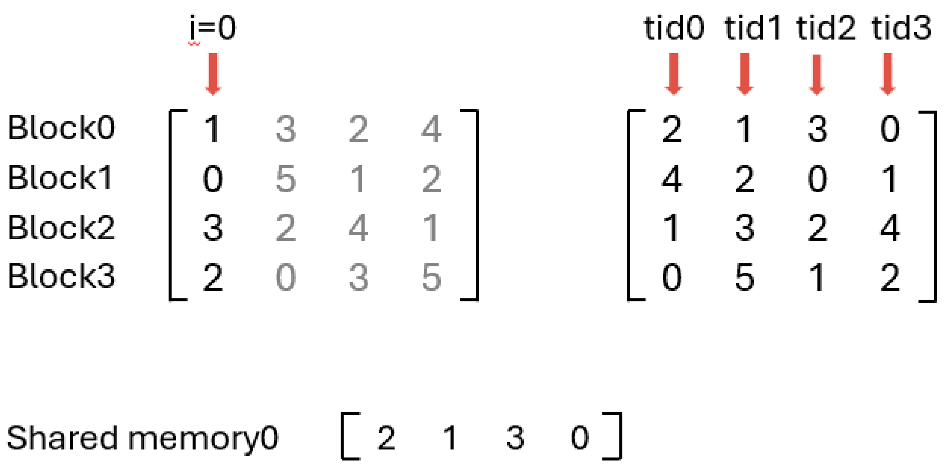


Figure 1: GPU-only CSR kernel processing row $i = 0$. The block loads all nonzero elements in row 0 and accumulates the corresponding partial products.

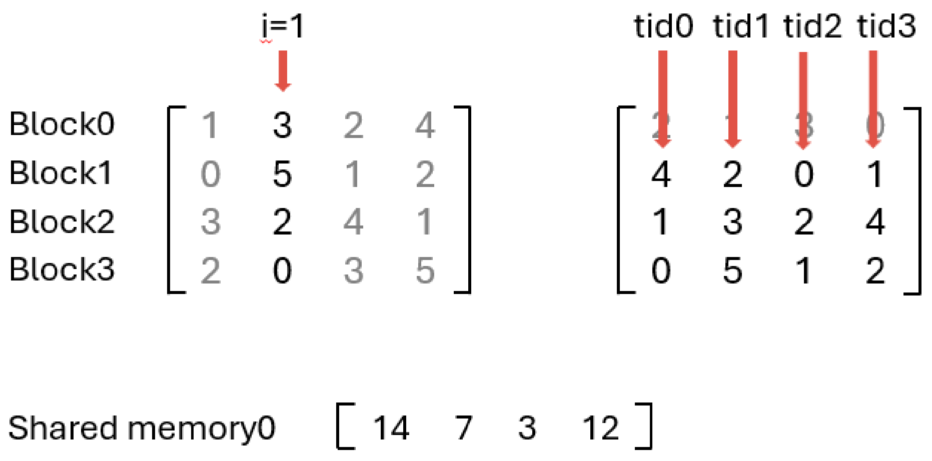


Figure 2: GPU-only CSR kernel processing row $i = 1$. Threads fetch the row's nonzero entries from CSR and accumulate all contributions for the next output row.

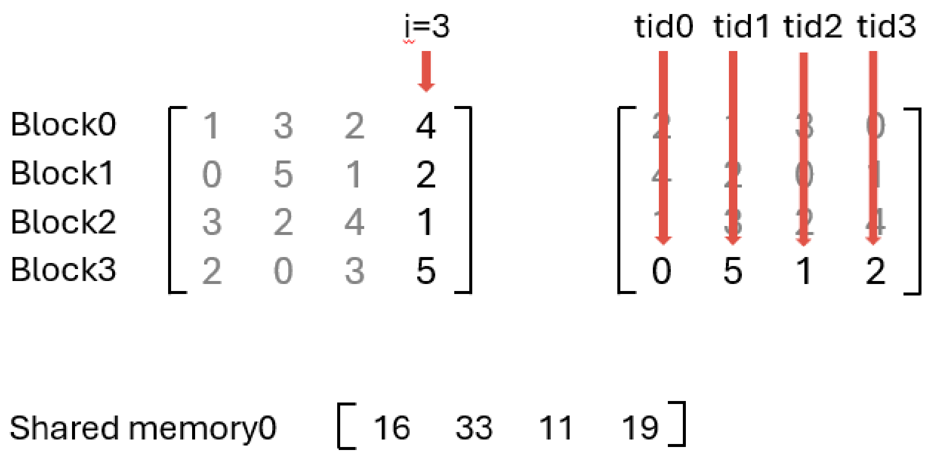


Figure 3: GPU-only CSR kernel processing row $i = 3$. After accumulation, the block writes the final values of this row to global memory.

4.3 Hybrid CPU–GPU Pipeline Implementation

4.3.1 CPU–GPU Pipeline CSR

To further improve throughput, we implemented a double-buffered execution pipeline that overlaps CPU preprocessing with GPU computation.

