Deliverable report

Antonio Gelain: Mat: 258784,
antonio.gelain@studenti.unitn.it,
antonio.gelain@eagletrt.it,
GitRepo: https://github.com/Tonidotpy/GPU-Computing-2025-258784

***Abstract*—This report focuses on optimizing the Sparse Matrix-Vector multiplication (SpMV) algorithm using the CSR format. Building on the best implementation from the previous deliverable, two new versions were developed: one using shared memory exploiting partial prefix sums, and another using an adaptive approach. Both were benchmarked against the previous implementation and NVIDIA's cuSPARSE library outperforming the first one, having the shared memory implementation being the best, but falling short on the latter one highlighting the performance gap with industrial-grade libraries.**

***Index Terms*—Sparse Matrix, SpMV, CUDA, Parallelization, Storage Format**

I. Introduction

Sparse Matrix-Vector Multiplication (SpMV), which multiplies a sparse matrix by a dense vector, is a core operation in many high-performance computing applications. However, its **low computational intensity** and **irregular memory access** patterns make it memory-bound and challenging to optimize. The irregularity of sparse data structures complicates parallelization, leading to issues like poor cache usage and load imbalance. Efficient SpMV implementation also requires careful use of **hardware-specific** features such as warp execution, memory coalescing, and limited shared memory. Despite its simplicity, SpMV remains a complex and actively researched problem due to these performance challenges.

II. Problem Statement

Sparse Matrix-Vector multiplication is a well-known problem whose purpose is to calculate the product between a sparse matrix, that is, whose values are mostly zeros and therefore do not contribute to the final result, and a dense vector.

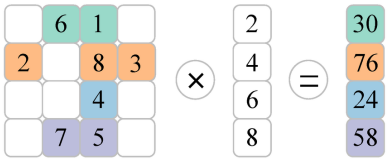


Fig. 1: SpMV multiplication

The product between a matrix A of size $n \cdot m$ and a vector x (which **must be** of size m) gives as a result a vector y of size n as shown in figure 1.

The i-th element of the resulting vector can be calculated given the following formula:

$$y_i = \sum_{j=0}^{m} A_{ij} x_j$$

Where y_i is the i-th element of the resulting vector.

III. Storage format

Storing a large sparse matrix as a dense one is not memory efficient at all since most of the values are zeros and therefore do not contribute to the final result.

To store sparse matrices efficiently many storage formats were, and still are, being developed such as the **Compressed Sparse Row** matrix format, CSR in short.

This format consists of three arrays used to store the rows, columns and non-zero values of the matrix, the main advantage is that the array containing the row indices is constructed by sorting and creating a prefix sum of the original row indices reducing the length of the row array from the number of non-zero to the number of rows plus one.

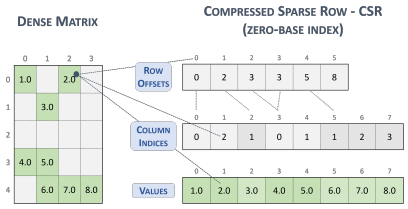


Fig. 2: CSR matrix format

Using the CSR format, access to the i-th row can be obtained just by taking the i-th value of the row array as shown in figure 2.

IV. State of the Art & Related Work

As a starting point, the **Compressed Sparse Row** (CSR) format [1] introduced in 1977 is used to store sparse matrices efficiently in memory.

In the previous deliverable, we observed that simple implementations like **CSR-scalar**, which assigns one thread per row [2], can exhibit highly variable performance depending on the input matrix's structure. In contrast, the baseline used for this work is the **Balanced CSR SpMV** approach [3], which offers improved robustness by being largely insensitive to the matrix's *non-zero- per-row* distribution. This makes it more suitable for applying further optimizations.

