Optimization for SpMV

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

***Abstract*—This project investigates optimizing Sparse Matrix-Vector (SpMV) multiplication on GPU architectures, building upon initial CPU and basic GPU implementations. The study addresses the challenges of irregular memory access patterns and load imbalance inherent in sparse computations by exploring advanced CUDA-level optimizations, including strategic use of shared memory and fine-tuned kernel designs. The refined approach aims to significantly improve execution time, memory throughput, and overall computational efficiency. This report details these advanced methodologies and quantifies the performance gains achieved across diverse sparse matrix structures.**

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

I. Introduction

Sparse Matrix-Vector (SpMV) multiplication is an operation used in a vast range of scientific and engineering fields, like machine learning algorithms and computational physics simulations. At its core, SpMV involves multiplying a sparse matrix, in which the vast majority of the elements are zero, by a dense vector. The defining characteristic of sparse matrices is their efficiency potential: in fact, only the non-zero elements need to be stored and processed, which can drastically reduce memory footprint and computational requirements compared to the dense matrix multiplication.

However, this inherent sparsity introduces significant challenges: the non-zero elements are often scattered irregularly throughout the matrix, leading to highly non-contiguous and irregular memory access patterns. This can result in poor cache utilization and low arithmetic intensity, particularly in large-scale problems. Furthermore, effectively parallelizing SpMV on modern many-core architectures like GPUs demands careful consideration of data locality, thread synchronization, and memory bandwidth constraints. Issues like load imbalance, where different threads are assigned rows with varying numbers of non-zero elements, can severely impact performance. Achieving efficient parallel execution, therefore, hinges on balancing workload distribution, minimizing synchronization overhead, and maximizing memory throughput.

Building on initial explorations of SpMV on both CPU and GPU, this second phase of the project dives deeper into advanced GPU optimization techniques. The objective is to overcome the performance bottlenecks observed in simpler implementations by focusing on strategies that leverage GPU-specific features, such as shared memory and more sophisticated thread organization, to improve memory access patterns and computational efficiency.

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.

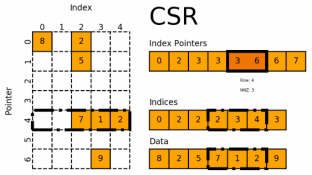

Fig. 1: CSR format

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.

III. State of the Art

Sparse matrix–vector multiplication (SpMV) 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, focusing on four different key areas: hybrid data structures and advanced algorithm,

parameters auto-tuning, predictive optimization using machine learning and specific hardware features.

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.

In a modern SpMV implementation, considering the high number of tunable parameters, auto-tuning frameworks can optimize the task, by automating the process of finding the best configuration for a given input matrix on a specific GPU, by tuning parameters as the storage format, the thread and grid size and the number of thread per row.

The auto-tuning could lead to the need of reducing the overhead due to the runtime benchmarking. This could be replaced by a predictive model using machine learning [2], that, given the structural features of an input matrix, instantly predict the best performing kernel and storage format.

For what reguards the hardware features, for example, the NVIDIA Hopper Architecture [3] supports bidirectional asynchronous memory transfer between shared and global memory, enabling efficient techniques for reducing the memory latency.

IV. Methodology and Contributions

The project is based on the version implemented in the *Deliverable-1*. By starting from the **warp** based SpMV, used as the baseline, the goal was trying to improve the performance obtained by focusing on using shared memory, read-only cache, loop unrolling and work balancing. With this goal in mind, other four kernel have been written.

1) GPU warp based (Deliverable-1): The baseline implementation 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 is very fast but it could be affected by work imbalance in case of rows with very different number of non-zero elements.

2) GPU warp loop: This implementation (Algorithm 1) has the same concept of the baseline algorithm, it works with a warp-based approach, but has a dynamic scheduling that allows to distribute work in a better way respect to the baseline. It is composed of a while loop that distributes the work by assigning a row to a *leader* thread (with laneId = 0), then the row is broadcasted to the other 31 thread of the warp and it is computed the dot product and the partial sum for all the non-zero elements in the row in a for loop. At the end of the while loop, after a parallel reduction, only the leader thread writes the sum in the result vector. In order to improve also the performance of the algorithm, it has been used also the loop unrolling and the read-only cache access.

3) GPU warp tiled: The GPU warp tiled algorithm has the goal of reducing the high latency memory accesses when reading from the input vector. The main idea is to process the matrix in tiles, which are precomputed on the CPU, that allows to load into the GPU's shared memory the required parts of the input vector, in order to reach an high data reuse and increase the performance. These tiles are computed based on a maximum width of $BLOCK\_COL\_CHUNK = 2024$, used also to assign the static shared memory size. To each tile is assigned a CUDA block, and each thread within the block work together to load the respective tile of the input vector into the shared memory. As in the other implementation, after computing a dot multiplication and partial sum, there is a parallel reduction and only the thread in the $lane = 0$ assign the resulting sum in the output vector.

