Optimization for SpMV

Nicolo' Marconi: Mat: 258859,
`nicolo.marconi@studenti.unitn.it`,
GitRepo: `https://github.com/nicolomarconi02/GPU-Computing-2025-258859`

***Abstract*—This project examines the performance of different sparse matrix-vector multiplication (SpMV) algorithms on CPU and GPU architectures. Starting with a basic CPU implementation in CSR format, progressively optimized versions were developed, including parallel CPU and multiple GPU-based strategies, such as thread-per-row, element-wise, and warp-level approaches. These implementations were tested on a diverse set of matrices from the SuiteSparse Matrix Collection that varied in size, sparsity, and structural properties. Execution time, throughput, and memory bandwidth were collected and analyzed as performance metrics. The results demonstrate that, although GPU implementations generally outperform CPU ones, their efficiency depends heavily on the matrix's structure.**

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

I. Introduction

Sparse matrix–dense vector multiplication (SpMV) is a fundamental computational process in many scientific and engineering applications, including machine learning, linear algebra, and computational physics. It involves multiplying a large, sparse matrix—composed of a high number of zero elements and a low number of non-zero elements—by a dense vector.

Due to the sparsity pattern, only the non-zero entries are stored and processed, reducing memory usage and computational resources compared to dense matrix operations. However, the sparsity of the elements leads to highly non-contiguous and irregular memory access patterns, resulting in low arithmetic intensity and poor cache utilization, especially in large-scale problems.

Also, parallelizing this operation requires considering data locality, thread synchronization, and memory bandwidth constraints. In fact, load imbalance can arise when different threads process rows with varying numbers of non-zero elements.

Efficient parallel execution can be achieved by balancing the workload distribution, minimizing synchronization overhead caused by accessing the same resources, and maximizing memory throughput.

II. Problem Statement

Sparse matrix-dense vector multiplication is essentially a multiplication task between a matrix and a vector. The worst case scenario is a dense matrix-dense vector multiplication which has a complexity of $O(N^3)$. This operation can be calculated using the following formula:

$$y_i = \sum_{j=0}^{M} A_{i,j} \cdot x_j$$

where A is a sparse matrix of N rows and M columns, $A_{i,j}$ is the element of the matrix in the i^{th} row and j^{th} column, x is the dense vector of size M and y is the result dense vector of size N.

A. Storage Format

The **key feature** of SpMV is that, since the matrix is sparse, most of its elements are zeros that can be **ignored** during storage and computation. This leads to the selection of a storage format that stores all the matrix's essential information.

In this case, the **Compressed Sparse Row (CSR)** format has been chosen. It consists of three arrays that store the matrix's rows, columns, and values. The columns and values arrays contain information only about the non-zero elements and are the same size as the number of non-zero elements in the matrix. The columns array stores the indexes of the cells that contain the non-zero elements, and the values array contains the values of the non-zero elements.

The rows array, on the other hand, is constructed by **sorting** and creating a **prefix sum** of the row indexes. This results in an array whose length is equal to the number of rows N of the matrix plus one. Each element in the modified rows array represents the number of non-zero elements in that row. In this format, the first element of the row array is always zero, and the last element is the number of non-zero elements in the matrix.

B. Parallelization

Two CPU-optimized implementations and three GPU implementations were developed based on a CPU implementation used as a baseline for the next optimizations. All were written in C++ and CUDA.

1) CPU: The main CPU problem is the time spent parsing the matrix market file and sorting the matrix to compute the CSR array. Due to this limitation, which leads to a long execution time, many sorting strategies have been applied. One approach was a *QuickSort* with an $O(N \cdot log(N))$ complexity. However, this approach did not perform well when the size of the matrix increased due to the recursion.

The second approach involved sorting while parsing each element. This approach used *binary search* to find the correct position of the new element, then swapped the elements until the vector was sorted. This approach drastically reduced the execution time compared to *QuickSort* but had limitations with matrices containing millions of non-zero elements.

The third and final approach is to implement a parallel algorithm, such as the **Parallel Bitonic Merge Sort**, which has an $O(log^2(N))$ complexity and uses $\mathrm{std::jthread}$.

One implementation of SpMV multiplication on the CPU uses OpenMP for parallelization.

2) GPU: On the GPU, always by using the *Parallel Bitonic Merge Sort*, which this time is a kernel function executed on the **device**, the first part of the problem has been resolved, achieving very good performance with low execution time for the parsing and sorting of the matrix. The main problem on the GPU is finding the right **memory access pattern**, in order to speed up the process and keep an high throughput.

