High-Performance Sparse-Dense Matrix Multiplication with CUDA

1st Amir Hastor
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
ahastor1@etf.unsa.ba

2nd Asmir Prašović
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
aprasovic1@etf.unsa.ba

3rd Berin Mašović
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
bmasovic1@etf.unsa.ba

4th Elhattab Yahia Aissa
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
eyahiaaiss1@etf.unsa.ba

5th Nedim Džindo
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
ndzindo2@etf.unsa.ba

***Abstract*—Sparse-dense matrix multiplication is a fundamental operation in scientific computing, machine learning, and graph analytics, where efficient GPU implementations are critical for high performance. In this work, we introduce `casperSPARSE`, a CUDA-based framework optimized for multiplying large sparse matrices in CSR format with dense matrices. The implementation is compared against NVIDIA's highly optimized cuSPARSE library across a wide range of matrix dimensions, row counts, and sparsity levels. Performance is evaluated in terms of execution time, memory bandwidth, GFLOPS, and computational accuracy.**

Experiments show that `casperSPARSE` consistently achieves speedups for low-to-moderate sparsity matrices, while maintaining high numerical precision. Detailed scaling studies reveal that the kernel is memory-bound, achieving up to 95% memory throughput, whereas compute utilization remains limited. In contrast, cuSPARSE demonstrates higher arithmetic throughput but underutilizes available memory bandwidth. The results provide insights into the interplay between sparsity, matrix size, and GPU resource utilization, highlighting scenarios where `casperSPARSE` offers advantages in execution time and bandwidth efficiency. These findings offer practical guidance for selecting appropriate sparse-dense multiplication strategies in GPU-accelerated applications.

***Index Terms*—Sparse-dense multiplication, CUDA, GPU computing, cuSPARSE, performance analysis, memory bandwidth, matrix sparsity**

I. Introduction

Sparse–dense matrix multiplication (SpMM) is a fundamental operation in many high-performance computing and data-intensive applications, including scientific simulations, graph analytics, machine learning, and numerical linear algebra. In SpMM, a sparse matrix $\mathbf{A} \in \mathbb{R}^{M\times K}$ is multiplied with a dense matrix $\mathbf{B} \in \mathbb{R}^{K\times N}$ to produce a dense output matrix $\mathbf{C} \in \mathbb{R}^{M\times N}$.

Modern GPU architectures provide massive parallelism and high memory bandwidth, making them well suited for dense linear algebra. However, sparse computations remain challenging due to irregular memory access patterns, load imbalance, and limited data reuse. These challenges become more pronounced in SpMM, where both sparsity in $\mathbf{A}$ and dense access to $\mathbf{B}$ must be handled efficiently.

NVIDIA's cuSPARSE library provides highly optimized implementations of sparse linear algebra routines, including SpMM [4]. While cuSPARSE delivers strong performance across a wide range of sparsity patterns, its general-purpose design may not fully exploit problem-specific characteristics such as fixed numbers of nonzero elements per row or vectorized dense matrix layouts.

In this paper, we present `casperSPARSE`, a custom CUDA kernel for high-performance sparse–dense matrix multiplication. The proposed approach is based on a warp-level decomposition, where each warp processes one row of the sparse matrix stored in Compressed Sparse Row (CSR) format. The kernel leverages shared memory, warp-synchronous programming, and vectorized `float4` operations to improve memory efficiency and computational throughput.

This work makes several key contributions to the field of GPU-accelerated sparse-dense matrix multiplication. First, we design and implement a custom `warp-centric SpMM CUDA` kernel, carefully optimized for `CSR` matrices with a near-regular distribution of nonzeros per row, achieving efficient thread utilization and high memory throughput. Second, we introduce an optimized data layout and computation strategy that exploits vectorized accesses to `dense matrices`, reducing memory latency and improving bandwidth utilization. Finally, we present a comprehensive performance evaluation against NVIDIA's highly optimized `cuSPARSE SpMM` routine, analyzing execution time, computational throughput, and memory bandwidth. Our results demonstrate that the proposed approach can significantly accelerate sparse-dense multiplication for matrices with regular sparsity patterns, offering both high efficiency and scalability.

