Deliverable report

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

***Abstract*—Sparse Matrix–Vector multiplication (SpMV) is a fundamental linear algebra kernel widely used in scientific computing, graph analytics, and machine learning. Despite its conceptual simplicity, SpMV is challenging to accelerate on GPUs due to the irregular sparsity patterns that lead to poor memory locality and load imbalance.**

In this deliverable, we focus on optimizing SpMV for NVIDIA GPUs using CUDA. Building upon the baseline implementation from Deliverable 1, we introduce shared memory optimizations and advanced CUDA features to improve memory efficiency and reduce latency. We compare our new implementation against the previous GPU kernel and the vendor-optimized cuSPARSE library, reporting execution time, FLOPS, and memory throughput.

I. Introduction

Sparse matrix–vector multiplication (SpMV) is widely used in applications where large datasets contain mostly zero entries. Examples include iterative solvers for scientific simulations, PageRank calculations on web graphs [1], and feature extraction in machine learning.

While GPUs provide massive parallelism that can accelerate such computations, SpMV often remains limited by memory performance rather than computation. The uneven distribution of non-zero elements across matrix rows leads to irregular memory access patterns and workload imbalance among threads. Efficient implementations must therefore carefully manage memory coalescing, warp execution, and shared memory usage to fully exploit GPU capabilities.

In this deliverable, we focus on developing CUDA kernels that improve SpMV performance by using shared memory and other hardware-specific optimizations. We compare these new kernels against the baseline from Deliverable 1 and the vendor-optimized cuSPARSE library, analyzing both performance gains and trade-offs.

II. Problem Statement

SpMV computes $A = B \cdot X$, where B is a sparse matrix and X is a dense vector. In a sparse matrix, most entries are zero and are typically not stored explicitly; only the non-zero elements of B participate in the computation.

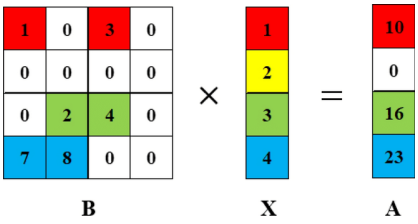

Fig. 1. Sparse Matrix–Vector Multiplication

Conceptually, each element of the result vector is obtained by multiplying the non-zero entries in a row of B with the corresponding elements of X and summing them up.

Efficiently implementing SpMV on GPUs is challenging due to the irregular structure of sparse matrices. The uneven distribution of non-zero elements across rows causes load imbalance and leads to irregular memory accesses when fetching entries of the dense vector X. These non-sequential patterns limit the effectiveness of cache and shared memory, reducing overall computational efficiency.

III. Storage Format: Compressed Sparse Row (CSR)

This project uses the CSR format to store sparse matrices efficiently. Unlike the Coordinate (COO) format, which stores both row and column indices for every non-zero element, CSR avoids redundancy by using a sorted row pointer array to indicate the start of each row in the *values* array. This allows the number of elements in each row to be determined by the difference between consecutive entries in *row_ptr*, reducing memory usage and improving access patterns.

CSR represents a matrix using three arrays:

- **row_ptr**: an array of length $n_{rows}+1$, where $row\_ptr[i]$ points to the index of the first non-zero in row i within the *values* array.
- **columns**: column indices corresponding to each non-zero value.
- **values**: the non-zero elements of the matrix.

CSR is memory-efficient, but the irregular distribution of non-zeros complicates parallelization on GPUs, as highlighted above.

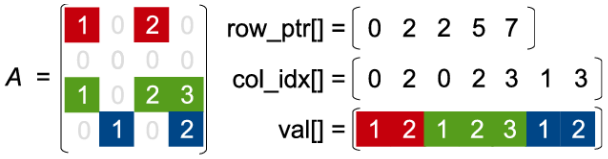

Fig. 2. Example of a sparse matrix and its CSR representation

IV. 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` [2], [3] and AMD's `rocSPARSE`/`hipSPARSE` [3], [4] 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 [5], Ginkgo [6], and Kokkos Kernels [7] offer alternative CSR implementations that focus on improving load balancing and performance on irregular sparsity patterns.

