Deliverable report

"Matteo Di Noia": Mat: 258426, matteo.dinoia@unitn.it, GitRepo: matteo.dinoia/GPU-Computing-2025-258426

***Abstract*—The sparse matrix-dense vector multiplication (SpMV) is a common linear algebra operation involving a sparse matrix and a dense vector. SpMV is widely used in many real-world applications such as scientific computing and graph analytics.**

This deliverable covers one of the possible representation used for sparse matrix, COO. It also only discuss GPU implementation without shared memory usage, using only primitive synchronization and CPU implementation. This paper discusses what are the limiting factors of COO.

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

I. Introduction

Sparse matrix vector multiplication (SpMV) is a core computation that is at the foundation of every implicit sparse linear algebra solver. It is also one of the most common primitive in scientific computing, and graph analytics, eg. it is present in software such as MatLab and Numpy.

Due to being an intrinsically memory-bound algorithm, its performance are limited by the memory bandwidth between the process and the memory, especially on highly parallelized, throughput-oriented architectures such as GPU.

II. Problem Statement

Sparse matrix vector multiplication (SpMV) is define as the matrix multiplication between a sparse matrix A of size $n \times m$ and a dense vector x of size $m \times 1$ which result in a vector y of size $n \times 1$ computed as follows:

$$y = A \cdot x$$

$$y_i = \sum_{k=1}^{n} A_{ik} x_k$$

An efficient implementation of SpMV present various challenges, most of which are related to the storage format and memory access. As said before SpMV is memory-bound and as such an efficient memory access is required to reduce the memory bottleneck. In fact the arithmetic intensity of the worst case of SpMV is:

$$Intensity = \frac{2\ flop}{6 \cdot 4\ bytes} = 0.083 \frac{flop}{bytes}$$

In order to have a efficient access to the memory a storage format that facilitate such access is required.

A. Storage Format

Multiple format exists such as Compressed Sparse Row (CSR), Coordinate (COO), ELLPACK (ELL), etc. In the present paper, the analysis is concentrated on the COO format. The latter is the simplest one in which three array are used:

- **rows (/Ys)**: representing for every i the row (or y value), in which the i-th element of the matrix is;
- **cols (/Xs)**: representing for every i the column (or x value), in which the i-th element of the matrix is
- **values**: representing for every i the value of i-th element;

Instead the CSR format, which is explained here to understand the limiting factors of COO, is still composed of the same three array but the rows array is compressed as the name implies. The compressed rows array contains pointers that mark the beginning of rows within the column array. This compression can be achieved by the fact that the points in the three array are assumed sorted by row index.

B. Parallelization

Being SpMV a conceptually easy operation, the simplest way to parallelize it is to exploit independence between operation on some blocks of data, to obtain data parallelism. In the case of SpMV, the way to do it is to execute the computation of each matrix's row concurrently. This is possible because to compute the product of a row and the vector it is only needed read access to both and write access to a single element of the result vector which is independent to the other elements in the same vector, so there is no write conflict and as such no synchronization primitive are required to preserve correctness of the algorithm. This idea can be implemented in CSR as we can get the start of each row.

In COO instead this is not easily possible. To achieve this result we either have to convert the matrix to CSR or we can find at runtime the start of each row and only after execute thread/s for each row. Both require something similar to a conversion to CSR defeating the purpose of storing the matrix as COO instead of CSR, and the latter massively increase the time as that is not an easily parallelized operation and require to scan the entirety of the Ys array before computing the SpMV.

For this reason we avoid this strategy and instead focus on the use of atomic operation to maintain correctness, in particular, with the use of $atomicAdd$.

III. State of the Art

The state of art SpMV for CPU is OpenBLAS, which is an optimized BLAS (Basic Linear Algebra Subprograms). On the other hand, cuSPARSE is a Cuda based GPU accelerated

BLAS which perform significantly faster than CPU only alternatives thanks to the higher degree of parallelization. Both of this API are massively more efficient than the algorithm presented here.

IV. Methodology and Contributions

For the analysis, a COO 1 and CSR 2 naive CPU implementations were created. Both access data in the sparse matrix A sequentially and the resulting vector Y partially sequentially, sacrificing instead the sequentiality in the access of the dense vector v.

Because of the size difference between CSR and COO, with CSR being smaller because of the compressed rows array, we expect the throughput of the latter to be greater than the one of COO as both are memory-bound application.

Algorithm 1 COO SpMV on CPU

Require: The input vectors Ar, Ay, Av representing the COO matrix of size nnz, the vector X of size $nrows$.

procedure FUNCTION(Ar, Ay, Av, X, nnz)
  for k in $\{1 \dots nnz\}$ **do**
    $Y[Ay[k]] = Y[Ay[k]] + Av[k] * X[Ax[k]]$
  end for
  return Y ▷ the result vector
end procedure

Algorithm 2 CSR SpMV on CPU

Require: The input vectors Ar, Acy, Av representing the CSR matrix of size nnz, the vector X of size $nrows$.