II. Problem Formulation and Data Representation

We consider the sparse–dense matrix multiplication problem defined as

$$\mathbf{C} = \mathbf{A} \times \mathbf{B}, \tag{1}$$

where $\mathbf{A} \in \mathbb{R}^{M \times K}$ is a sparse matrix, $\mathbf{B} \in \mathbb{R}^{K \times N}$ is a dense matrix, and $\mathbf{C} \in \mathbb{R}^{M \times N}$ is the resulting dense matrix.

A. Sparse Matrix Format

The sparse matrix $\mathbf{A}$ is stored using the Compressed Sparse Row (CSR) format.

The `value` and `colidx` arrays, each of length nnz, store the values of the non-zero elements and their corresponding column indices in the order in which they appear across the rows. The `rowidx` array, of length $M + 1$, indicates where each matrix row starts and ends within the `value` and `colidx` arrays. Specifically, $\texttt{rowidx}[i]$ represents the total number of non-zero elements in the first i rows [1]. Assuming row indices start from 0, the number of non-zero elements in row i can be computed as:

$$\texttt{rowidx}[i+1] - \texttt{rowidx}[i]. \tag{2}$$

B. Dense Matrix Layout

The dense input matrix $\mathbf{B}$ and output matrix $\mathbf{C}$ are stored in row-major order. To improve memory throughput and arithmetic intensity, the dense matrices are accessed in a vectorized manner using `float4` types. Consequently, the number of columns N is constrained to be divisible by four.

C. Computation Model

Each row of the sparse matrix is multiplied with all columns of the dense matrix. This operation involves accumulating products of sparse values from $\mathbf{A}$ with corresponding rows of $\mathbf{B}$. Efficient handling of this access pattern is critical for achieving high performance on GPUs.

III. casperSPARSE Kernel Design

This section describes the design principles and implementation details of the proposed `casperSPARSE_kernel`.

A. Warp-Level Parallelization

The kernel adopts a warp-centric execution model, where each warp is responsible for computing one row of the output matrix $\mathbf{C}$. This design maps well to the CSR format, as each row can be processed independently.

Within a warp, individual threads cooperate to load sparse matrix elements and to compute multiple output columns. Warp-synchronous programming is used to avoid unnecessary synchronization overhead.

B. Shared Memory Utilization

Sparse matrix column indices and values are loaded into shared memory in chunks of up to 32 elements. This reduces redundant global memory accesses and allows fast reuse of sparse values across threads in the warp.

Shared memory arrays are statically allocated per block and partitioned so that each warp accesses a distinct region, eliminating bank conflicts and ensuring correctness.

C. Vectorized Dense Matrix Access

To increase computational efficiency, dense matrix elements are loaded using `float4` vector types. Each thread processes one vector of four columns at a time, accumulating partial results in vector registers.

This approach improves memory coalescing and increases arithmetic intensity, leading to better utilization of GPU compute resources.

D. Kernel Execution Flow

For each assigned row, the kernel processes the nonzero elements in chunks. In each chunk, sparse values and column indices are first staged in shared memory, after which the warp performs vectorized multiply–accumulate updates over the corresponding dense matrix tiles. Finally, the accumulated results are written back to the output matrix $\mathbf{C}$.

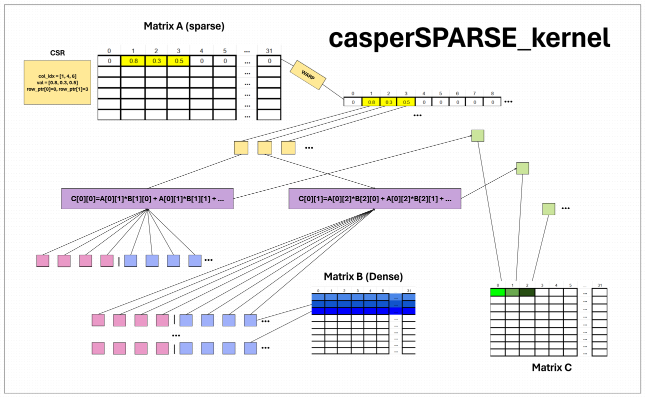
Matrix A (sparse)
casperSPARSE_kernel
CSR
col_idx = [1, 4, 6]
val = [0.8, 0.3, 0.5]
row_ptr[0]=0, row_ptr[1]=3
WARP
C[0][0]=A[0][1]*B[1][0] + A[0][1]*B[1][1] + ...
C[0][1]=A[0][2]*B[2][0] + A[0][2]*B[2][1] + ...
Matrix B (Dense)
Matrix C