It is important to note that there is no single SpMV kernel that performs optimally in all scenarios. Most kernels are designed to exploit specific sparsity patterns, row distributions, or hardware features. Consequently, the choice of the most efficient kernel depends on the input matrix structure and the underlying GPU architecture. In this report, we compare the kernels developed in Deliverable 1—including their optimizations—with cuSPARSE, to illustrate how relatively simple kernels can be adapted or optimized to achieve better performance on particular inputs.

V. Methodology and Contributions

In Deliverable 2, the four GPU kernels developed in Deliverable 1 have been optimized to further exploit hardware capabilities and improve performance. These kernels are:

1) **OneThreadPerRow**, which assigns one thread per row of the matrix.
2) **OneWarpPerRow**, where a single warp computes the product for each row.
3) **CoalescedBins**, which groups rows into bins processed by warps, enabling more coalesced memory access.
4) **Hybrid**, a multi-class kernel that assigns short, medium, and long rows to threads, warps, and multi-warp blocks, respectively.

A. Common Optimization Strategies

Across all kernels, several common techniques were applied to enhance performance and better exploit GPU hardware:

- **Memory optimizations:** All input arrays are marked with `__restrict__` to prevent pointer aliasing and accessed via `__ldg` when possible. This leverages the read-only cache, reduces global memory latency, and increases memory throughput.
- **Warp-level reductions:** For kernels processing multiple threads per row (medium and long rows), `__shfl_down_sync` is used for intra-warp reductions, eliminating the need for atomic operations and reducing serialization.
- **Loop unrolling:** Inner loops over nnz are unrolled (either manually or via `#pragma unroll`) to increase instruction-level parallelism (ILP), reduce loop overhead, and allow the compiler to schedule instructions more efficiently.
- **Launch Bounds and Occupancy Tuning:** The kernel is annotated with `__launch_bounds__(256, minBlocksPerSM)`, which fixes the maximum number of threads per block to 256. The second parameter (`minBlocksPerSM`) guides the compiler to optimize register allocation and block scheduling to achieve at least this number of active blocks per SM when possible. By maintaining sufficient occupancy, memory latency can be hidden more effectively. In practice, the choice of `minBlocksPerSM` is guided by matrix characteristics (such as average nnz per row), so that the kernel remains efficient for both light and heavy workloads. The final conditions for `minBlocksPerSM` were derived experimentally, by profiling different launch-bound configurations and validating their effect on bandwidth and occupancy (see the comparison here).
- **Grid-stride Loops and Coalesced Memory Access:** While grid-stride loops are often discouraged when they break memory coalescing, in these kernels they are applied carefully: each thread or warp processes elements with a stride equal to the warp size or block size, ensuring that consecutive threads access consecutive memory locations in the CSR arrays. This preserves coalesced access even when a warp is processing multiple rows or when the total number of rows exceeds the number of threads, thus improving load balancing without sacrificing memory throughput.
- **Shared memory usage:** For workloads requiring multi-warp reductions (e.g., long rows in the Hybrid kernel), shared memory is used to store intermediate results. This reduces global memory traffic and improves coalescing.

B. Kernel-Specific Optimizations

1) One Thread per Row Kernel: This simple kernel uses the common optimizations described above, with additional kernel-specific improvements. Each thread is responsible for one or more rows of the matrix, cycling over rows using a grid-stride loop to handle cases where the number of rows exceeds the number of available threads. Within each row, the main loop that sums the nnz is split into two parts: the first employs a manual loop unrolling technique by a factor of four, and the second handles any remaining elements. This approach increases instruction-level parallelism and reduces loop overhead:

Algorithm 1 Pseudocode of OneThreadPerRow kernel

```
for row = threadIdx + blockIdx * blockDim; row ¡ rows; row += blockDim * gridDim do
    start ← row_ptr[row]
    end ← row_ptr[row+1]
    sum ← 0
    for j = start to end − 3 step 4 do
        sum ← sum + values[j] * vec[col_idx[j]]
        sum ← sum + values[j+1] * vec[col_idx[j+1]]
        sum ← sum + values[j+2] * vec[col_idx[j+2]]
        sum ← sum + values[j+3] * vec[col_idx[j+3]]
    end for
    for remaining elements j from index to end-1 do
        sum ← sum + values[j] * vec[col_idx[j]]
    end for
    result[row] ← sum
end for
```

The **launch bounds** annotation is effective for this kernel because each thread processes a row whose length can vary

substantially. Without launch bounds, this irregularity may reduce occupancy on matrices with many short rows, as some warps complete early and leave fewer active threads. By enforcing a minimum number of resident blocks per SM, the compiler limits register usage so that additional blocks can be scheduled, mitigating idle cycles and improving latency hiding.