This method appears in `matmul_sparse_csr_pipeline.cu` and its coordinating logic in the pipeline wrapper. It enables CSR conversion, PCIe data transfer, and GPU computation to occur in parallel, effectively hiding data preparation and transfer latency and improving overall performance.

Overlapping CSR Conversion and GPU Compute

Sparse multiplication requires CSR conversion of blocks of matrix A. Instead of computing all CSR data before launching the kernel, this pipeline overlaps the tasks:

- CPU converts block $t + 1$ into CSR using OpenMP for parallel row-wise processing
- GPU processes block t

Two CPU–GPU buffers are used alternately to allow CSR preparation of the next block while the GPU processes the current one.

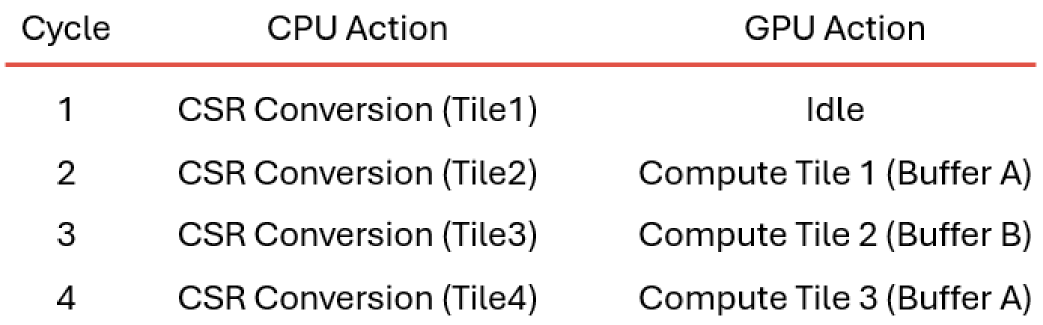
Asynchronous Transfers and CUDA Streams

The pipeline uses:

- `cudaMemcpyAsync`
- separate CUDA streams
- alternating device buffers

These techniques allow CSR blocks to be sent to the GPU while earlier blocks are still being processed.

The following time graph illustrates the execution flow and overlapping behavior of our double-buffer CPU–GPU pipeline design.



Cycle	CPU Action	GPU Action
1	CSR Conversion (Tile1)	Idle
2	CSR Conversion (Tile2)	Compute Tile 1 (Buffer A)
3	CSR Conversion (Tile3)	Compute Tile 2 (Buffer B)
4	CSR Conversion (Tile4)	Compute Tile 3 (Buffer A)

Figure 4: Execution flow of the double-buffer CPU–GPU pipeline. The CPU performs CSR conversion while the GPU computes the previous tile, enabling overlapped operation.

Benefits and Limitations

Although the design reduces idle GPU time, its effectiveness depends on the relative cost of CSR conversion and sparse compute. In practice, CSR generation may still dominate runtime at certain sparsity levels. Nevertheless, the pipeline provides a complete heterogeneous strategy that illustrates the interaction between computation and data movement.

5 Performance Evaluation

This section presents the performance comparison across all five implementations: CPU-only Dense Baseline (Naive), CPU-only CSR Parallel Version (OpenMP), GPU-only Dense Baseline (Naive), GPU-only CSR Optimized, and CPU–GPU Pipeline CSR. We evaluate execution time and speedup under different matrix sizes, sparsity levels, and CPU thread counts.

Our experiments reveal several consistent and important performance trends.

5.1 Overall Comparison Across All Implementations

Across all matrix sizes, every optimized method significantly outperforms the CPU naive baseline. Even the slowest optimized method—GPU naive dense multiplication—achieves substantial reductions in runtime compared with CPU naive. CPU-only CSR Parallel provides another major improvement, demonstrating that sparse-aware computation and multithreading already deliver meaningful acceleration over dense computation.

However, GPU-based approaches achieve an even larger performance advantage. Despite being dense, the GPU naive baseline consistently outperforms CPU-only CSR Parallel due to massive thread-level parallelism and higher memory bandwidth. This makes GPU naive the natural baseline for further GPU-side optimization.

Both GPU-only CSR and CPU–GPU Pipeline CSR outperform GPU naive by a wide margin.

By exploiting sparsity, reducing unnecessary memory traffic, and leveraging persistent-thread and shared-memory optimizations, GPU-only CSR becomes the most effective stand-alone GPU method. The pipeline adds additional improvement by overlapping CPU preprocessing and GPU execution.

These trends, observed across 1024 to 8192 matrix sizes, demonstrate that exploiting sparsity is essential for scaling sparse matrix multiplication.

We evaluate all methods under multiple CPU thread counts (4, 8, 16, 32) and sparsity levels (85%, 90%, 95%) to ensure that our comparisons reflect behavior across a range of parallel workloads.

