# Deliverable report

*Alberto Messa: Mat: 258855,* `alberto.messa@studenti.unitn.it`*, GitRepo:* `https://github.com/MessaAlberto/GPU-Computing-2025-258855.git`

***Abstract*—The sparse matrix-dense vector multiplication (SpMV) is a common linear algebra operation involving a sparse matrix and a dense vector. SpMV is widely used in many real-world applications such as scientific computing, network analysis and notably machine learning, where sparse and large datasets are common. One widely used format for sparse matrices is the Compressed Sparse Row (CSR) format, which allows us to efficiently store and access non-zero (NNZ) elements in the matrix.**

**This deliverable discusses how to effectively accelerate SpMV computations on GPUs by leveraging CUDA programming, focusing on parallelization strategies guided by the GPU's architecture analysis. In this project we implemented several CUDA kernels, which are compared and analyzed. Their performance is evaluated based on key parallelization characteristics such as memory access patterns, occupancy, coalescence and scalability.**

## I. Introduction

The project provides a couple of C implementations running on the CPU—a naive version and a cache-optimise one—and multiple CUDA kernels executed on the GPU, all designed to compute SpMV. For all codes, the process begins by reading a sparse matrix from a file. The retrieved matrix is in Coordinate (COO) format, which must be converted into CSR format.

The difference between these formats lies in how information is stored: in COO format, the row and column indices are explicitly stored for each NNZ element, resulting the size of both row and column arrays of size NNZ. While, the CSR format uses a row pointer array of length equals to the number of rows plus one. The number of elements in a row can be determined by subtracting two consecutive entries in the row pointer array. This representation is possible because CSR data is sorted by rows and then by columns.

Next, a dense vector is generated with the same seed to ensure reproducibility. Then, a number of warm-up cycles are performed before the actual benchmarking starts. After this phase, time measurements are recorded and performance statistics are generated.

Although the SpMV is relatively simple—each NNZ element in the matrix calls a multiplication and a summation—the main challenges lie in memory management. SpMV is a memory-bound problem, where the performance are strongly influenced by how efficiently memory is accessed. In fact in this deliverable, we will give special attention about the memory access patterns and strategies to minimize them during computation. The benchmarking phase reports the average execution time of the SpMV computation. From these results, key performance metrics such us memory bandwidth and achieved GFlops are derived and analyzed.

## II. Problem Statement

SpMV is challenging on GPUs because of irregular memory accesses caused by the sparse data structure.

### A. Storage Format

This work uses the CSR format, which stores the matrix in three arrays:

- **row_ptr** — points to the start of each row in *values*; the number of NNZ elements per row is the difference between consecutive entries;
- **columns** — column indices for each element in *values*;
- **values** — all the NNZ elements of the matrix.

The main challenge is that the compressed row structure makes it hard to balance work evenly among CUDA threads.

### B. Parallelization

A CUDA kernel is a small function launched on the GPU and executed by many threads in parallel. The kernel execution is organized hierarchically into a grid of blocks, where each block contains multiple threads. Each thread and block has a unique ID, which determines the portion of data it processes.

CUDA uses the SIMD (Single Instruction, Multiple Data) execution model, where all threads in a warp (32 threads on NVIDIA A30) execute the same instruction simultaneously using different data. When threads diverge due to conditional branches, execution becomes serialized, which significantly reduces performance.

The goal of CUDA parallelization is to maximize GPU occupancy and workload distribution, while minimizing divergence and synchronization overhead. This becomes challenging in the SpMV problem, especially with CSR format, where rows may have a different number of NNZ elements.

## III. State of the Art

SpMV in CSR format is a core operation widely optimized in HPC and machine learning. On GPUs, vendor libraries like NVIDIA's `cuSPARSE` [1], [2] and AMD's `rocSPARSE`/`hipSPARSE` [2], [3] provide highly tuned CSR SpMV kernels. These libraries are often the default choice for their robustness and performance on regular matrices.

In addition, open-source libraries such as CUSP [4], Ginkgo [5], and Kokkos Kernels [6] offer alternative CSR implementations that focus on improving load balancing and performance on irregular sparsity patterns.

## IV. Methodology and Contributions

This section presents the implementation strategy, the employed algorithms, and the tools used for evaluation.