4) GPU merge based: This implementation has the goal of achieving the perfect load balancing by parallelizing over the non-zero elements of the matrix rather than the rows or tiles. This method divides the work in equally sized contiguous block of non-zero elements, uses binary search over the *CSR* offset array in order efficiently find the row which belongs to the non-zero element, and then iterates through its assigned chunk of elements and performs the multiplication and partial sum, handling also row boundaries. The result must be saved in the output vector by using an atomic addition because different threads will be processing elements belonging to the same row at the same time. The main issue with this access pattern choice are the loss of the coalesced memory access, the reliance on the *atomicAdd* function that could become a bottleneck and the computational overhead due to the binary search and the inner while loop that handles the cross row boundaries.

5) GPU merge path: After other two optimization of the GPU merge based algorithm, the final version of this concept has been implemented into three different kernel, creating the **merge path algorithm** (later referred as *merge_v4*). The first step is the *planning phase*, in which the goal is to divide the work perfectly among all threads. This step could be visualized as a 2D grid as in the figure 2, in which is represented the CSR offset on the x-axis, meanwhile on the y-axis it is represented an index array corresponding to the number of non-zero element of the matrix. This phase defines a path that starts from the top-left of the grid, and moves down for each non-zero and right at the start of each new row, then the total path length is divided into perfectly equal segments, one for each thread in the grid, and as an output there are two arrays that stores the information about the coordinates where each thread's work begins by performing a binary search.

The second step does the actual multiplication by using the two arrays computed in the previous step: each thread can store the partial result in a temporary buffer if it calculates the sum for the ending part of its starting row, write in the output vector atomically when the thread computes the sum for the beginning part of its ending row, and non-atomically when the thread performs a sum for a full row.

The third step is called *cleanup phase* and take care of atomically adding into the output vector the partial sum saved in the temporary buffer of the previous step.

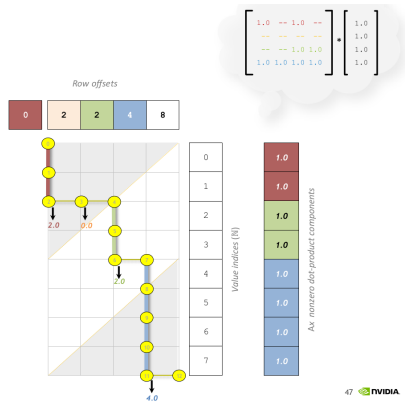

Fig. 2: 2D grid visualization

Algorithm 1 GPU: Parallel SpMV (Warp-centric, Loop-based)

procedure PARALLELMULTIPLICATIONWARPLOOP($csr, cols, vals, vec, res, globalCounter, N\_ROWS$)
  $laneId \leftarrow$ threadId in warp
  while true **do**
    if $laneId = 0$ **then**
      $row \leftarrow$ AtomicallyIncrement($globalCounter$)
    end if
    $row \leftarrow$ BroadcastFromLane0(row)
    if $row \geq N\_ROWS$ **then**
      break ▷ All rows have been processed
    end if
    $rowStart \leftarrow$ LoadReadOnly($csr[row]$)
    $rowEnd \leftarrow$ LoadReadOnly($csr[row+1]$)
    $sum \leftarrow 0$
    for all j from $rowStart + laneId$ to $rowEnd$ step $warpSize$ **do**
      $col \leftarrow$ LoadReadOnly($cols[j]$)
      $val \leftarrow$ LoadReadOnly($vals[j]$)
      $x \leftarrow$ LoadReadOnly($vec[col]$)
      $sum \leftarrow sum + val \cdot x$
    end for
    for $offset \in \{16, 8, 4, 2, 1\}$ **do**
      $sum \leftarrow sum +$ GetValueFromLane($laneId + offset$)
    end for
    if $laneId = 0$ **then**
      $res[row] \leftarrow sum$
    end if
  end while
end procedure