2) One Warp per Row Kernel: The **One Warp per Row** kernel applies the baseline optimizations described above. Unlike the previous kernel, `launch bounds` are not necessary because each warp is dedicated to a single row, which ensures stable occupancy. Each thread in the warp processes a subset of nnz in the row using a stride equal to the warp size. The partial sums are then combined across the warp using warp-level shuffle operations. The inner loop over nnz is manually unrolled to reduce loop overhead and increase instruction-level parallelism, while lane computations use bitwise operations instead of divisions or modulo for efficiency.

Algorithm 2 Pseudocode of OneWarpPerRow kernel

```
1: tid ← threadIdx + blockIdx * blockDim
2: lane ← threadIdx & (WARP_SIZE - 1)
3: warps_per_grid ← (gridDim * blockDim) / WARP_SIZE
4: for warp = tid / WARP_SIZE; warp ¡ rows; warp += warps_per_grid do
5:     start ← row_ptr[warp]
6:     end ← row_ptr[warp+1]
7:     sum ← 0
8:     for j = start + lane to end-1 step WARP_SIZE do
9:         sum ← sum + values[j] * vec[col_idx[j]]
10:    end for
11:    sum ← sum reduced across warp using shuffle down
12:    if lane == 0 then
13:        result[warp] ← sum
14:    end if
15: end for
```

3) Coalesced Bins Kernel: The **coalesced bins** kernel was restructured compared to its first version. In the old implementation, each thread had to search for the row corresponding to its nonzero element, which introduced irregular control flow and frequent use of `atomicAdd`. The new version follows a design closer to the `one warp per row` kernel: each warp processes entire rows inside its bin, reducing divergence and avoiding atomics by accumulating partial sums within a warp. Although this requires a more complex logic to distribute work among threads, the design improves memory coalescing when loading values and vector elements from global memory, and leverages warp shuffle instructions for efficient intra-warp reduction.

Algorithm 3 Pseudocode of Coalesced Bins kernel

```
1: for bin bw = tid/WARP_SIZE to num_bins step warps_per_grid do
2:     row_start ← bin_rows[bw]
3:     row_end ← bin_rows[bw + 1]
4:     for row = row_start to row_end - 1 do
5:         start ← row_ptr[row]
6:         end ← row_ptr[row + 1]
7:         sum ← 0
8:         for j = start + lane to end - 1 step WARP_SIZE do
9:             sum ← sum + values[j] * vec[col_idx[j]]
10:        end for
11:        for offset = WARP_SIZE / 2 downto 1 step /=2 do
12:            sum ← sum + __shfl_down_sync(sum, offset)
13:        end for
14:        if lane == 0 then
15:            result[row] ← sum
16:        end if
17:    end for
18: end for
```

4) Hybrid Kernel: The **hybrid kernel** was redesigned to split rows into three categories instead of two: short rows are handled by individual threads, medium rows by warps, and long rows by entire blocks. This finer classification improves workload balance and avoids underutilization when rows are too large for a thread but too small for a block. For long rows, the new kernel employs a block-wide reduction scheme using shared memory, allowing multiple warps to collaborate efficiently on the same row. This design increases parallelism within large rows and reduces divergence across the grid.

