

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.

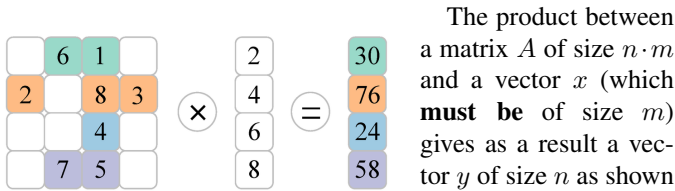


Fig. 1: SpMV multiplication

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.

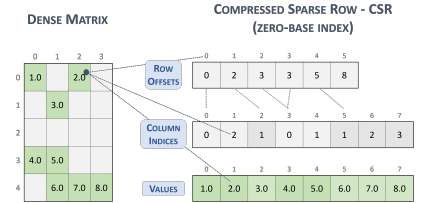


Fig. 2: CSR matrix format

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.

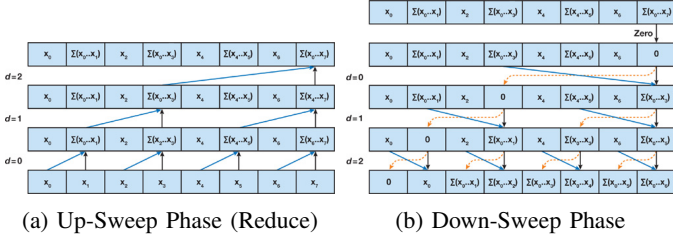


Fig. 3: Parallel Prefix Sum

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 .

```

1: procedure SUMANDSTORE( $A, y, n, m$ )
2:    $y \leftarrow \emptyset$ 
3:    $sum \leftarrow 0$ 
4:   % Sum all the blocks values of the current row
5:   for  $i \in row\_blocks$  do
6:      $sum = sum + A[i + block\_size - 1]$ 
7:   end for
8:   % Add missing values of the current row
9:   if  $rows[tid + 1] \bmod block\_size < block\_size$  then
10:     $sum = sum + A[rows[tid + 1] - 1]$ 
11:  end if
12:  % Subtract values of the previous row
13:  if  $rows[tid] \bmod block\_size > 0$  then
14:     $sum = sum - A[rows[tid] - 1]$ 
15:  end if
16:   $y[tid] = sum$ 
17: return  $y$  ▷ the result vector
18: 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( $A, x, y, n, m, blocks, meta$ )
2:    $y \leftarrow \emptyset$ 
    $rows \leftarrow blocks[bid + 1] - blocks[bid]$  ▷  $bid$  is the block index
4:   if  $rows \geq NUM\_ROWS\_STREAM$  then
   Execute CSR-Stream.
6:   else if  $rows = 1$  then
    $start \leftarrow blocks[bid]$ 
8:    $nnz \leftarrow rows[start + 1] - rows[start]$  ▷  $rows$  contains the matrix row indices
   if  $nnz > NNZ\_PER\_WG$  then
10:    Execute CSR-VectorL.
   else
12:    Execute CSR-Vector.
   end if
14: end if
   return  $y$  ▷ the result vector
16: 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. System Description

For the development of the project the following requirements were used inside the university cluster:

- SLURM 32.11.3
- CUDA 12.1
- GCC 12.3.0
- GNU Make 4.3
- OpenMP 17.0.6
- Eigen 12.3.0 (for the utilities)
- Python 3.9.18 (for the utilities)

The project can be compiled using the provided Makefiles with GCC and CUDA compilers and can be run into the cluster via the SLURM workload manager utilities.

TABLE I: System details

System	Processor	Cores per Socket	RAM	Accelerator
laptop	Intel(R) Core(TM) i5-1035G1 CPU	4 at 3.6 GHz	8 GB	Intel Iris Plus Graphics G1 (Ice Lake)
baldo	Intel(R) Xeon(R) Silver 4214 CPU	12 at 2.2 GHz	405 GB	
edu-short	AMD EPYC 9334 32-Core Processor	32 at 3.9 GHz	810 GB	NVIDIA L40S

B. 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.

TABLE II: 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

C. Experimental Set-up

VII. EXPERIMENTAL RESULTS

VIII. CONCLUSION

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.