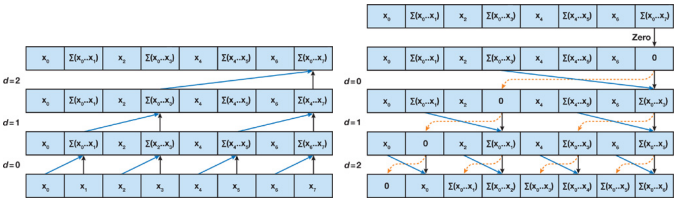
One of the two new implementations builds upon this balanced strategy by **adaptively** selecting among three different methods: CSR-stream, CSR-vector and CSR-vectorL, based on the non-zero density of each row [4], aiming to maximize performance under varying sparsity patterns.

The second implementation is inspired by the CSR5 format [5], which enhances parallelism by reorganizing the CSR data layout and leveraging **partial prefix sums**. In this approach, the matrix-vector product is divided into **fixed-size** blocks, within which partial prefix sums are computed. These are then

efficiently reduced into final row-wise results using **shared memory**. This strategy minimizes redundant computations and significantly improves parallel efficiency, especially for matrices with irregular sparsity patterns.

V. Methodology and Implementation Strategy

For the **shared memory** optimized implementation, two separate CUDA kernels are used: one kernel performs **partial prefix sums** using shared memory, the second computes the final sum for each row, storing the result in the output vector.



(a) Up-Sweep Phase (Reduce) (b) Down-Sweep Phase

Fig. 3: Parallel Prefix Sum

The first kernel is launched with fixed-size blocks of 32 threads. Each thread computes the product between elements of the matrix and the input vector, storing the intermediate results in shared memory.

As illustrated in Figure 3, a **parallel prefix sum** (scan) is performed using the technique described in *GPU Gems 3* [6]. This method consists of two main stages: Up-Sweep (Reduction) and Down-Sweep, achieving faster prefix sum calculation.

Once the prefix sum is computed for each block, the second kernel is used to calculate the final row sums.

Algorithm 1 Total sum

Require: The input matrix A and the output vector y.

procedure SumAndStore(A, y, n, m)
  $y \leftarrow \emptyset$
  $sum \leftarrow 0$
  % Sum all the blocks values of the current row
  for $i \in row\_blocks$ **do**
    $sum = sum + A[i + block\_size - 1]$
  end for
  % Add missing values of the current row
  if $rows[tid+1] \mod block\_size < block\_size$ **then**
    $sum = sum + A[rows[tid+1] - 1]$
  end if
  % Subtract values of the previous row
  if $rows[tid] \mod block\_size > 0$ **then**
    $sum = sum - A[rows[tid] - 1]$
  end if
  $y[tid] = sum$
  return y ▷ the result vector
end procedure

To get the final result, **only the last element** of each row and of each block's prefix sum is copied back to global memory. This is sufficient, since prefix sums require only the values at the end of the current and previous range. Because each block computes its prefix sum independently, the last value in the block is **critical** for correctness.

The second kernel is launched with one thread per row, which combines the prefix sum results to produce the final output vector.

For the adaptive approach, **two auxiliary arrays** are generated to dynamically partition the matrix rows into processing groups. This partitioning is based on the number of non-zero (*nnz*) elements per row, aiming to **balance** the workload and improve efficiency.

Algorithm 2 Adaptive SpMV

Require: Input matrix A, input vector x and the auxiliary arrays $blocks$ and $meta$.

procedure SpMV(A, x, y, n, m, $blocks$, $meta$)
  $y \leftarrow \emptyset$
  $rows \leftarrow blocks[bid+1] - blocks[bid]$ ▷ bid is the block index
  if $rows \geq NUM\_ROWS\_STREAM$ **then**
    Execute CSR-Stream.
  else if $rows = 1$ **then**
    $start \leftarrow blocks[bid]$
    $nnz \leftarrow rows[start+1] - rows[start]$ ▷ $rows$ contains the matrix row indices
    if $nnz > NNZ\_PER\_WG$ **then**
      Execute CSR-VectorL.
    else
      Execute CSR-Vector.
    end if
  end if
  return y ▷ the result vector
end procedure

The number of processing groups (or GPU blocks) is determined during this preprocessing step. Each block uses a fixed number of threads to process its assigned range.