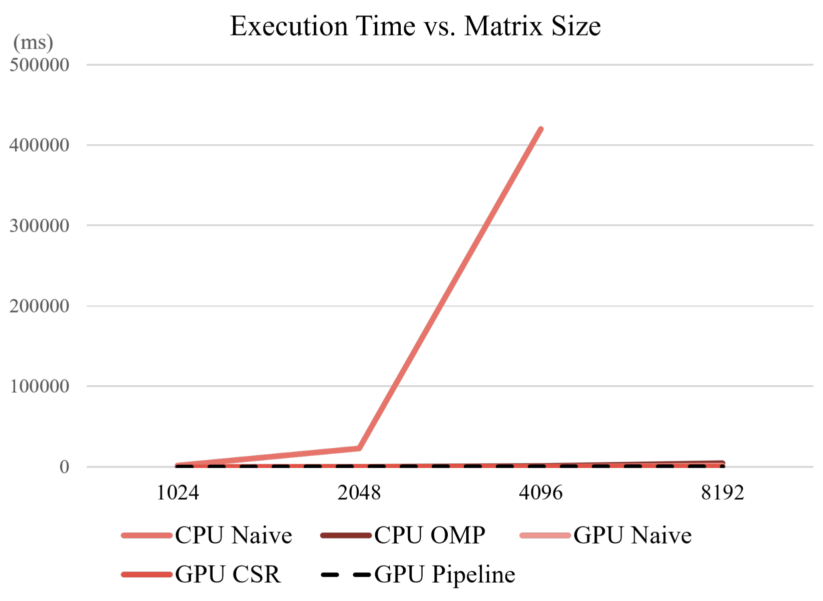


Figure 5: Execution time vs. matrix size for all implementations (full scale).

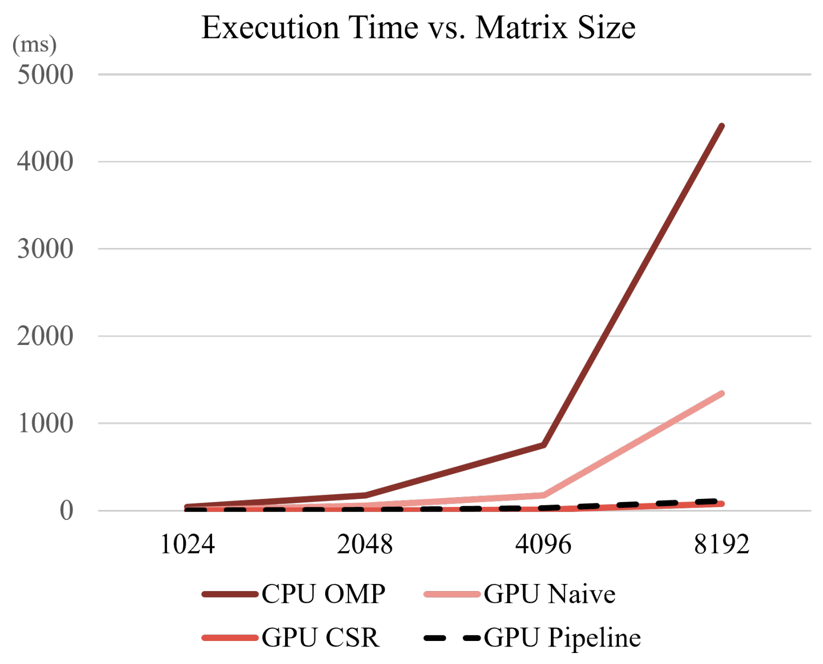


Figure 6: Zoomed-in execution time comparison of CPU-only CSR Parallel, GPU Naive, GPU-only CSR, and GPU Pipeline.

5.2 Detailed GPU-only CSR vs. CPU CSR Parallel Performance

This section analyzes GPU-only CSR and CPU–GPU Pipeline CSR under three sparsity settings: 85%, 90%, and 95%. All comparisons use 32 CPU threads unless otherwise specified.

5.2.1 Sparsity = 85%

With 32 CPU threads, GPU-only CSR and CPU–GPU Pipeline CSR both scale significantly better than CPU-only CSR Parallel Version as matrix size increases.

CPU-only CSR Parallel runtime increases steeply with matrix dimension, reaching more than 17 seconds at size 8192. GPU-only CSR maintains near-linear growth and stays far below 1000 ms even at the largest size. CPU–GPU Pipeline CSR performs similarly to GPU-only CSR, benefiting from overlapped CSR generation.