## A. Implemented Algorithms

Five main implementations are developed and analyzed in this project:

*1) Naive CPU Implementation:* The naive CPU version iterates over each row of the matrix in CSR format and performs a standard dot product with the input vector. It does not take into account memory locality or cache behavior.

*2) Cache-Optimised CPU Implementation:* This version enhances the naive approach by optimizing memory access patterns to better exploit cache locality. Techniques include loop unrolling and minimizing indirect memory accesses.

*3) Naive GPU Implementation:* The CUDA implementation assigns one thread per row of the matrix. While this ensures simplicity and high parallelism, its performance may degrade due to workload imbalance and thread divergence, especially when rows contain a highly variable number of NNZ elements.

However, this kernel can perform relatively well on sparse matrices with very few NNZ elements per row (e.g., 1 or 2). In such cases, each thread performs minimal computation and memory access, which reduces the impact of load imbalance. Moreover, when adjacent threads access nearby elements of the input vector, global memory accesses to `vec[col_idx[j]]` can be partially coalesced, improving memory efficiency despite the lack of explicit warp-level coordination.

**Algorithm 1** Naive GPU SpMV (One Thread per Row)

```
1: procedure SpMV-OneThreadPerRow($row\_ptr$, $col\_idx$, $values$, $x$, $n\_rows$)
2:     parallel for each thread $t$ in $0 \dots n\_rows$
3:         $sum \leftarrow 0$
4:         $start \leftarrow row\_ptr[t]$
5:         $end \leftarrow row\_ptr[t+1]$
6:     for $j \leftarrow start$ to $end - 1$ do
7:             $sum \leftarrow sum + values[j] \cdot x[col\_idx[j]]$
8:     end for
9:         $result[t] \leftarrow sum$
10: end procedure
```

*4) Warp GPU Implementation:* This implementation used the warp scheduling strategy to improve performance. Each row is processed by a single warp, improving memory coalescing and reducing the overhead of thread divergence. However, this approach may lead to underutilization of GPU resources if the number of NNZ elements in a row is less than the warp size. When the block size is a multiple of the warp size, multiple warps are used within a single block and multiple rows are processed in parallel.

**Algorithm 2** GPU SpMV (One Warp per Row)

```
1: procedure SpMV-OneWarpPerRow($row\_ptr$, $col\_idx$, $values$, $x$, $n$)
2:     compute warp_id and lane_id
3:     $sum \leftarrow 0$
4:     for each thread $t$ in warp $w$ do
5:         $j \leftarrow row\_ptr[warp\_id] + lane\_id$
6:         if $j < row\_ptr[warp\_id + 1]$ then
7:             $sum \leftarrow sum + values[j] \cdot x[col\_idx[j]]$
8:         end if
9:     end for
10:    for reduction of $sum$ in warp
11:    if first thread in warp then
12:        store $sum$ in $result[warp\_id]$
13:    end if
14: end procedure
```

*5) Coalesced Binned Warp Implementation:* This kernel aims to improve memory coalescing by assigning each warp to a contiguous group of adjacent matrix rows, called a bin. These bins are precomputed on the CPU by grouping rows until a target number of NNZ elements—typically twice the warp size—is reached. This method works well for matrices where rows have few NNZs, as memory accesses are coalesced and atomic additions rarely contend, leading to efficient parallel execution.

However, if a single row contains many NNZs, it forms a bin by itself, causing workload imbalance across warps. Furthermore, threads must perform a local linear scan to identify which NNZ belongs to which row, reducing warp-level parallelism as some threads wait for others to complete scanning. The use of `atomicAdd` for accumulating results can also lead to contention and serialization when multiple threads update the same output row.

**Algorithm 3** GPU SpMV (Coalesced Binned Warps)

```
1: procedure SpMV-CoalescedBins($row\_ptr$, $col\_idx$, $values$, $x$, $bin\_rows$, $num\_bins$)
2:     compute warp_id and lane_id
3:     $row\_start \leftarrow bin\_rows[warp\_id]$
4:     $row\_end \leftarrow bin\_rows[warp\_id + 1]$
5:     $nnz\_start \leftarrow row\_ptr[row\_start]$
6:     $nnz\_end \leftarrow row\_ptr[row\_end]$
7:     $total\_nnz \leftarrow nnz\_end - nnz\_start$
8:     for $i \leftarrow lane\_id$ to $total\_nnz$ step $WARP\_SIZE$ do
9:         $idx \leftarrow nnz\_start + i$
10:        determine row such that $row\_ptr[row] \leq idx < row\_ptr[row + 1]$
11:        $val \leftarrow values[idx] \cdot x[col\_idx[idx]]$
12:        atomicAdd to $result[row]$ with $val$
13:    end for
14: end procedure
```

*6) Hybrid Short-Long Row GPU Implementation:* This kernel combines two strategies to handle matrices with rows of very different lengths. On the CPU, each row is classified as **short** or **long** based on its number of elements. Two arrays, `short_rows` and `long_rows`, store the indices of these rows.