Fig. 1. Warp-centric execution flow of `casperSPARSE_kernel` for computing one output row from CSR data and dense matrix tiles.

The kernel exploits warp-level parallelism and `float4` vectorization for the output matrix elements. This approach follows the general idea of warp-centric SpMM designs such as GE-SpMM [2].

IV. Implementation

This section describes the complete implementation of the proposed `casperSPARSE` framework, including data generation, kernel execution, baseline comparison using cuSPARSE, and performance measurement.

A. Overall Application Structure

The application is implemented in CUDA C++ and consists of three main stages: (i) GPU-based data generation, (ii) sparse–dense matrix multiplication using both cuSPARSE and the proposed `casperSPARSE_kernel`, and (iii) result verification and performance analysis.

All computations are performed entirely on the GPU to avoid unnecessary host–device transfers. The host code is responsible only for kernel launches, memory management, timing, and result validation.

B. GPU-Based Matrix Generation

To ensure reproducibility and eliminate CPU-side overhead, both sparse and dense matrices are generated directly on the GPU using dedicated CUDA kernels.

1) CSR Row Pointer Generation: The `row_ptr` array of the CSR representation is generated by the `generator_matrica_row_ptr_kernel`. To simulate more realistic sparse patterns while maintaining high performance, the number of non-zero elements for each row i is randomly sampled from a uniform distribution:

$$nnz_i \in \text{unif}\left(\frac{1}{2} \cdot \frac{K \cdot S}{100}, \frac{K \cdot S}{100}\right) \tag{3}$$

where S is the target sparsity percentage. After the individual row counts are determined, a parallel prefix sum kernel is executed to compute the final `row_ptr` values. This approach introduces a controlled degree of load imbalance, allowing for a more robust evaluation of the kernel's performance across slightly irregular sparsity patterns.

2) Sparse Matrix Value and Index Generation: The column indices and values of the sparse matrix are generated using the `generator_matrica_sparse_kernel`. Each thread initializes one nonzero element using a lightweight linear congruential generator (LCG). The column indices are uniformly distributed in the range $[0, K)$, while the values are generated as single-precision floating-point numbers in the interval $(0, 1)$.

This method provides sufficient randomness for benchmarking while keeping the generation overhead minimal and fully parallel.

3) Dense Matrix Generation: The dense input matrix $\mathbf{B}$ is generated by the `generator_matrica_dense_kernel`. Each thread initializes one element of the matrix using the same LCG-based random number generator. The matrix is stored in row-major order to match the expected memory layout of both the custom kernel and the cuSPARSE baseline.

C. Custom casperSPARSE Kernel Implementation

The core of the proposed approach is the `casperSPARSE_kernel`, which implements a warp-centric SpMM computation.

1) Thread and Warp Mapping: Each warp is assigned to compute one row of the output matrix $\mathbf{C}$. The global warp index is derived from the block and thread indices, and threads within a warp cooperate to process multiple columns of the dense matrix in a vectorized manner. The kernel is designed for a fixed warp size of 32 threads and employs a block size of `THREADS_PER_BLOCK`, allowing multiple independent warps per block to execute concurrently. This warp-per-row strategy enables efficient utilization of GPU resources for sparse matrix–dense matrix multiplication, following the design principles presented in [3].

2) Shared Memory Organization: To reduce global memory traffic, sparse matrix column indices and values are loaded into shared memory in chunks of up to 32 nonzero elements. Each warp uses a private region of shared memory, preventing interference between warps within the same block.

Warp-level synchronization primitives are used to ensure that all threads have completed loading before proceeding with computation.