Based on the size and density of each group, a different strategy is applied:

- **CSR-Stream**: Used for batches of multiple rows. It uses **scalar or logarithmic reduction** to process rows efficiently within a warp.
- **CSR-VectorL**: Used for single dense rows, utilizing multiple warps and atomic accumulation operations.
- **CSR-Vector**: Applied to single rows that are not dense enough to justify warp-level parallelism.

This adaptive strategy ensures **high performance** across matrices with varying row densities by applying the most suitable execution model per group.

VI. Experimental Setup & Performance Analysis

A. Dataset description

Eight different matrices were selected as dataset, each with specific characteristics of size and sparsity to evaluate

the performance of the various implementations in different scenarios.

TABLE I: Matrix Details

Name	Rows	Columns	Non-Zeros	Sparsity	Symmetric
1138_bus	1,138	1,138	4,054	99.69%	Yes
bcsstk15	3,948	3,948	117,816	99.24%	Yes
bcsstk31	35,588	35,588	1,181,416	99.91%	Yes
pwtk	217,918	217,918	11,524,431	99.98%	Yes
cage15	5,154,859	5,154,859	99,199,551	99.99%	No
heart1	3,557	3,557	1,385,317	89.05%	No
relat8	345,688	12,347	1,334,038	99.97%	No
connectus	512	394,792	1,127,525	99.44%	No

B. Results & Visualization

Figure 4 clearly shows that the cuSPARSE implementation outperforms all other algorithms in terms of **execution time**, as expected from a *state-of-the-art* library.

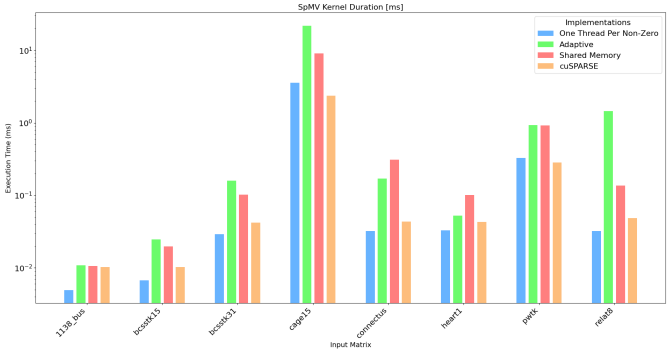


Fig. 4: Kernel Execution Time

Special consideration should be given to the first deliverable's implementation, where part of the computation is performed **outside** the kernel. As a result, the measured kernel execution time does not reflect the total time required for the operation, which is **significantly slower** than the others.

This limitation is also evident in the low compute throughput shown in figure 5, and the high memory throughput in figure 6, as large amounts of data must be transferred back to the host to complete the calculation.

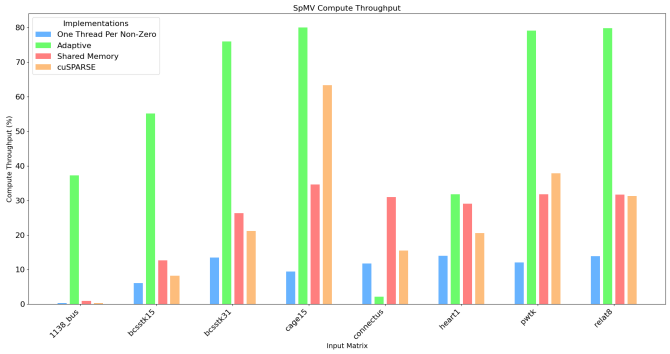


Fig. 5: Kernel Compute Throughput

While the adaptive implementation appears to outperform the shared memory version for matrices with long rows and low sparsity, the shared memory approach is generally **faster** on average. For the largest matrix tested, it offers up to a 60% improvement.