Algorithm 4 Pseudocode of Hybrid kernel (focus on long rows)

```
1: if block handles short rows then
2:     // Each thread computes one short row (as in OneThreadPerRow)
3: else if block handles medium rows then
4:     // Each warp computes one medium row (as in OneWarpPerRow)
5: else if block handles long rows then
6:     row ← long_rows[block index]
7:     Each thread computes partial sum for its elements
8:     Warp-level reduction using shuffle
9:     Lane 0 writes warp sums to shared memory
10:    Synchronize threads
11:    First warp reduces shared memory sums
12:    Lane 0 writes final result[row]
13: end if
```

C. Shared memory usage

All kernels were tested with versions that exploit shared memory to buffer partial results, but this consistently degraded performance. When rows are assigned entirely to a

single thread or warp (as in the baseline kernels and the early hybrid version from Deliverable 1), the introduction of shared memory only added synchronization and extra memory accesses, resulting in slower execution. In particular, using shared memory for long rows in the first hybrid design (one warp per row) proved counterproductive, since the warp could already complete the reduction internally without additional storage.

Shared memory became beneficial only in the final hybrid kernel, where long rows are split across multiple warps. In this case, a block-wide reduction is required, and shared memory enables efficient coordination among warps, making its overhead justified and improving performance.

D. Performance Evaluation

For all GPU kernels, execution time was measured using CUDA events, capturing only the kernel runtime. The achieved performance in GFLOPs was computed as $2 \cdot nnz/time$, assuming one multiplication and one addition per non-zero element. Memory bandwidth was estimated individually for each kernel, considering the exact arrays read (e.g., CSR arrays, input vector) and the result elements written. This provides a worst-case approximation of the effective bandwidth for each specific kernel implementation.

VI. System Description and Experimental Set-up

A. System Description

All tests were performed on a high-performance node equipped with 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 for parallel computation. Threads are organized in warps, each warp containing 32 threads.

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

Component	Max Clock	Other Specifications
NVIDIA A30	1.44 GHz	• Warp size: 32 • Max threads/block: 1024 • SMs: 56

TABLE II
GPU SPECIFICATIONS

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. .

VII. Experimental Results

The performance evaluation focused on two main comparisons: (i) the improvement of the optimized kernels developed in Deliverable 2 over the kernels from Deliverable 1, and (ii) a baseline comparison against NVIDIA's cuSPARSE library for the same dataset and datatype.

A. Deliverable 1 vs Deliverable 2 Kernels

Figure 4 presents a histogram for each tested matrix, showing the relative performance of each kernel. For each matrix, the columns indicate the percentage improvement or degradation in execution time of the Deliverable 2 kernel compared to its Deliverable 1 counterpart.

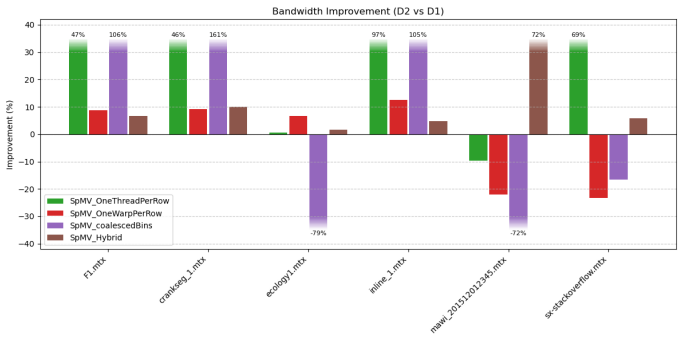

Fig. 3. Bandwidth comparison between kernel deliverables

Overall, the optimized kernels consistently outperform the older versions, with the following trends observed:

- **OneThreadPerRow**: Performance improved for all matrices except `ecology_1`, which has only 3 nnz per row;

threads finish very quickly and almost simultaneously, leaving little opportunity to hide memory latency or exploit ILP. For other matrices, better occupancy and increased instruction-level parallelism yield significant speedup.

- **OneWarpPerRow**: Minor gains for most matrices, but performance degraded on `sx-stackoverflow` (avg $\approx$ 14, max $\approx$ 38k nnz) due to severe warp imbalance: most warps finish quickly on short rows, while a few warps handle extremely long rows, lowering effective occupancy and increasing idle time.
- **CoalescedBins**: Significant improvements for most matrices, thanks to the redesigned strategy that reduces per-thread row searches and implement a one-warp-per-row method. Exceptions are `ecology_1` (-79%), where the previous approach handled very short rows more efficiently despite the per-thread search, and `sx-stackoverflow` (-15%), where extreme row length variability limits the benefits of the new strategy.
- **Hybrid**: Substantial improvements across almost all matrices, with an approximate 10% increase in effective bandwidth on most. The dynamic assignment of short, medium, and long rows, combined with selective use of shared memory for long rows, allowed this kernel to handle extreme cases efficiently, e.g., `mawi_201512012345` (avg $\approx$ 1, max $\approx$ 7M) with a +72% increase in effective bandwidth.