3) Vectorized Computation: The dense matrix $\mathbf{B}$ is accessed using `float4` vector loads, allowing each thread to process four output columns simultaneously. Accumulators are maintained in vector registers to minimize memory access and maximize instruction-level parallelism.

For each nonzero element in the sparse row, the kernel performs a vectorized multiply–accumulate operation:

$$\mathbf{acc} \mathrel{+}= a_{ij} \times \mathbf{B}_j, \tag{4}$$

where **acc** is a four-element vector accumulator.

4) Output Storage: After processing all nonzero elements for the assigned row and column range, each thread writes its accumulated `float4` result to the corresponding location in the output matrix $\mathbf{C}$.

```
casperSPARSE_kernel (SpMM: $C = A_{\text{CSR}} \cdot B$)
Input: $M$, $N_{vec}$, $row\_ptr$, $col\_idx$, $val$, $B$, $C$
Output: Computes dense matrix $C$ from sparse CSR matrix $A$ and dense matrix $B$
warpId ← $blockIdx \cdot \left(\frac{blockDim}{32}\right) + \left\lfloor \frac{threadIdx}{32} \right\rfloor$
laneId ← $threadIdx \bmod 32$
if $warpId \geq M$ then
    return
end if
start ← $row\_ptr[warpId]$
end ← $row\_ptr[warpId + 1]$
Shared: $sh\_col[THREADS\_PER\_BLOCK]$, $sh\_val[THREADS\_PER\_BLOCK]$
// shared memory cache per warp
warpBase ← $\left\lfloor \frac{threadIdx}{32} \right\rfloor \cdot 32$
scol ← $\&sh\_col[warpBase]$
sval ← $\&sh\_val[warpBase]$
for $col\_offset$ ← 0 to $N_{vec} - 1$ step 32 do
    $current\_col$ ← $col\_offset + laneId$
    $active$ ← $(current\_col < N_{vec})$
    $acc$ ← $(0, 0, 0, 0)$                              // float4 accumulator
    for $i$ ← $start$ to $end - 1$ step 32 do
        $t$ ← $\min(32, end - i)$              // number of nonzeros in this chunk
        if $laneId < t$ then
            $scol[laneId]$ ← $col\_idx[i + laneId]$
            $sval[laneId]$ ← $val[i + laneId]$
        end if
        syncwarp()                             // ensure shared memory is filled
        if $active$ then
            for $k$ ← 0 to $t - 1$ do
                $a$ ← $sval[k]$
                $colA$ ← $scol[k]$
                $b$ ← $B[colA \cdot N_{vec} + current\_col]$
                $acc$ ← $acc + a \cdot b$          // 4 multiplications + 4 additions
            end for
        end if
        syncwarp()                                      // before next chunk
    end for
    if $active$ then
        $C[warpId \cdot N_{vec} + current\_col]$ ← $acc$
    end if
end for
```

Fig. 2. Pseudocode of the proposed `casperSPARSE_kernel` (warp-per-row SpMM).

For comparison, the application integrates NVIDIA cuSPARSE using the `cusparseSpMM` routine. Sparse matrices are provided in CSR format, which is illustrated in Fig. 3, while dense matrices are represented using cuSPARSE dense matrix descriptors.

The implementation performs an initial preprocessing step, during which cuSPARSE determines the required auxiliary