III. State of the Art

Sparse matrix–vector multiplication (SpMV) is a critical kernel in scientific computing. It is characterized by low arithmetic intensity and irregular memory access patterns. Its performance is often limited not by computation, but by **memory bandwidth** and **latency**. Consequently, research in this area has explored various data formats, access patterns, and parallelization strategies to improve performance on different hardware platforms.

Some researchers have focused on matrix formats such as the *ELL* (ELLPACK) format [1], which performs well when the maximum number of non-zero elements per row differs little from the average. This format is well-suited for vectors and SIMD architectures. Other researchers have focused on formats such as the *HYB* (Hybrid) format [2], which combines *ELL* and *COO* to balance performance for matrices with non-uniform sparsity. Another format is the *CSR5* [3], an extension of CSR that splits the matrix into tiles and adds auxiliary data used by the GPU to improve performance.

Beyond matrix storage formats, some methodologies aim to increase SpMV performance. One example is row reordering with the *Reverse Cuthill-McKee* (RCM) algorithm, a graph-based method for reordering sparse matrices. It is effective in minimizing matrix bandwidth and enhancing the efficiency of matrix operations. [4]

IV. Methodology and Contributions

The project began with the implementation of a basic version capable of handling **Matrix Market files** with a limited number of non-zero elements. Once the baseline version was operational, the program was tested with progressively larger matrices. Each time an issue was encountered, a specific optimization was developed to address it, which led, step by step, to the final implementation.

A custom class named *Profiler* was used to profile and measure execution time using $\mathrm{std::chrono}$ for both the CPU and GPU implementations.

1) Naive CPU implementation: First, a sequential naive CPU implementation for the SpMV multiplication was developed in C++ (Algorithm 1).

2) CPU parallelized implementation: This version extends the sequential implementation by parallelizing the outer loop using OpenMP directives. The core algorithm remains the same as in the sequential implementation, with the addition of #pragma omp parallel for schedule(dynamic) before the main loop. This parallelization improves computational performance by reducing the time required to compute the SpMV multiplication, especially for large matrices. The schedlue(dynamic) was chosen because it handles better the load balancing due to the variable row sparsity of the matrix.

3) Naive GPU thread per row implementation: This version (Algorithm 2) is the direct evolution of the parallelized CPU implementation on the GPU by using CUDA threads and blocks. This approach is faster on the GPU respect to the CPU but it has limitations with large and unbalanced non-zero elements per row, because the load for each thread could be different.

4) GPU element wise implementation: Starting from the thread per row implementation, a completely different approach is to have CUDA threads working at the level of non-zero elements rather than rows (Algorithm 3). This leads to a more even distribution of work between threads and better performances, especially for irregular matrices, but it has concurrency limitations due to the atomic operation needed, which could become a bottleneck.

5) GPU warp based implementation: The third implementation (Algorithm 4) is a warp-based approach, in which each warp of size 32 is assigned to a row of a matrix. The thread within the warp cooperate with each other to compute the dot product of that row with the input vector. By using a warp-wide reduction, the partial sum is computed. This approach has a good load balancing across rows, uses an efficient memory access pattern and it does not need any atomicAdd() to work, which means better performance when using large matrices.

Algorithm 1 CPU: sequential multiplication

```
1: procedure THREADPERROW($rowsMat$, $colsMat$, $valuesMat$, $vec$, $N\_ROWS$)
2:     $res \leftarrow \emptyset$
3:     for $i$ in $\{1 \dots N\_ROWS\}$ do                ▷ with $i = i + 1$
4:         $rowStart = rowsMat[i-1]$
5:         $rowEnd = rowsMat[i]$
6:         $sum = 0$
7:         for $j$ in $\{rowStart \dots rowEnd\}$ do
8:             $sum = sum + valuesMat[j] * vec[colsMat[j]]$
9:         end for
10:        $res[i-1] = sum$
11:    end for
12:    return $res$                                  ▷ the result vector
13: end procedure
```

V. System Description and Experimental Set-up

A. System Description

For the development of the project, the University cluster has been used with these specifications:

Algorithm 2 GPU: thread per row multiplication

```
1:  procedure THREADPERROW($rowsMat$, $colsMat$, $valuesMat$, $vec$, $N\_ROWS$)
2:      $startIndex = threadIdx + (blockDim * blockIdx) + 1$
3:      $stride = blockDim * gridDim$
4:      $res \leftarrow \emptyset$
5:      for $i$ in $\{startIndex \dots N\_ROWS\}$ do        ▷ with $i = i + stride$
6:          $rowStart = rowsMat[i-1]$
7:          $rowEnd = rowsMat[i]$
8:          $sum = 0$
9:          for $j$ in $\{rowStart \dots rowEnd\}$ do
10:             $sum = sum + valuesMat[j] * vec[colsMat[j]]$
11:         end for
12:         $res[i-1] = sum$
13:     end for
14:     return $res$        ▷ the result vector
15: end procedure
```