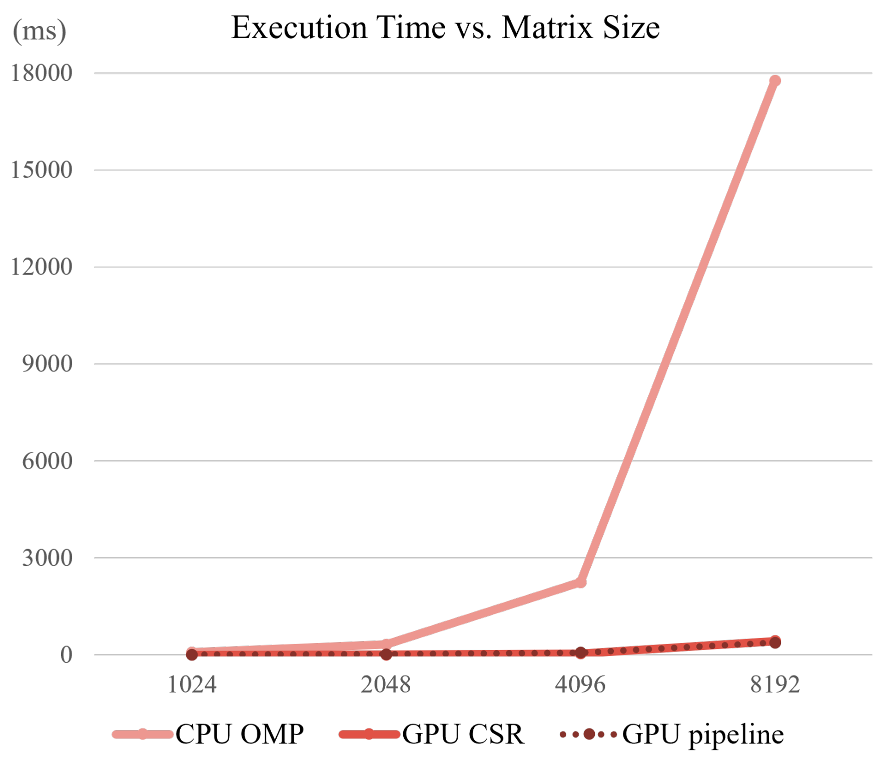


Figure 7: Execution time vs. matrix size for CPU-only CSR Parallel, GPU-only CSR, and CPU–GPU Pipeline CSR at sparsity 85%.

The speedup over CPU-only CSR Parallel is summarized in Table 1. Two observations emerge:

- GPU-only CSR obtains more than 50×–150× speedup depending on thread count.
- Speedup decreases as CPU threads increase, since OMP becomes more competitive.

Table 1: GPU-only CSR vs. CPU-only CSR Parallel speedup comparison (sparsity 85%).

thds / N	1024	2048	4096	8192	Avg
4	156.6	172.4	167.7	126.5	155.8
8	77.6	99.3	94.7	76.3	87.0
16	69.4	77.3	62.7	48.0	64.3
32	66.9	57.6	56.2	46.9	56.9

5.2.2 Sparsity = 90%

At 90% sparsity, GPU-only CSR gains even more advantage over CPU-only CSR Parallel.

CPU-only CSR Parallel surpasses 9 seconds at $N = 8192$, while GPU-only CSR remains under 600 ms. CPU–GPU Pipeline CSR closely follows GPU-only CSR as CPU preprocessing becomes lighter at higher sparsity.

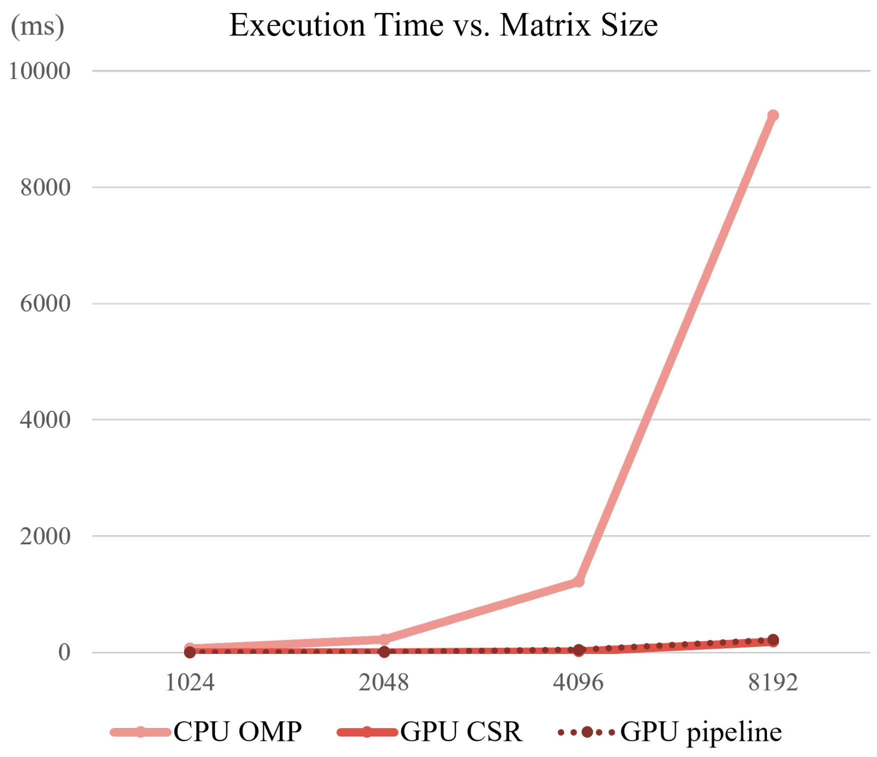


Figure 8: Execution time vs. matrix size for CPU-only CSR Parallel, GPU-only CSR, and CPU–GPU Pipeline CSR at sparsity 90%.

Speedup results are listed in Table 2. Average speedup is higher than the 85% case because GPU benefits more from reduced work.