At runtime, short rows are computed using one thread per row, while long rows use one warp per row with warp-level reduction. This hybrid approach avoids `atomicAdd` and adapts well to any NNZ distribution, combining the benefits of previous kernels.

To guarantee warp-aligned execution for long rows, the number of threads processing short rows is padded to a multiple of 32; the extra threads do no work and serve only to align subsequent warps cleanly.

Memory accesses remain coalesced for both the CSR data and the two new index arrays, ensuring good performance. The main drawback is the extra memory used and the preprocessing needed to build these arrays.

**Algorithm 4** GPU SpMV (Hybrid Short/Long Rows)

```
 1: procedure SpMV-Hybrid($row\_ptr$, $col\_idx$, $values$, $x$, $short\_rows$, $long\_rows$, $num\_short\_rows$, $num\_long\_rows$)
 2:     compute $tid$
 3:     if $tid < num\_short\_rows$ then
 4:         compute SpMV-OneThreadPerRow
 5:     else
 6:         compute $warp\_id$, $lane\_id$
 7:         if $warp\_id < num\_long\_rows$ then
 8:             compute SpMV-OneWarpPerRow
 9:         end if
10:     end if
11: end procedure
```

## B. Performance Evaluation and Tools

For the CPU tests, we used `valgrind` with the `cachegrind` tool to analyze cache usage, and `gettimeofday` to measure execution time. For the GPU kernels, we measured only the kernel execution time using CUDA events. We then used the time to calculate the effective memory bandwidth and the performance in GFLOPs.

## V. System Description and Experimental Set-up

### A. System Description

All tests were performed on a high-performance node equipped with an Intel Xeon Silver 4309Y CPU and an NVIDIA A30 GPU. Software environment details are summarized below:

| **Software** | **Version** |
|---|---|
| GCC (GNU Compiler Collection) | 12.3.0 |
| CUDA Toolkit (nvcc) | 12.3 |
| NVIDIA Driver | 550.144.03 |
| CUDA Version (runtime) | 12.4 |

TABLE I
Software environment versions

The NVIDIA A30 GPU consists of 56 Streaming Multiprocessors (SMs), each containing 64 CUDA cores, resulting in a total of 3584 CUDA cores available for parallel computation.

Threads are organized in groups called warps, each warp containing 32 threads.

While the GPU can manage thousands of threads concurrently by scheduling multiple warps across SMs, the exact number of threads running simultaneously depends on hardware resource availability and occupancy.

| **Model** | **Max Clock** | **Other Specifications** |
|---|---|---|
| Intel Xeon Silver 4309Y | 3.60 GHz | • 32 threads |
| NVIDIA A30 | 1.44 GHz | • Warp size: 32<br>• Max threads/block: 1024<br>• SMs: 56 |

TABLE II
System specifications for CPU and GPU components

### B. Dataset description

For the experimental evaluation, a diverse set of input matrices was carefully selected. Rather than simply increasing matrix sizes, the focus was on choosing matrices with varying characteristics such as sparsity patterns, number of NNZ elements, and structural properties.

This selection strategy aims to highlight the strengths and weaknesses of different SpMV kernels under various conditions. Some matrices are chosen to favor kernels optimized for certain sparsity patterns or memory access behaviors, while others challenge those kernels to better understand their limitations.

| **Dataset** | **Size $n$** | **NNZ** | **NNZ/row (avg,** | **min,** | **max)** |
|---|---:|---:|---:|---:|---:|
| crankseg_1 | 52,804 | 5,333,507 | 101.01 | 1 | 262 |
| ecology1 | 1,000,000 | 2,998,000 | 3.00 | 1 | 3 |
| F1 | 343,791 | 13,590,452 | 39.53 | 1 | 306 |
| inline_1 | 503,712 | 18,660,027 | 37.05 | 1 | 843 |
| mawi_201512012345 | 18,571,154 | 19,020,160 | 1.02 | 0 | 7,397,164 |
| sx-stackoverflow | 2,601,977 | 36,233,450 | 13.93 | 0 | 38,148 |

TABLE III
Statistics of selected square matrices ($n \times n$).