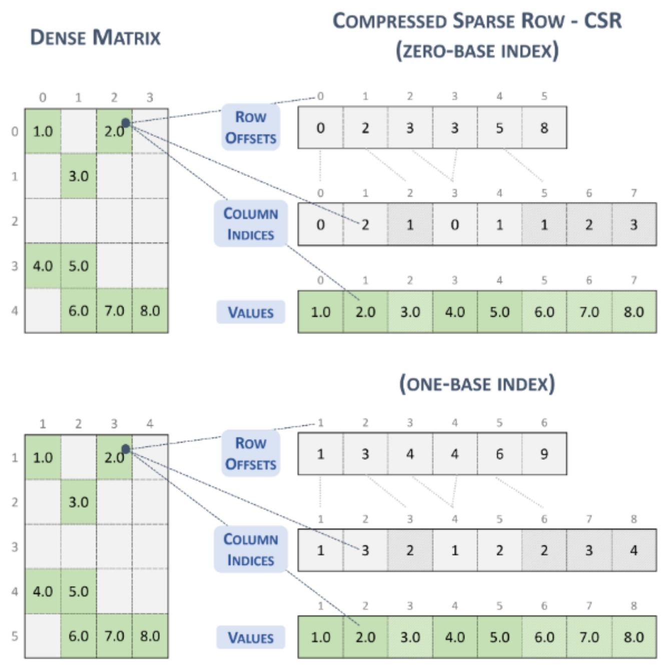

Fig. 3. Compressed Sparse Row (CSR) matrix representation used in our experiments and by cuSPARSE.

buffer size and performs internal optimizations. This buffer is reused across all subsequent iterations to ensure fair performance comparisons.

D. Performance Measurement

Kernel execution times are measured using CUDA events to provide accurate millisecond-level timing. Each experiment consists of a number of warm-up iterations followed by several timed iterations. The minimum, maximum, and average execution times are computed to account for runtime variability.

In addition to execution time, the application computes achieved floating-point throughput in GFLOPS and effective memory bandwidth in GB/s based on the total number of floating-point operations and memory transfers.

V. Performance Evaluation and Comparison

In this section, the performance characteristics of the proposed *casperSPARSE* implementation are evaluated and compared against NVIDIA cuSPARSE. The analysis focuses on scalability with respect to matrix shape and workload growth, as well as achieved throughput and memory efficiency.

The benchmarks were performed on an NVIDIA GeForce RTX 3050 Ti GPU (2560 CUDA cores, 4 GB GDDR6, 192 GB/s peak bandwidth). All experiments use sparse matrices stored in CSR format with a variable number of non-zero elements per row, sampled within a range to match the target sparsity while testing the kernel's handling of minor load imbalances.

wide matrices (large dense dimension N), approximately square matrices (balanced growth of M, K, and N), and tall matrices (large number of rows M).

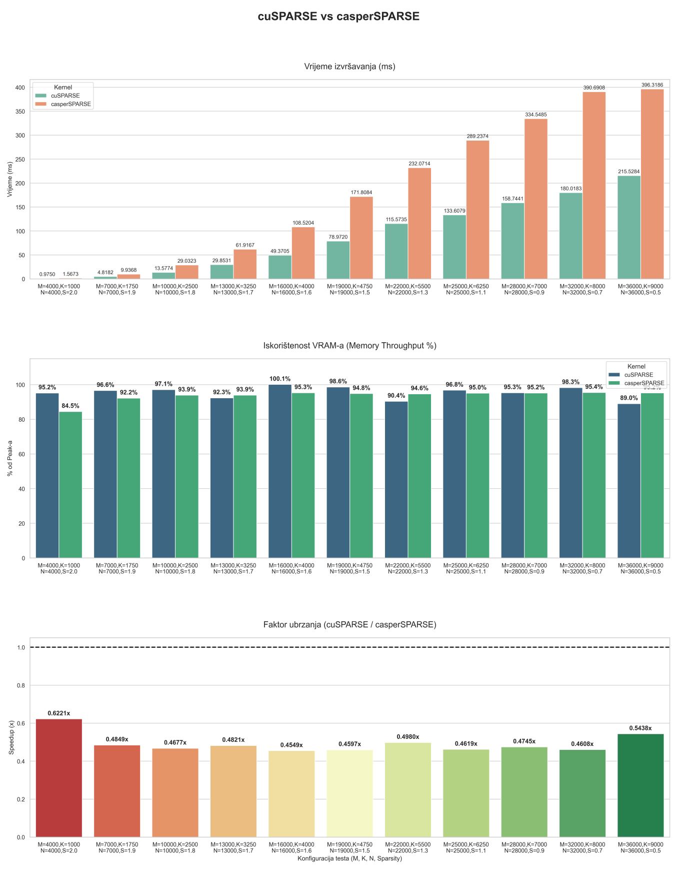

Fig. 4. cuSPARSE vs casperSPARSE for wide matrix configurations (large N).