Table 2: GPU-only CSR vs. CPU-only CSR Parallel speedup comparison (sparsity 90%).

thds / N	1024	2048	4096	8192	Avg
4	149.2	194.6	171.2	169.2	171.0
8	98.4	120.5	105.9	97.7	105.6
16	74.9	79.0	65.5	59.5	69.7
32	74.9	71.0	57.2	49.0	63.0

5.2.3 Sparsity = 95%

At 95% sparsity, workload reduction becomes significant.

CPU-only CSR Parallel still reaches over 4 seconds at $N = 8192$, while GPU-only CSR stays under 500 ms. CPU–GPU Pipeline CSR behaves similarly to GPU-only CSR, benefiting from almost negligible CSR preprocessing at very high sparsity.

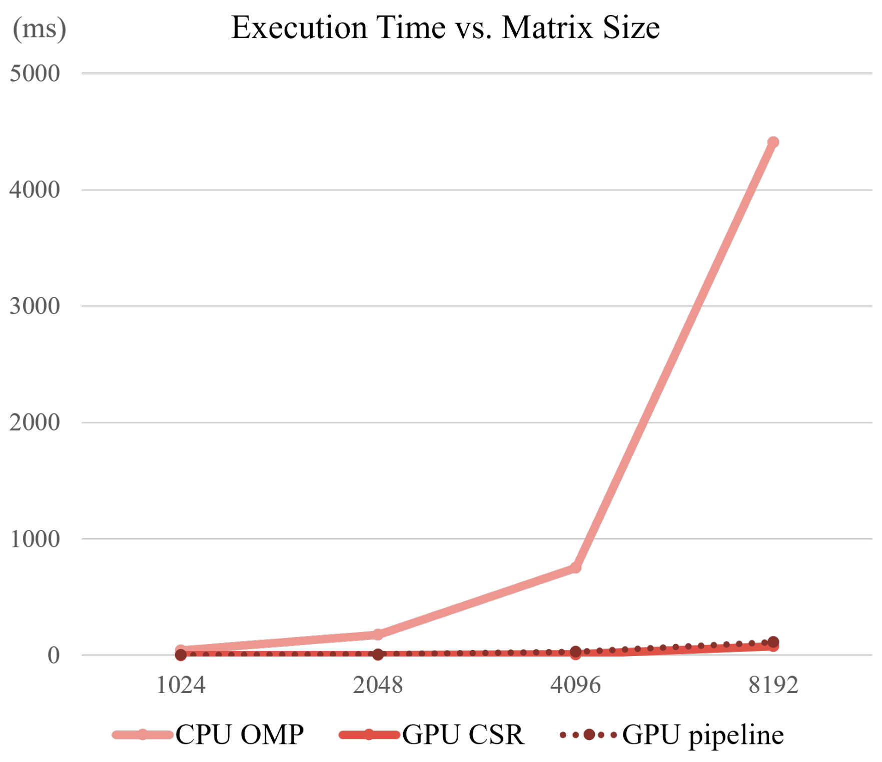


Figure 9: Execution time vs. matrix size for CPU-only CSR Parallel, GPU-only CSR, and CPU–GPU Pipeline CSR at sparsity 95%.

Speedup results are shown in Table 3.

Table 3: GPU-only CSR vs. CPU-only CSR Parallel speedup comparison (sparsity 95%).

thds / N	1024	2048	4096	8192	Avg
4	154.8	187.4	209.7	144.4	154.8
8	104.5	121.8	125.7	90.8	104.5
16	76.1	88.3	87.6	60.3	76.1
32	66.0	77.2	75.2	56.4	66.0

5.3 Speedup Trends Across Sparsity and Matrix Size

The speedup curves demonstrate two strong trends across matrix sizes:

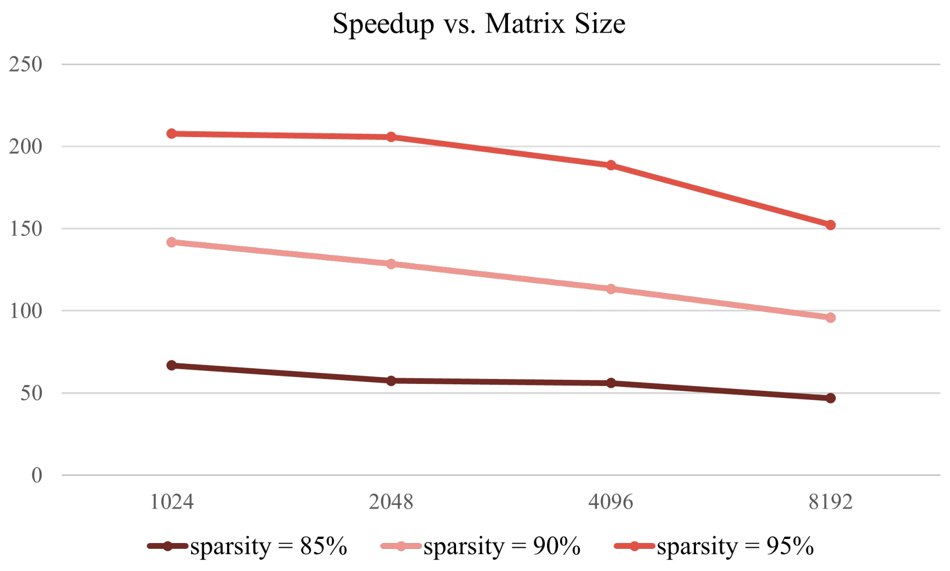


Figure 10: GPU-only CSR speedup over CPU-only CSR Parallel vs. matrix size under three sparsity levels, using 32 CPU threads.