Interestingly, cuSPARSE demonstrates relatively low compute throughput compared to the adaptive method, which achieves up to 80% utilization in most cases.

The compute throughput of the shared memory implementation is comparable to that of cuSPARSE, with a few exceptions as shown in figure 5.

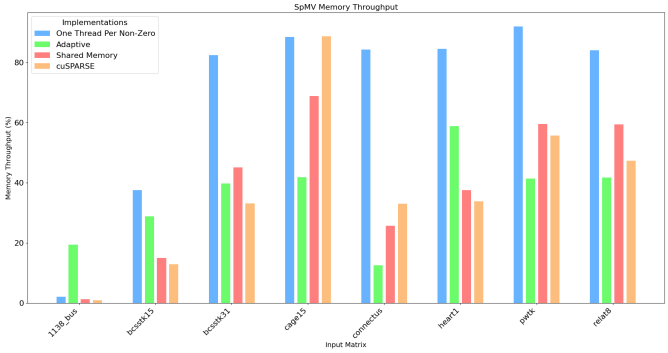


Fig. 6: Kernel Memory Throughput

In terms of memory throughput, the adaptive implementation typically exhibits higher values than the other methods, primarily due to its use of **auxiliary arrays**. However, it tends to perform better on larger matrices.

On the other hand, the shared memory implementation show memory throughput levels roughly similar to cuSPARSE on average.

Lastly, as previously mentioned, the initial deliverable's implementation incurs very high memory throughput due to the significant amount of data transferred back to the host to complete the SpMV operation.

VII. Conclusion

This report explored multiple CUDA-based implementations of the Sparse Matrix-Vector Multiplication (SpMV) algorithm using the Compressed Sparse Row (CSR) format, comparing them to a prior baseline and NVIDIA's cuSPARSE library.

Two new **optimized versions** were introduced: a shared memory approach leveraging partial prefix sums, and an adaptive implementation that selects different strategies based on row density.

Experimental results demonstrate that while cuSPARSE remains the best overall, the custom shared memory implementation **consistently** delivers strong performance, outperforming the previous deliverable and, in some cases, approaching cuSPARSE-level results.

In particular, it excels on large matrices, achieving up to 60% performance improvement over the adaptive method.

The adaptive implementation, though generally slower on average, showed particular advantages in throughput thanks to its **dynamic load balancing** and strategic kernel selection.

In conclusion even if cuSPARSE outpuerformed the other two implementations there is still room for improvement since the adaptive method can be improved by **carefull tuning** and the shared memory approach using prefix sums can also be optimized exploiting different hardware accellerated mechanisms. Further work can also be done trying **advanced matrix formats** like ELL or HYB to push performance even further.

REFERENCES

[1] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, *Yale Sparse Matrix Package. II. The Nonsymmetric Codes*, Aug. 1977. [Online]. Available: https://doi.org/10.21236/ada047725

[2] K. Isupov, V. Knyazkov, I. Babeshko, and A. Krutikov, "Computing the sparse matrix-vector product in high-precision arithmetic for gpu architectures," in *Mathematical Modeling and Supercomputer Technologies*, D. Balandin, K. Barkalov, V. Gergel, and I. Meyerov, Eds. Cham: Springer International Publishing, 2021, pp. 334–345.

[3] G. Flegar and E. S. Quintana-Ortí, "Balanced csr sparse matrix-vector product on graphics processors," in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Cham: Springer International Publishing, 2017, pp. 697–709.

[4] M. Daga and J. L. Greathouse, "Structural agnostic spmv: Adapting csr-adaptive for irregular matrices," in *Proceedings of the 2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, ser. HIPC '15. USA: IEEE Computer Society, 2015, p. 64–74. [Online]. Available: https://doi.org/10.1109/HiPC.2015.55

[5] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," 03 2015.

[6] H. Nguyen, *Gpu gems 3*, 1st ed. Addison-Wesley Professional, 2007, ch. 39.