From an architectural perspective, wide matrices increase the number of memory accesses to the dense matrix $\mathbf{B}$, making the computation strongly memory-bound. cuSPARSE appears to benefit from more efficient caching and scheduling, reducing memory stalls and achieving lower runtime.

For approximately square matrices, both kernels experience balanced workload distribution, which improves occupancy and reduces load imbalance. The custom kernel demonstrates stable scaling due to predictable memory access patterns and efficient vectorized loads.

Tall matrices highlight the advantage of the warp-per-row execution model, as a large number of rows enables high parallelism and effective latency hiding. This confirms that the proposed kernel performs particularly well when row-level parallelism dominates the workload.

For each scenario, the plots show execution time, achieved VRAM throughput, and the speedup factor (cuSPARSE / casperSPARSE), where values below 1 indicate that cuSPARSE is faster.

A. Wide Matrix Scenario

For wide matrices, cuSPARSE consistently achieves lower execution times across all tested configurations. The speedup factor stays below 1 (roughly in the 0.55–0.90 range), meaning the custom kernel is slower in this regime. Both kernels still

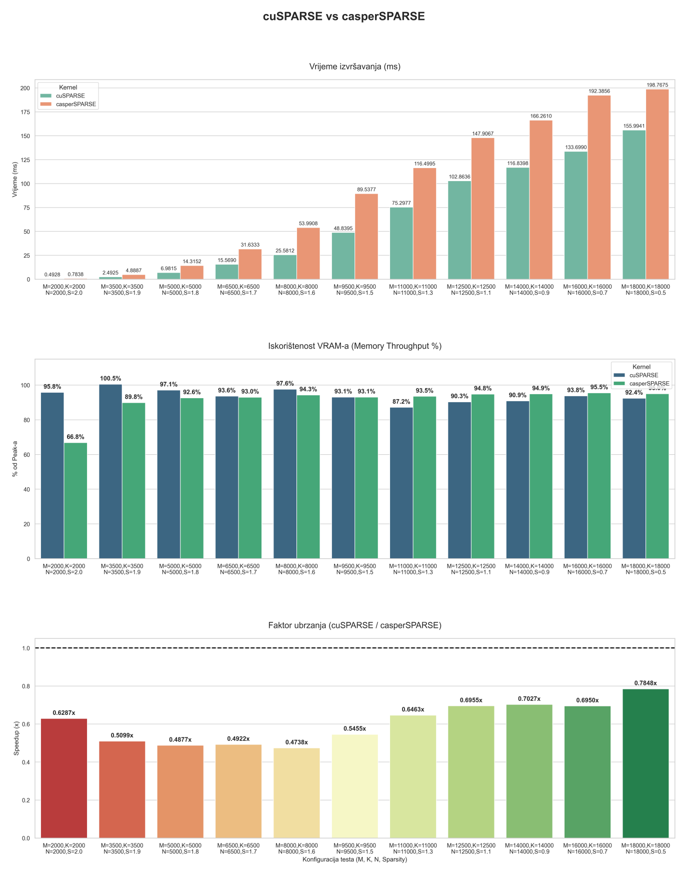

Fig. 5. cuSPARSE vs casperSPARSE for approximately square matrix configurations.