### C. Experimental Set-up

To ensure reliable performance measurements, each test includes a warm-up phase followed by repeated benchmark runs. During the warm-up (2 iterations), the GPU kernel is executed without measuring time. This helps eliminate cold-start effects by initializing memory transfers and preparing GPU caches. We also compute the sum of the output vector in the first warm-up run to verify correctness.

The actual benchmarking phase consists of 10 timed runs using CUDA events to measure kernel execution time. These values are then used to compute average runtime, memory bandwidth, and GFLOPs. .

## VI. Experimental Results

The results mostly meet the expectations based on the characteristics described in Section IV. Since the SpMV problem is mainly memory-bound, we focused more on analyzing memory accesses than on raw GFLOPS.

Measuring bandwidth accurately is difficult due to caching and coalescing effects. To be conservative, we counted every device memory access as a global memory access—even when accesses were to adjacent cells. This means the reported bandwidth is likely overestimated compared to the real value, and in some cases, it approached the peak bandwidth of the A30 GPU.

Looking at all kernels through the bandwidth results:

- **One Thread per Row:** When the average number of NNZ elements per row is low and uniform, all threads have a similar, small workload, which is ideal. However, some matrices like `mawi_201512012345` have mostly low NNZ per row but also a single row with millions of NNZ. This causes one thread to do much more work than others, slowing down the whole process.
- **One Warp per Row:** This kernel works well when the average NNZ per row is larger than the warp size (32). For example, in `crankseg_1`, each thread handles about 3 NNZ, so a warp of 32 threads processes roughly 96 NNZ per row, which fits well. In matrices like `F1` and `inline1`, where NNZ per row varies more, the kernel benefits from frequent block switching. But when the average NNZ is much smaller than the warp size, many threads in a warp do no useful work, causing underutilization.
- **Coalesced Binned Warp:** This kernel generally performs worse than the others because it mixes ideas from CSR and COO formats. Each warp is fully active, but it must search locally to find which row corresponds to each element. It benefits when the average NNZ is low because atomic additions rarely conflict, but atomic operations still slow it down compared to the simpler approach of `OneThreadPerRow`.
- **Hybrid Kernel:** This kernel combines the best parts of `OneThreadPerRow` and `OneWarpPerRow`. It performs well with both low and high NNZ counts and handles variable NNZ distributions effectively. This advantage is especially clear in large matrices like `sx-stackoverflow`.

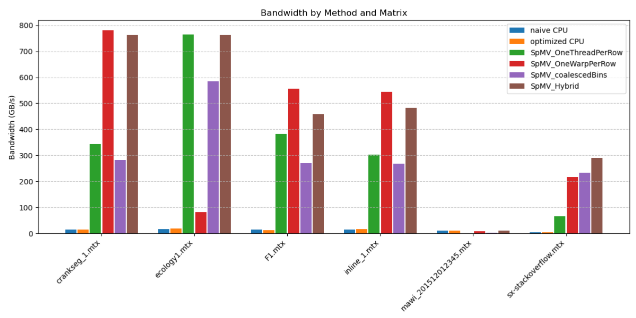

Fig. 1. Measured bandwidth for different kernels across matrices.

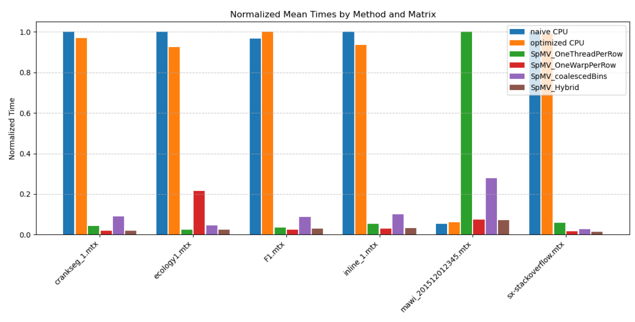

Fig. 2. Normalized execution time of kernels across matrices.