Overall, the comparison between Deliverable 1 and Deliverable 2 kernels shows that targeted optimizations consistently improve performance, although the extent of the gain depends on the sparsity pattern of each matrix.

B. Comparison with cuSPARSE

Here all the custom kernels were compared against the cuSPARSE library. The results show that cuSPARSE generally performs well across all tested matrices, offering consistent performance. However, in specific cases, some of the custom kernels outperform cuSPARSE, as they were tailored to exploit particular structural characteristics of certain matrices.

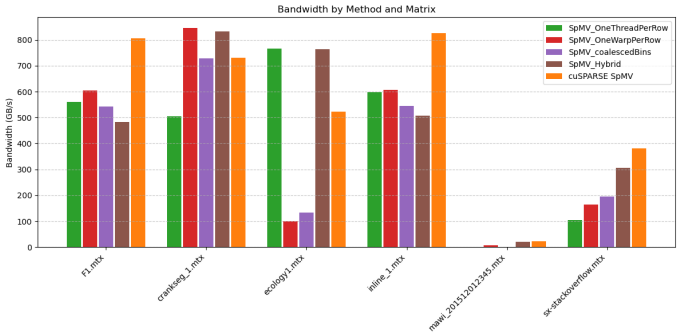

Fig. 4. Bandwidth comparison against cuSparse

For instance, on the *ecology_1* matrix, both *OneThreadPerRow* and the *Hybrid* kernel (which employs a similar logic) achieve comparable performance and surpass cuSPARSE. This is probably because *ecology_1* is a very regular matrix with an average and maximum of 3 non-zeros per row, meaning that each thread performs roughly the same amount of work as the others. This uniformity allows the GPU to exploit more parallelism across threads and better benefit from instruction-level optimizations such as manual loop unrolling.

A different scenario arises for the *crankseg_1* matrix, which has a higher variability in row lengths (average 101, maximum 262). In this case, kernels like *OneWarpPerRow* and *Hybrid* outperform cuSPARSE because their thread-to-row mapping is better aligned with the row structure. By assigning entire rows to warps or blocks and employing intra-warp or block-wide reductions, these kernels balance the workload more effectively, minimizing idle threads and reducing divergence compared to the more generic scheduling in cuSPARSE.

It is also interesting to note that although *OneWarpPerRow* and *CoalescedBin* kernels are based on similar warp-per-row principles, the *CoalescedBin* kernel gains some performance advantage on matrices with very short rows, such as *ecology_1* and *sx-stackoverflow*. By grouping multiple short rows into bins, more threads per warp are kept active and memory accesses become more coalesced, which improves throughput. Nonetheless, this gain comes at the cost of additional preprocessing outside the kernel to build the bin structures.

VIII. Conclusions

This work evaluated several GPU SpMV kernels on matrices with different sparsity patterns. Performance was shown to depend strongly on the structure of the matrix, the efficiency of memory access, and the degree of warp utilization.

The analysis highlighted that no single kernel is universally optimal: while cuSPARSE provides a solid baseline, tailored kernels can surpass it on specific matrices by better balancing the workload or exploiting coalesced memory accesses.

Overall, the results emphasize the importance of matching kernel design to matrix characteristics, demonstrating that a careful alignment between algorithmic strategy and sparsity pattern can yield significant performance gains on modern GPUs.

References

[1] T. Wu, B. Wang, Y. Shan, F. Yan, Y. Wang, and N. Xu, "Efficient pagerank and spmv computation on amd gpus," in *2010 39th International Conference on Parallel Processing*, 2010, pp. 81–89.

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

[3] 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.

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

[5] 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

[6] 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.

[7] 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.