Algorithm 2 GPU: Parallel SpMV (Warp-centric, Tiled)

procedure PARALLELMULTIPLICATIONWARPTILED($N_{rows}, csr, cols, vals, vec, res, tileStarts, tileEnds, tileRows$)
  $WARP\_SIZE \leftarrow 32$
  $threadId \leftarrow$ thread index in block
  $warpId \leftarrow threadId / WARP\_SIZE$
  $laneId \leftarrow threadId \pmod{WARP\_SIZE}$
  $tileIdx \leftarrow$ block index
  $rowBlock \leftarrow tileRows[tileIdx]$
  $blockRowStart \leftarrow rowBlock \times$ ROWS_PER_BLOCK
  $row \leftarrow blockRowStart + warpId$
  if $row \geq N_{rows}$ **then**
    return
  end if
  $colStartBlock \leftarrow tileStarts[tileIdx]$
  $colEndBlock \leftarrow tileEnds[tileIdx]$
  $colSpan \leftarrow colEndBlock - colStartBlock + 1$
  for all $tileOffset$ from 0 to $colSpan$ step BLOCK_COL_CHUNK **do**
    $tileStartCol \leftarrow colStartBlock + tileOffset$
    $tileSize \leftarrow \min($BLOCK_COL_CHUNK$, colSpan - tileOffset)$
    for all c from $threadId$ to $tileSize - 1$ step blockSize **do**
      $s\_vec[c] \leftarrow vec[tileStartCol + c]$
    end for
    SynchronizeThreadsInBlock()
    $rowStart \leftarrow csr[row]$
    $rowEnd \leftarrow csr[row+1]$
    $partialSum \leftarrow 0$
    for all j from $rowStart + laneId$ to $rowEnd - 1$ step $WARP\_SIZE$ **do**
      $col \leftarrow cols[j]$
      if $col \geq tileStartCol$ **and** $col < tileStartCol + tileSize$ **then**
        $partialSum \leftarrow partialSum + vals[j] \times s\_vec[col - tileStartCol]$
      end if
    end for
    $totalSum \leftarrow$ WarpReduceSum($partialSum$)
    if $laneId = 0$ **then**
      AtomicAdd(&res[row], $totalSum$)
    end if
  end for
end procedure

V. System Description and Experimental Set-up

A. System Description

For the development of the *Deliverable-2*, due to the time limit of five minutes on the University cluster, a personal computer has been used with these specifications:

- GCC 11.4.0
- CUDA 12.9
- MAKE 4.3
- CMake 3.28.3
- NCU 2025.2.1.0

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.

System	Processor	Cores per Socket	RAM	Accelerator
PC	24× AMD Ryzen 9 5900X CPU	12 at 4.20 GHz	64 GB	NVIDIA GeForce RTX 4090

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.

TABLE II: Dataset

Name	Rows	Columns	Non-Zeros	Sparsity	Symmetric	Type
air02	50	6,774	61,555	81.23%	No	binary
dielFilterV3real	1,102,824	1,102,824	89,306,020	99.99%	Yes	real
mawi_201512020330	226,196,185	226,196,185	480,047,894	99.99%	Yes	integer
mycielskian	98,303	98,303	100,245,742	98.96%	Yes	binary
sls	1,748,122	62,729	6,804,304	99.99%	No	real
spal_004	10,203	321,696	46,168,124	98.59%	No	real

C. Experimental Set-up

For the experimental setup, all the kernels were profiled using **ncu** using the *–set full* flag. After parsing and sorting each matrix on the GPU, two initial **warmup cycles** were applied, followed by ten **measured cycles**. Due to some issue

Algorithm 3 Merge-Path SpMV: Kernel 2 - Consume Path and Reduce

procedure CONSUMEPATHANDREDUCE($N_{rows}, N_{elem}, csr, cols, vals, vec, \ldots, res, pa$
  $tid \leftarrow$ global thread index
  $totalThreads \leftarrow$ total number of threads
  if $tid \geq totalThreads$ **then**
    return
  end if
  $start\_row \leftarrow path\_starts\_row[tid]$
  $start\_nnz \leftarrow path\_starts\_nnz[tid]$
  $end\_row \leftarrow (tid + 1 < totalThreads)?path\_starts\_row[tid + 1] : N_{rows}$
  $end\_nnz \leftarrow (tid + 1 < totalThreads)?path\_starts\_nnz[tid + 1] : N_{elem}$
  if $start\_row \geq N_{rows}$ **then**
    return
  end if
  $accumulator \leftarrow 0.0$
  if $start\_row == end\_row$ **then**
    for k from $start\_nnz$ to $end\_nnz - 1$ **do**
      $accumulator \leftarrow accumulator + vals[k] \times vec[cols[k]]$
    end for
    $partial\_sums[tid] \leftarrow accumulator$
    return
  end if
  for k from $start\_nnz$ to $csr[start\_row + 1] - 1$ **do**
    $accumulator \leftarrow accumulator + vals[k] \times vec[cols[k]]$
  end for
  $partial\_sums[tid] \leftarrow accumulator$
  for row from $start\_row + 1$ to $end\_row - 1$ **do**
    $accumulator \leftarrow 0.0$
    for k from $csr[row]$ to $csr[row + 1] - 1$ **do**
      $accumulator \leftarrow accumulator + vals[k] \times vec[cols[k]]$
    end for
    $res[row] \leftarrow accumulator$ ▷ Direct write is safe for full rows
  end for
  $accumulator \leftarrow 0.0$
  for k from $csr[end\_row]$ to $end\_nnz - 1$ **do**
    $accumulator \leftarrow accumulator + vals[k] \times vec[cols[k]]$
  end for
  if $accumulator \neq 0.0$ **then**
    AtomicAdd(&res[end_row], $accumulator$)
  end if
end procedure