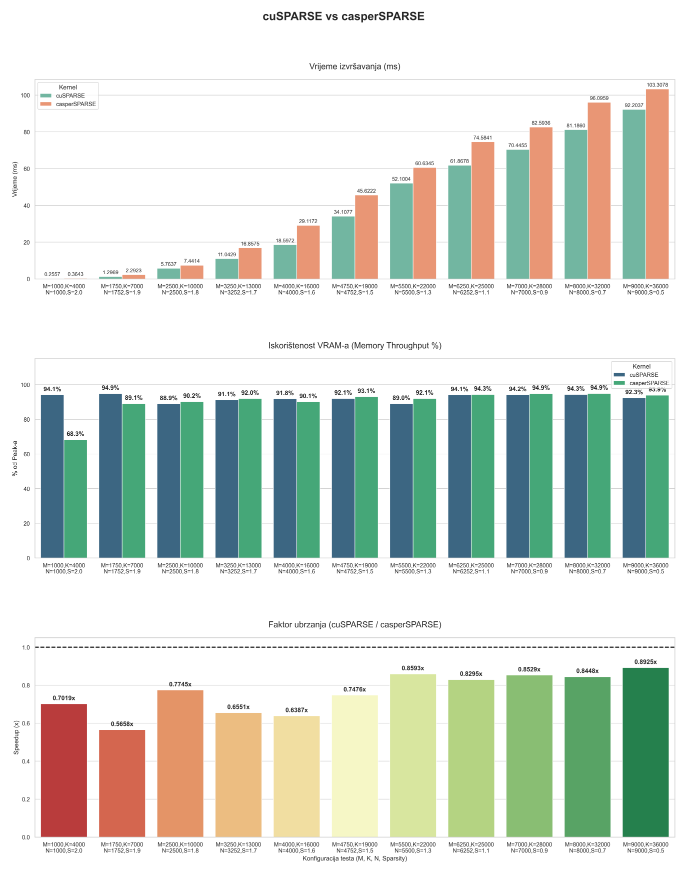

Fig. 6. cuSPARSE vs casperSPARSE for tall matrix configurations (large M).

reach high VRAM throughput (often above 90%), indicating a strongly memory-bound workload.

The observed runtime gap becomes more visible as N grows, suggesting that cuSPARSE benefits from more advanced internal blocking and better reuse of dense matrix tiles for wide outputs, while `casperSPARSE` performs mainly streaming accesses with limited reuse across threads.

B. Approximately Square Matrix Scenario

For balanced (approximately square) matrices, both implementations scale smoothly as the workload increases. The `casperSPARSE` kernel sustains high VRAM throughput (typically around 90%–95%), confirming efficient streaming of CSR data and dense loads.

Although cuSPARSE remains faster in execution time, the relative gap is generally smaller than in the wide-matrix case, which indicates that the warp-per-row strategy and `float4` vectorization in `casperSPARSE` match this shape better.

C. Tall Matrix Scenario

For tall matrices, both kernels scale close to linearly with problem size due to the large number of independent rows. This case maps well to the warp-per-row design in `casperSPARSE`, since it exposes abundant parallelism and helps maintain good occupancy.

cuSPARSE still delivers lower execution times overall, but the speedup factor typically improves compared to the wide scenario, showing that the custom kernel benefits from increased work distribution when M dominates.

D. Summary of Observations

Overall, the experimental results confirm that sparse–dense matrix multiplication on modern GPUs is predominantly memory-bound. Both cuSPARSE and the proposed `casperSPARSE` kernel achieve high VRAM throughput, indicating that performance is largely limited by memory bandwidth rather than compute capacity.

Across all tested scenarios, cuSPARSE consistently achieves lower execution times due to its highly optimized internal kernels, advanced scheduling, and more effective reuse of data. However, the custom kernel demonstrates strong scaling behavior and maintains competitive performance, particularly for balanced and tall matrix configurations where row-level parallelism is high.

The experiments also show that matrix shape plays a crucial role in performance. Wide matrices amplify memory pressure due to increased dense matrix accesses, while larger workloads improve GPU occupancy and reduce the relative performance gap between implementations.

These findings highlight that custom kernels can provide valuable performance insights and achieve high efficiency when tailored to specific sparsity patterns, while general-purpose libraries remain advantageous for broader workloads.

References

[1] A. Mehrabi et al., "Learning Sparse Matrix Row Permutations for Efficient SpMM on GPU Architectures," in ISPASS, 2021.

[2] G. Huang, G. Dai, Y. Wang, and H. Yang, "GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks," in Proc. SC20: Int. Conf. for High Performance Computing, Networking, Storage and Analysis, 2020.

[3] L. Xiang, O. Asudeh, G. Sabin, A. Sukumaran-Rajam, and P. Sadayappan, "cuTeSpMM: Accelerating Sparse-Dense Matrix Multiplication Using GPU Tensor Cores," arXiv preprint, arXiv:2504.06443, 2025.

[4] NVIDIA Corporation, cuSPARSE: CUDA Sparse Matrix Library, NVIDIA Developer Documentation, 2024.