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*—Sparse Matrix-Vector multiplication (SpMV) is a fundamental operation in fields such as scientific computing, graph analysis and machine learning, which can often becomes a performance bottleneck due to the large amount of data which needs to be processed. This paper will cover various implementations and optimization strategies used to achieve faster computation leveraging both the Graphics Processing Units (GPUs) and the Compressed Sparse Row (CSR) matrix format.**

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

I. Introduction

Sparse Matrix-Vector Multiplication involves multiplying a sparse matrix in which most elements are zero by a dense vector. Due to its low computational intensity and irregular memory access patterns, SpMV is often memory-bound, making its efficient execution crucial for the overall performance of many high-performance computing applications.

Parallelizing SpMV poses significant challenges, mainly for the irregular structure of sparse matrices, which leads to unpredictable memory access and poor cache utilization. Unlike dense matrix operations, the sparsity pattern is not uniform and can vary widely across applications, making load balancing across threads or processing units non-trivial.

Moreover, modern heterogeneous architectures add further complexity. Efficiently mapping SpMV to such platforms requires consideration of hardware-specific features like warp-based execution, memory coalescing, or limited on-chip memory. As a result, multiple storage formats and algorithmic variants have been developed to adapt SpMV to specific hardware characteristics.

Despite its conceptual simplicity, SpMV remains a challenging and active area of research in various research fields.

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 vector. 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 which can be calculated given the following formula:

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

III. Storage format

Storing every element of large sparse matrices is not memory efficient since most of the values are zeros and therefore do not contribute to the final result.

There exists various storage formats for sparse matrices but the chosen one is the Compressed Sparse Row matrix format, CSR in short. The 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.

IV. Parallelization

Starting from a full CPU implementation as baseline, four different CUDA implementation were developed each one being a variation of the previous ones taking into account a specific problem.

A. One Thread Per Row

For the first CUDA implementation, since each value of the output array can be calculated independently from the others, the naive approach is to create a single thread for each row which multiplies all non-zero values of the rows with the corresponding input vector values and sums them together sequentially.

B. Additional Sorting Kernel

Since the CSR format requires sorting, for large matrices a lot of time is is spent on constructing the row array of the matrix so as a solution a Bitonic Parallel Sort algorithm was implemented to achieve better total execution time.

C. One Thread Per Non-Zero

One of the main problems of the naive implementation is the fact that if the matrix consists of a small number of rows and a large number of non-zeros, a low number of thread will be dispatched which has to do most of the calculations and this will result in a lower number of floating point operations per second.

A possible approach to mitigate this prolem is to use a single thread for each non-zero element of the matrix with the additional challenge of subdividing the workload between different groups and accessing the correct values.

Another problem is concurrency caused by the access to the same memory location from different threads, mainly during the sum operation which is executed sequentially by the CPU.

D. Warp Reduction

To further reduce the time taken by the sequential step of the previous implementation a technique known as warp reduction is used for each row of the array to parallelize the sum hence increasing the performance.

V. State of the Art

From the original Compressed Sparse Row (CSR) [1] implementation many more research has been conducted to improve its use in algorithms such as Sparse Matrix-Vector Multiplication.

One of the main problems of the format was its performance which is highly dependent on the used hardware and the actual structure of the matrix. To achieve this, several optimized variants of CSR have been developed such as CSR5 and its extension CSR5BC which improve parallelism and memory access on modern GPUs using segmented sum techniques and index compression [2].

The CSR & RV format reduces memory usage by storing redundant values only once, significantly lowering storage costs and boosting throughput [3].

On heterogeneous systems, formats like CSR-k restructure data hierarchically for better adaptability across CPUs and GPUs [4]. Auto-tuning frameworks such as AlphaSparse generate customized kernels tailored to both sparsity patterns and hardware, delivering further gains [5]. Additionally, emerging architectures like Processing-In-Memory (PIM) are prompting new SpMV implementations, such as SparseP, that harness in-memory computing advantages [6]. Together, these innovations demonstrate that while CSR remains foundational, ongoing adaptations are essential for maximizing SpMV efficiency on evolving platforms.

VI. Methodology and Contributions

VII. System Description and Experimental Set-up

A. System Description

System	Processor	Cores per Socket	RAM	Accelerator
edu-short	AMD EPYC 9334 32-Core Processor	32 at 3.9 GHz	810 GB	NVIDIA L40S

TABLE I
System details

B. Dataset description

C. Experimental Set-up

VIII. Experimental Results

IX. 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] W. Zhang, Y. Chen, B. Zhang, X. Shen, J. Xue, and C. J. Xue, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Springer, 2016, pp. 77–93.

[3] Z. Tang, X. Zhu, L. Wang, W. Song, D. Cheng, and Q. Wang, "Csr&rv: A compressed sparse row format with reduced value redundancy," in *Proceedings of the 19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*. Springer, 2023.

[4] W. Liu, J. Chen, G. Lin, X. Wang, and M. Liu, "Csr-k: A fast and portable spmv kernel with hierarchical row blocking," *arXiv preprint arXiv:2203.05096*, 2022. [Online]. Available: https://arxiv.org/abs/2203.05096

[5] J. Yin, X. Li, Y. Li, X. Wang, and J. Chen, "Alphasparse: Auto-tuning and accelerating spmv with machine-designed formats," *arXiv preprint arXiv:2212.10432*, 2022. [Online]. Available: https://arxiv.org/abs/2212.10432

[6] S. Lee, M. Kim, M. Joung, J. Park, and J. W. Lee, "Sparsep: Toward realistic performance modeling of sparse matrix-vector multiplication on processing-in-memory systems," *Proceedings of the ACM on Measurement and Analysis of Computing Systems (POMACS)*, vol. 6, no. 1, pp. 1–35, 2022.