with the profiling of *warpLoop* kernel when executed with *mawi_201512020330* matrix, those data are unavailable.

VI. Experimental Results

All plots have a logarithmic Y-axis scale to accurately represent GPU data of the different kernels. As it could be seen from all the figures, the *cuSPARSE* implementation outperforms the other algorithms both in terms of **execution time** and **compute throughput**.

A. Execution time

Figure 3 shows the raw execution time of each algorithm. As discussed in chapter IV, some kernels are limited due to their memory access pattern or to the needs for synchronization between threads, as the case of the *warpLoop*, *warpTiled* and *merge*. In particular, the *warpLoop* and *warpTiled* were implemented also with the goal to reduce the necessary latency due to accessing data from global memory and reusing data, but this gained performance has been limited from the thread being stalled.

B. Compute throughput

Figure 4 shows very well the difference between a state of the art algorithm like the *cuSPARSE* and the other kernels.

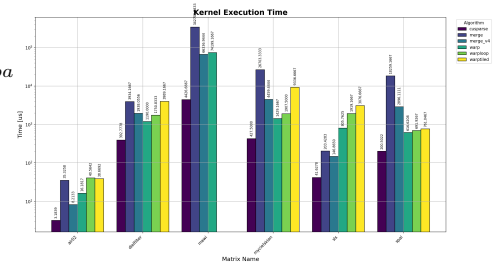

Fig. 3: Execution time SpMV

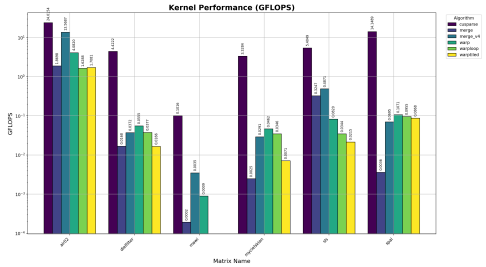

Fig. 4: Estimated FLOPs SpMV

A particular aspect to consider is the compute throughput of *warp* and *merge_v4*, in fact for the matrices that have a balanced number of non-zero per row as *dielfilter*, *mycielskian* and *spal*, the *warp* kernel has an higher throughput respect to the *merge_v4*, and viceversa. This is due to the different access pattern, one row based, the other element wise.

C. Memory throughput

Figure 5 shows that for matrices with balanced number of non-zero element per row, the algorithm with higher memory throughput is *warp*. Also from this graph is evident that the shape of the matrix influences the memory throughput, for example in the case of *sls* matrix, *warpTiled* kernel has an higher throughput respect to the *warp*, in fact the *sls* matrix is a vertical matrix with a low number of columns.

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. Results shows that there is not always a best algorithm to use, and that the peak performance

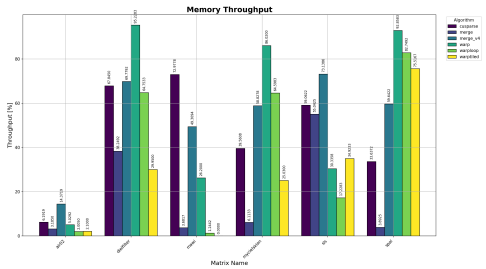

Fig. 5: Estimated memory throughput SpMV

is achieved through a deep knowledge of the interplay between algorithms, data structures and the memory stystem. By knowing these information, it is useful to select dynamically the algorithm that fits better, by changing the storing method, the memory access pattern, implementing also the use of the shared memory, but it must be taken in account also the preprocessing overhead, as in the example of the *merge path* algorithm.

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] Q.-O. E. Dufrechou E, Ezzatti P, "Selecting optimal spmv realizations for gpus via machine learning," *The International Journal of High Performance Computing Applications*, 2021.

[3] "Nvidia hopper architecture in-depth." [Online]. Available: https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/