Algorithm 3 GPU: element wise approach

```
1:  procedure ELEMENTWISE($rowsMat$, $colsMat$, $valuesMat$, $vec$, $N\_ROWS$)
2:      $startIndex = threadIdx + (blockDim * blockIdx)$
3:      $stride = blockDim * gridDim$
4:      $NNZ = rowsMat[N\_ROWS]$
5:      $res \leftarrow \emptyset$
6:      for $i$ in $\{startIndex \dots NNZ\}$ do        ▷ with $i = i + stride$
7:          $lhs = 0$
8:          $rhs = N\_ROWS$
9:          while $lhs < rhs$ do
10:             $mid = (lhs + rhs)/2$
11:             if $rowsMat[mid + 1] <= i$ then
12:                 $lhs = mid + 1$
13:             else
14:                 $rhs = mid$
15:             end if
16:         end while
17:         $product = valuesMat[i] * vec[colsMat[i]]$
18:         $atomicAdd(res[rhs], product)$
19:     end for
20:     return $res$        ▷ the result vector
21: end procedure
```

- GCC 11.5.0
- CUDA 12.1.1
- MAKE 4.3
- CMake 3.26.5
- SLURM 23.11.3

The CMake files automatically check if CUDA is installed on the system and compiles the CUDA files if it is found or not. In table I there is a detailed description of the system specifications, obtained with *lscpu* and *free* commands.

Algorithm 4 GPU: warp based approach

```
1:  procedure WARP($rowsMat$, $colsMat$, $valuesMat$, $vec$, $N\_ROWS$)
2:      $warpSize = 32$
3:      $threadId = threadIdx + (blockDim * blockIdx)$
4:      $warpId = threadId/warpSize$
5:      $lane = threadIdx\%warpSize$
6:      if $warpId >= N\_ROWS$ then
7:          $return$
8:      end if
9:      $row = warpId$
10:     $rowStart = rowsMat[row]$
11:     $rowEnd = rowsMat[row + 1]$
12:     $sum = 0$
13:     $res \leftarrow \emptyset$
14:     for $j$ in $\{rowStart + lane \dots rowEnd\}$ do        ▷ with $j = j + warpSize$
15:         $sum = sum + valuesMat[j] * vec[colsMat[j]]$
16:     end for
17:     for $offset$ in $\{warpSize/2 \dots 0\}$ do        ▷ with $offset = offset/2$
18:         $sum = sum + shfl\_down(sum, offset)$
19:     end for
20:     if $lane == 0$ then
21:         $res[row] = sum$
22:     end if
23:     return $res$        ▷ the result vector
24: end procedure
```

System	Processor	Cores per Socket	RAM	Accelerator
Baldo	48× Intel Xeon Silver 4214 CPU	12 at 2.2 GHz	376 GB	
edu-short	32× Intel Xeon Silver 4309Y CPU	8 at 2.80 GHz	251 GB	NVIDIA L40S

TABLE I
SYSTEM DETAILS

B. Dataset description

All the matrices were downloaded from the *sparse.tamu.edu dataset*. As shown in Table II, all the matrices in the dataset are highly sparse, except for the *air02*, which is denser respect to the other (18% of the elements of the matrix are non-zero). The first three matrices are symmetric of different type. In particular, *mawi* is the matrix with the largest number of rows available on the sparse.tamu.edu dataset.

The last two matrices were chosen because they have a rectangular shape: *spal_004* with more columns respect to the rows, and *sls* with more rows respect to the columns.

C. Experimental Set-up

For the experimental setup, the **execution time**, **throughput**, and **bandwidth** of each SpMV implementation were measured. After parsing and sorting each matrix on the GPU, two initial **warmup cycles** were applied, followed by ten **measured cycles**. However, the *mawi* matrix was excluded due to the long execution time of the algorithm caused by its

TABLE II
DATASET

Name	Rows	Columns	Non-Zeros	Symmetric	Type
dielFilterV3real	1102824	1102824	89306020	Yes	real
mycielskian	98303	98303	100245742	Yes	binary
mawi_201512020330	226196185	226196185	480047894	Yes	integer
air02	50	6774	61555	No	binary
spal_004	10203	321696	46168124	No	real
sls	1748122	62729	6804304	No	real

large size. For this matrix, only five cycles were measured to stay within the five-minute **time limit** on the cluster.

Also for the same reason, every CPU sequential and parallel implementation with all the matrices has been executed on Baldo and not on the cluster.