$k \leftarrow 0$
procedure FUNCTION(Ar, Acy, Av, X, nnz, $nrows$)
  for r in $\{1 \dots nrows\}$ **do**
    $end \leftarrow Acy[r+1]$
    while $k < end$ **do**
      $Y[r] \leftarrow Y[r] + Av[k] * X[Ax[k]]$
      $k \leftarrow k + 1$
    end while
  end for
  return Y ▷ the result vector
end procedure

After this it is possible to expand the COO implementation by simply parallelize the for and replacing the additions with a $atomicAdd$ 3 .

Algorithm 3 COO SpMV on CPU

Require: The input vectors Ar, Ay, Av representing the COO matrix of size nnz, the vector X of size $nrows$.

procedure FUNCTION(Ar, Ay, Av, X, nnz)
  for k in $\{1 \dots nnz\}$ **parallel do**
    $atomicAdd(Y[Ay[k]],\ Av[k] * X[Ax[k]])$
  end for
  return Y ▷ the result vector
end procedure

But this simplicity comes with a cost: performance. In fact, atomic operation are never cheap and as such it is expected that this algorithm will not even be close to the theoretical worst-case, instead it will be much worse.

There are some possible change that can be done to reduce cache missed and improved performance, as detailed in the following chapters.

V. System Description and Experimental Set-up

A. System Description

The previous algorithms were tested with GCC 12.3, CUDA version 12.5 on the $Baldo$ cluster in the partition $edu-short$ only on the node $edu01$. That means the program were run on a system with a "AMD EPYC 9334" 32-Core CPU and as a GPU a "NVIDIA A30", the attributes of the latter can be seen in Table I .

Metrics	Value
Peak FP32 Compute	10 TFlops
Peak Memory Bandwidth (HBM2)	933 GBs
Maximum number of threads per block	1024
Warp size	32

TABLE I: System details

B. Dataset description

To test the various implementation we use four datasets II . The largest one *Mawi*, is also the one with some row and cols that are very dense. Of medium size there is Delaunay with radial elements and Circuit5M which is a mixture of the previous two. Finally the smallest is Model7 with somewhat irregular patterns.

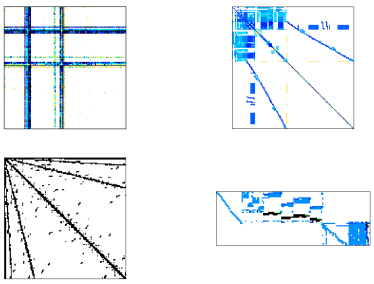

Fig. 1: In the top, the visualization of Mawi and Circuit5c and in the bottom Delaunay and Model7

Dataset	Size	Non-zero
mawi_201512020330	$226'196'185^2$	480'047'894
circuit5M_dc	$3'523'317^2$	14'865'409
delaunay_n23	$8'388'608^2$	50'331'568
model7	$3'358 \times 9'582$	51'027

TABLE II: Datasets used in the analysis

VI. Experimental Results

A. CPU implementations

Using Model7 as datasets to test cache misses, the results are the one in the Table III . These values are very rough as cachegrind doesn't allow testing of a single portion of code. As such, both test were conduct in the same exact codebase with the single difference of calling COO or CSR methods (so even the conversion from COO to CSR was done in both).

This minimize the difference between the test but doesn't solve the issue of the initialization causing some miss. To reduce the impact of this the SpMV was compute 500 times, to reduce the impact of the initial misses.

As expected, CSR being compressed cause lower reference and also slightly less cache misses.

Type	D Refs	D1 misses	LLd misses	D1 miss rate	LLd miss rate
COO	429,885,147	5,348,505	13,417	1.2%	0.0%
CSR	382,093,606	4,031,502	13,417	1.1%	0.0%

TABLE III: Valgrind's cachegrind tool results

It was then measured the average time (over 10 cycles with 2 warmup). Time wise instead we see that CSR indeed wins in Model7 and Circuit5c but massively lose in Delaunay and Mawi datasets. In the Table IV are also reported the bandwidth calculated from the average times, instead of the flops, that is because it is memory bound, so we care more about bandwidth.

Datasets	COO avg (ms)	COO (GB/s)	CSR avg (ms)	CSR (GB/s)
Model7	0.118	10.32	0.111	10.992
Delaunay	77.2	7.824	126	4.8
Circuit5c	49.8	9.252	45.4	10.152
Mawi	691	8.34	976	5.892

TABLE IV: CPU performance over datasets

Unfortunately due to time constraints and technical problem with Valgrind over the cluster, it was not possible to further analyze if these CSR loss are cause by a high cache misses. But it may be the case that with larger sample COO may be faster because of higher hit rate.