5.3.1 Trend 1: Higher sparsity gives higher speedup

At 95% sparsity, speedup exceeds 150× for small matrices and stays near 200× for $N = 4096$. At 85%, the speedup is much lower (around 60×). This is consistent with reduced memory traffic, fewer nonzeros, and lighter CSR row traversal.

5.3.2 Trend 2: Speedup decreases as matrix size increases

The downward slope across all sparsity levels can be attributed to:

1. **`cudaMalloc/cudaFree` overhead.** As matrix size increases, larger buffers must be allocated and freed, and this overhead becomes a noticeable portion of GPU time.

2. **Scaling host–device memory transfers.** Even with CSR, large matrices require substantial data movement.
3. **Memory-bound shared-memory accumulation.** Large rows pressure shared memory and global memory bandwidth.
4. **CPU-only CSR Parallel scaling better at large N.** Larger workloads saturate CPU threads more effectively, reducing GPU's relative advantage.

These combined factors explain the decline in speedup as matrix size grows.

5.4 Summary of Performance Findings

All optimized methods dramatically outperform the CPU naive dense baseline. GPU-only naive is significantly faster than both CPU-only naive and CPU-only CSR Parallel, making it the correct GPU baseline. GPU-only CSR delivers the best standalone performance, often achieving 50×–200× speedup over CPU-only CSR Parallel. The CPU–GPU Pipeline CSR provides small but consistent improvement on top of GPU-only CSR, especially at moderate sparsity levels. Higher sparsity produces higher GPU speedup. Speedup decreases at very large matrix sizes due to memory-management overheads and scaling behavior of `cudaMalloc`/`cudaFree` and host–device transfers.

Overall, exploiting sparsity and GPU parallelism is essential for high-performance sparse matrix multiplication.

6 Bottleneck Analysis

Although the GPU-only CSR and CPU–GPU Pipeline CSR implementations provide substantial acceleration over CPU-based methods, their performance is not uniform across all problem sizes and sparsity settings. This section analyzes two major bottlenecks observed in our experiments: (1) the limited effectiveness of the CPU–GPU Pipeline CSR, and (2) synchronization overhead in the GPU-only CSR kernel. These bottlenecks explain why certain optimizations do not yield speedup under specific conditions.

6.1 CPU–GPU Pipeline Bottleneck

The CPU–GPU Pipeline aims to overlap CPU-side CSR conversion with GPU sparse matrix multiplication so that both processors remain busy. However, our execution time measurements show that the pipeline often provides little or no speedup compared with standalone GPU-only CSR execution.

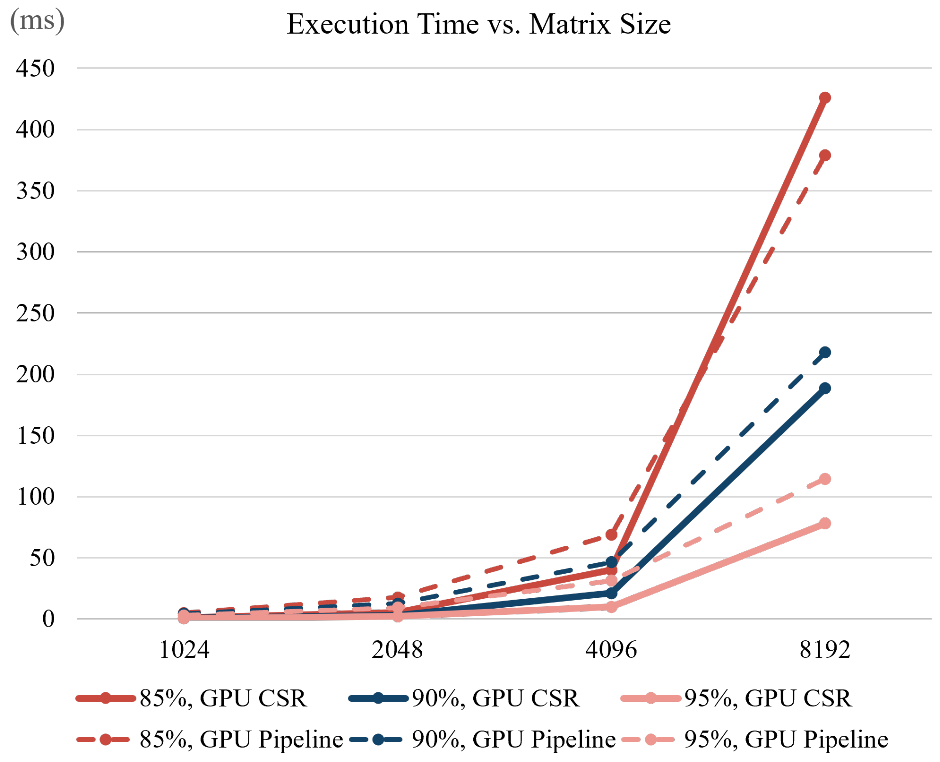


Figure 11: Execution time versus matrix size for GPU-only CSR and CPU–GPU Pipeline CSR under different sparsity levels. The pipeline curve closely follows GPU-only CSR.