VI. Experimental Results

All plots have a logarithmic Y-axis scale to accurately represent GPU data alongside CPU data.

A. Execution time

Figure 1 shows the raw execution time of each algorithm. As shown, the GPU-based methods consistently outperform the CPU implementations in terms of speed.

The *GPU warp* generally achieved the lowest execution time for all matrices thanks to the optimized access pattern. It is clear from this chart that the lack of parallelization significantly increases execution time.

It is important to note that the *GPU thread-per-row* algorithm has a higher execution time than both CPU implementations for the *mawi* matrix. This is due to the pattern of the non-zero elements in the matrix, which alternates between very sparse and highly dense rows.

B. Throughput

Figure 2 shows the advantage of GPU computation for large matrices. In fact, the highest throughput was achieved by the *GPU warp* implementation. The CPU sequential algorithm has an almost constant throughput for each matrix. It is also evident here the difficulty of the GPU thread per row with the mawi matrix.

C. Bandwidth

Figure 3 shows all the previous observations made for throughput and execution time. Focusing on the *mawi* matrix, we can see that all GPU implementations are similar to or worse than CPU ones. This could be due to the matrix's high sparsity, which causes an uncoalesced memory access pattern and worsens performance.

VII. Conclusions

Evaluating the performance of various sparse matrix-vector multiplication (SpMV) algorithms across multiple matrices provides key insights into how matrix characteristics affect computational efficiency. In general, GPU-based implementations outperform CPU-based ones in terms of throughput and bandwidth, especially for structured or denser matrices.

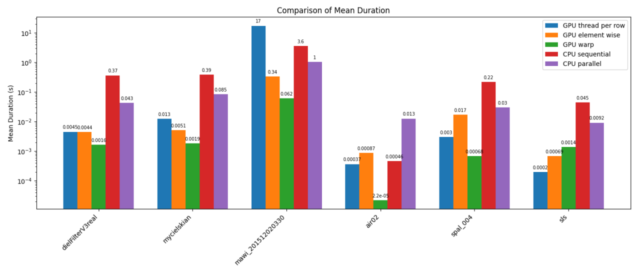

Fig. 1. Execution time SpMV

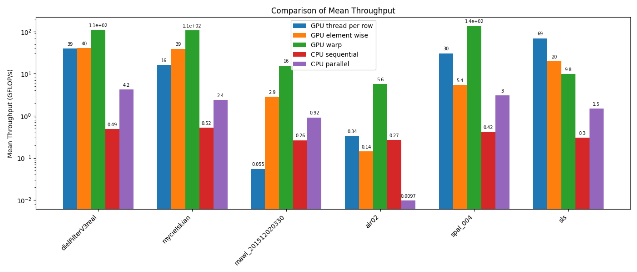

Fig. 2. Estimated throughput SpMV

Among GPU strategies, *warp-based* and *element-wise* methods are the most efficient when the workload is **balanced** and memory accesses are **coalesced**. However, the results for the *mawi* matrix reveal a significant limitation: despite the GPU's computational potential, memory bandwidth remains underutilized due to the matrix's extreme sparsity and irregular structure, leading to inefficient memory access and high thread divergence. These observations suggest that **no single implementation is optimal** for all matrix types.

Future work will focus on designing adaptive SpMV frameworks that can dynamically select the most suitable algorithm based on matrix features, such as density, row distribution, and symmetry.

References

[1] D. R. Kincaid and D. M. Young, "A brief review of the itpack project," *Journal of Computational and Applied Mathematics*, vol. 24, no. 1, pp. 121–127, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0377042788903470

[2] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking,*

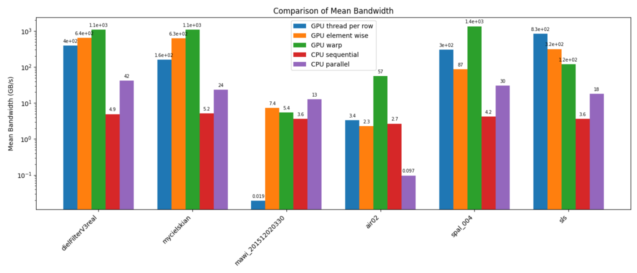

Fig. 3. Estimated bandwidth SpMV

Storage and Analysis, ser. SC '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1654059.1654078

[3] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," 2015. [Online]. Available: https://arxiv.org/abs/1503.05032

[4] J. Hou, H. Liu, and S. Zhu, "Rcm++:reverse cuthill-mckee ordering with bi-criteria node finder," 2024. [Online]. Available: https://arxiv.org/abs/2409.04171