*1) Effect of Block Size on Kernel Performance:* Performance improves notably when increasing the block size from 32 to 64 threads, as more warps per block enhance latency hiding and scheduling.

*a) OneThreadPerRow & OneWarpPerRow:* `OneThreadPerRow` benefits from more than one warp per block, improving scheduling up to 128 threads. `OneWarpPerRow` requires block sizes multiple of 32; it sees major gains from 32 to 64 threads, especially on matrices with many NNZ per row (e.g., `inline1`), but plateaus due to resource limits.

*b) CoalescedBins:* Larger blocks slightly help due to better parallelism across bins, but atomic operations and warp-row grouping limit gains beyond 64 threads.

*c) Hybrid:* Combines the above methods. Gains with block size increase up to 128 threads, but unbalanced workloads reduce further benefit.

*d) Matrix Impact:* `ecology1` (low, uniform NNZ) benefits most in `CoalescedBins` when going from 32 to 64 threads, as increased warps allow more bins to be processed in parallel, improving memory coalescing. Other kernels improve steadily up to 128 threads due to balanced, lightweight workloads. `inline1` (high, variable NNZ) favors warp-level and memory-efficient kernels, with `OneWarpPerRow` showing the largest boost at 64 threads.

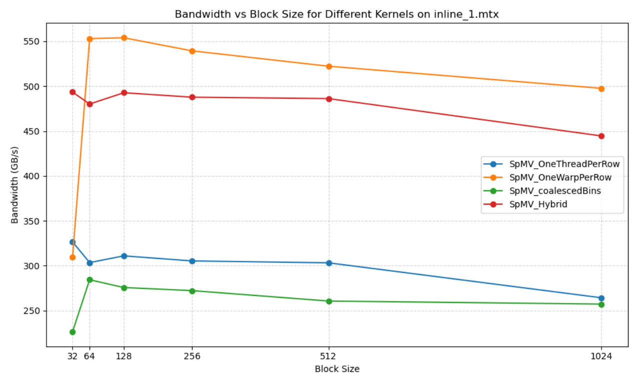

Fig. 3. Bandwidth vs. block size for `inline1` matrix.

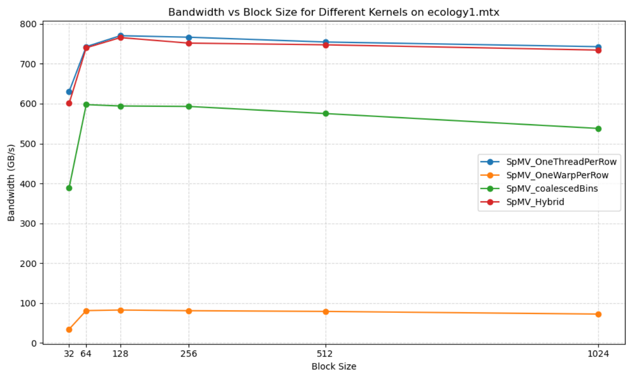

Fig. 4. Bandwidth vs. block size for `ecology1` matrix.

## VII. Conclusions

This work evaluated several GPU SpMV kernels on matrices with different sparsity patterns. Results highlight the importance of choosing the right strategy based on matrix structure and block size.

Performance is heavily influenced by memory access patterns and occupancy. Significant future improvements are expected by leveraging shared memory to stage row data and intermediate results, reducing global memory latency and contention.

Overall, tailoring the kernel to the matrix structure and hardware limits is essential to fully exploit GPU capabilities in sparse computations.

## References

[1] NVIDIA, "cuSPARSE library," https://developer.nvidia.com/cusparse, accessed 2025-05-18.

[2] S.-C. Tsai, X. Liu *et al.*, "Performance analysis of sparse matrix-vector multiplication on modern gpus," *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[3] AMD ROCm Team, "rocSPARSE and hipSPARSE Libraries," https://rocmdocs.amd.com/en/latest/ROC-sparse/index.html, accessed 2025-05-18.

[4] N. Bell and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," NVIDIA, Tech. Rep., 2009. [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/CUSP$_1.1.pdf$

[5] H. Anzt, S. Tomov, and J. Dongarra, "Ginkgo: A high performance sparse linear algebra library," in *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '19)*, 2019.

[6] C. R. Trott *et al.*, "Kokkos kernels: A performance portability library for sparse linear algebra," in *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing*, 2014.