B. GPU implementations

The simple idea in Algorithm 3 can be implemented in various ways, the tested in this paper are:

- **"Kernel 1 size"**: where each thread acquires locally adjacent elements;
- **"Kernel threads size"**: where the distance between an element and the next that the thread acquire is the # threads;
- **"Kernel warp size"**: where the distance between an element and the next that the thread acquire is the size of a warp;
- **"Kernel block size"**: where the distance between an element and the next that the thread acquire is the size of a block;

It was, then, tested the performance of them, with respect to the number of thread per block and the number of blocks. As expected the first kernel performance are worst one and warp size and block size perform around the same as first contenders. This is because of the fact that warp share cache and as such threads in same thread should access close position. The reason for "Kernel threads size" to be about half the performance of the warp one it is possibly connected to the fact that each thread need to access location that are pretty far apart increasing cache misses.

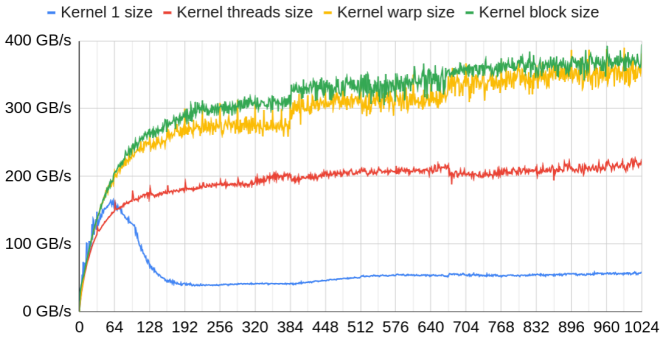

Fig. 2: Throughput over block size of the various kernels

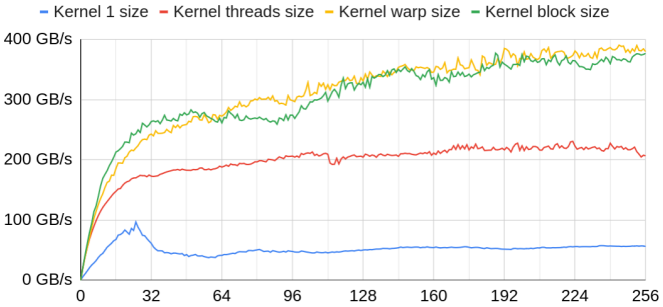

Fig. 3: Throughput over blocks number of the various kernels

Also as expected the two graph look very alike. That is because the A30 only has 4 SM with each a warp of size 32 so increasing the value above this values is the same if similar if done via block size or number of blocks.

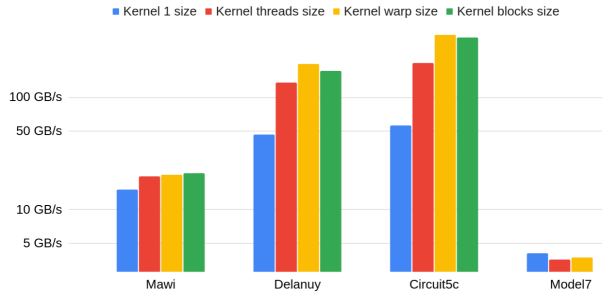

Fig. 4: Performance of the kernels based on dataset

As expected the worst performance are on the smallest dataset. That is because GPU are optimized for throughput and as such they have higher latency which is the cause of that performance. Also because it is so small the comparison between the various kernel become meaningless and more importance is put on the various overhead (eg. creating a kernel).

Finally we can node how, ignoring Model7, the ”Kernel warp size” has the most reliable performance as expected while sometimes the ”Kernel block size” has better performance but not always.

Kernel	Mawi	Delaunay	Circuit5c	Model7
Kernel 1 size	15.1 GB/s	46.7 GB/s	56.1 GB/s	4.08 GB/s
Kernel threads size GB/s	19.8 GB/s	136 GB/s	202 GB/s	3.6 GB/s
Kernel warp size	20.5 GB/s	198 GB/s	361 GB/s	3.74 GB/s
Kernel blocks size	21.1 GB/s	172 GB/s	341 GB/s	2.8 GB/s

TABLE V: Results in table format

VII. Conclusions

In conclusion COO present various problematic for parallelization, requiring atomic operation, differently from the CSR implementation. Because of this the performance obtained are even worse than the theoretical worse case, as it is possible to see in Image 5 . To be exact the best result obtained is only 37% of the theoretical worst case scenario.

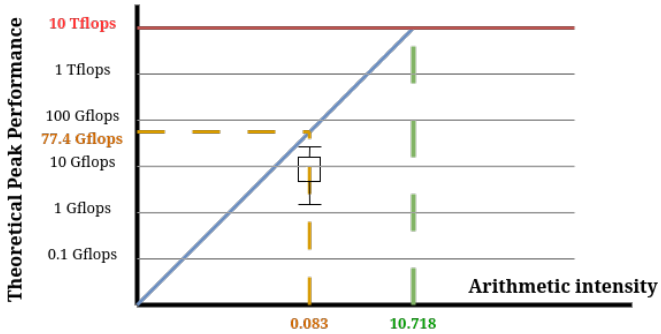

Fig. 5: Roofline model of a NVIDIA A30

As such we can say that COO is not suitable for GPU parallelization especially for a naive implementation, where CSR is best.