Why the pipeline does not speed up in most cases

The primary bottleneck arises from the cost asymmetry between CPU and GPU workloads. Converting dense matrices to CSR on the CPU takes significantly longer than executing the sparse matrix multiplication on the GPU. As a result, the GPU frequently completes its tile before the CPU finishes generating the CSR representation for the next tile. Consequently, the GPU is forced to remain idle, and the intended overlap disappears.

This imbalance is aggravated at smaller matrix sizes. When the matrix is small, several overhead components dominate:

- **Double-buffer management on the CPU**: switching buffers incurs bookkeeping and memory processing overhead.
- **Host-to-device data transfers**: for small matrices, transfer time is comparable to compute time.
- **Kernel launch latency**: the GPU launches a large number of small kernels, preventing high occupancy.
- **Low GPU parallelism**: with few tiles, the GPU processes only one tile at a time, leaving many SMs idle.

Together, these factors make the pipeline ineffective for a wide range of sparsity levels and matrix sizes. In many cases, the pipeline runtime nearly matches the GPU-only CSR runtime, demonstrating that overlap is negligible.

When the pipeline becomes effective

The pipeline provides benefits only when CPU and GPU workloads become comparable in duration. This balance can be achieved by modifying three workload parameters:

1. **Increase matrix size.** When N becomes large (e.g., $N \geq 8192$), GPU computation time grows faster than CSR conversion time. The relative share of kernel-launch overhead and transfer latency also decreases, allowing the GPU to remain busy long enough for true overlap to occur.
2. **Increase the number of CPU threads.** Higher CPU parallelism significantly accelerates CSR generation. With 8 or more CPU threads, row-by-row conversion becomes fast enough to keep up with GPU execution, enabling the intended pipeline behavior.
3. **Reduce sparsity.** Lower sparsity (e.g., 80–85%) increases the number of nonzeros per row, making GPU computation more expensive. This enlarges the GPU workload relative to the CPU's CSR conversion, again helping to balance the two stages.

Empirically, we observe that the pipeline becomes effective only when all three conditions align. In our experiments, the pipeline shows visible speedup primarily when:

$$\#\text{CPU threads} \geq 8, \quad N \geq 8192, \quad \text{sparsity} \leq 85\%.$$

Outside this regime, CPU-side preprocessing dominates the timeline, and the GPU waits for input, nullifying the pipeline's purpose.

6.2 GPU Kernel Bottleneck

GPU-only CSR delivers strong performance on highly sparse matrices, but its speedup diminishes as matrices become less sparse. This behavior is shown in Figure 10, where the speedup curve slopes downward as matrix size increases.

Synchronization overhead in CSR kernels

A key limitation of CSR-based GPU kernels is the need for thread cooperation within each row. Unlike dense matrix multiplication, where every thread can compute independently, CSR SpMM requires:

- row-level accumulation into shared memory,
- thread coordination to traverse variable-length nonzero segments,
- frequent `__syncthreads()` to ensure correctness.

These synchronization barriers reduce warp utilization and prevent the kernel from scaling with the GPU's full compute capability. As sparsity decreases, rows contain more nonzeros, forcing more threads to collaborate and increasing the number of required synchronizations.

Comparison with GPU Naive kernel

Interestingly, GPU Naive (dense) has no such synchronization. Every thread operates independently on fixed-size loops, leading to stable and predictable scaling. When matrices become less sparse, the CSR kernel loses its advantage and may even fall behind GPU Naive due to:

- higher synchronization frequency,
- irregular memory access patterns,
- reduced occupancy caused by variable row lengths.

This explains why GPU-only CSR excels primarily in the high-sparsity regime and why speedup curves drop as sparsity decreases or as matrix size grows.

7 Risk Management

To ensure stable performance across different input sizes and sparsity patterns, we examined several optimization attempts that introduced potential risks or failed to produce measurable improvements. Understanding these risks is essential for selecting robust design choices that generalize across workloads. This section summarizes the major risks encountered during development and explains why certain optimizations were not adopted in the final system.

7.1 Processing Multiple Rows per GPU Block

Optimization Attempt

We explored assigning a single GPU block to process multiple rows of matrix A simultaneously. The expected benefits included:

- higher arithmetic intensity due to shared reuse of partial data,

- fewer kernel launches, reducing launch overhead,
- potentially improved GPU utilization for small tiles.

Observed Outcome

No measurable speedup was observed in practice. In several cases, performance was slightly worse than the baseline CSR kernel.

Risk Analysis

Two major risks were identified:

- **Reduced GPU Occupancy**
  Each block must allocate enough shared memory and registers to handle multiple rows. This significantly increases per-block resource usage, reducing the number of simultaneously resident blocks per SM. Lower occupancy prevents the GPU from hiding memory latency and results in underutilization of compute resources.

  This effect is especially pronounced in sparse matrices, where row lengths vary widely and the block must provision storage for the maximum row length among all rows it handles.
- **Increased Global Memory Traffic**
  Rows processed in a single block often access different regions of matrix B. This reduces spatial locality and prevents coalesced memory access. The GPU must issue more global memory loads than in the one-row-per-block strategy, negating theoretical arithmetic-intensity gains.

7.2 Tiling Requirement for Very Large Matrices ($N \geq 16{,}000$)

Motivation

For extremely large matrices, the CSR or dense representation of B exceeds available GPU memory. To prevent allocation failure, the computation must be partitioned into tiles of B and processed sequentially.

Observed Outcome

Tiling guarantees correctness but leads to significant performance degradation. The tiled implementation consistently performs worse than the non-tiled kernel for smaller matrices.

Risk Analysis

The following risks explain the slowdown introduced by tiling:

- **Repeated Memory Transfers**
  Each tile of B must be copied from CPU to GPU, processed, and then evicted to load the next tile. Many small transfers replace one large transfer, amplifying PCIe overhead.
- **Loss of Temporal Locality**
  Without tiling, rows of A can reuse cached values from B. After tiling, B is repeatedly reloaded, eliminating temporal reuse opportunities.
- **Higher Synchronization and Launch Overhead**
  Tiling breaks the computation into many small kernels, increasing:
  - kernel launch latency,
  - CPU–GPU synchronization,
  - bookkeeping overhead.
- **Underutilization of GPU Resources**
  Large matrices normally maximize SM occupancy, but tiling reduces the parallel workload per kernel. Each tile becomes too small to saturate GPU resources, lowering overall throughput.

7.3 Additional Risks Identified During Development

Beyond the two major risks above, several additional risks were observed in experiments and profiling.

- **Thread Divergence from Irregular CSR Row Lengths**
  CSR row lengths vary significantly. When a warp processes multiple rows or encounters rows with long NNZ segments:
  - some threads finish early and become idle,
  - others continue processing long segments,
  - warp efficiency drops substantially.
- **Shared-Memory Pressure for Dense-Like Rows**
  When sparsity decreases (e.g., ¡ 80%), some rows contain thousands of nonzeros, which risks:
  - exceeding shared-memory limits,
  - spills into global memory,
  - severe bank conflicts.

- **High Cost of `cudaMalloc` / `cudaFree` in Loops**
  Profiling shows that repeated device memory allocations dominate runtime when N is small. Allocations inside pipeline loops introduce additional jitter and reduce available overlap.

- **Non-Coalesced Accesses to B**
  Due to irregular column indices in CSR, threads within the same warp may load from unrelated addresses. This causes:
  - reduced effective memory bandwidth,
  - increased DRAM transactions,
  - lower overall throughput.

- **Pipeline Dead-Time Accumulation**
  Even when overlap exists, slight imbalances between CPU and GPU execution accumulate over many iterations. These gaps create dead time where one processor waits for the other, which degrades the expected benefit of pipelining.

8 Conclusion

This project provides a comprehensive investigation of sparse matrix multiplication across CPU-only, GPU-only, and hybrid CPU–GPU execution models. By implementing five representative methods on top of a unified CSR-based workflow, we isolated the performance impact of algorithmic structure, parallel work distribution, memory behavior, and hardware characteristics. Our results demonstrate that exploiting sparsity is far more important than relying on raw computational throughput: even a simple CSR-based CPU implementation outperforms dense computation by a large margin. GPUs extend this benefit substantially, with the optimized CSR kernel delivering up to two orders of magnitude speedup over multithreaded CPU CSR, particularly at high sparsity levels where shared-memory accumulation and persistent-thread scheduling are most effective.

The study also reveals fundamental limitations. CSR irregularity introduces synchronization overhead that grows with matrix size, while non-coalesced accesses to matrix B constrain achievable bandwidth. The hybrid CPU–GPU pipeline, though conceptually appealing, is limited by the high cost of CSR conversion on the CPU, which often outweighs GPU computation and suppresses overlap. Real performance gains appear only when matrix sizes are large, sparsity is moderate, and the CPU is sufficiently parallelized.

Overall, the results highlight that sparse matrix multiplication requires careful co-design of data structures, memory access patterns, and parallel scheduling. Future work may explore alternative sparse formats, adaptive tiling, warp-level aggregation, and preprocessing strategies that reduce conversion cost or shift it onto the GPU. These directions offer promising opportunities to improve scalability and efficiency for real-world large-scale sparse workloads.

References

[1] M. Deveci, C. Trott, and S. Rajamanickam, “Multi-threaded Sparse Matrix-Matrix Multiplication for Many-Core and GPU Architectures,” Sandia National Laboratories, 2018.