Deliverable report

"Diego Oniarti": Mat: 257835, diego.oniarti@studenti.unitn.it, GitRepo:
https://github.com/diego-oniarti/GPU-Computing-2025-257835

***Abstract*—Sparse matrix-dense vector multiplication (SpMV) is an operation consisting of the multiplication between a sparse matrix and a dense vector. It can be seen as a specific case of the general dot product between two matrices, where one mostly contains zeros and the other has a row size of 1.**

This deliverable discusses some of the ways these constraints on the matrices involved in the multiplication can be leveraged to improve the performance of the operation.
The techniques used will revolve around efficient cache usage and parallelization through the use of GPUs.

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

I. Introduction

SpMV is an operation that is used in many different fields like graph algorithms, graphics processing, numerical analysis, and conjugate gradients [1]. Sparse matrices appear naturally in many fields of computer science and data analysis, and the multiplication of these matrices by some vector can have different purposes and effects.

II. Problem Statement

Sparse matrix-dense vector multiplication poses many difficulties, mainly relating to the usage of memory, cache, and the efficiency of the parallelization approach.

A. Storage Format

The way the matrix is stored into memory can highly effect the performance of any algorithm iterating through its elements. Different strategies exist, each having pros and cons, and this deliverable will detail implementations that involve the Compressed Sparse Row (CSR) format.

This format represents the matrix through the use of three vectors:

1) **Values** containing all and only the nonzero values contained in the matrix. The values appear in the vector in the same order as they appear in the matrix if scanned top to bottom and left to right.
2) **Xs** containing the x indices of the nonzero elements. The elements in this vector are sorted in the same way as in the values vector, and the two can be aligned.
3) **Ys** has one element per row plus one. It contains the prefix sum of the count of nonzero values for each line.

This matrix representation saves space by not explicitly storing the nil entries of the matrix. The additional information (Xs and Ys) is negligible with respect to the size of the matrix if the values are sparse enough.

Furthermore, by storing the values in a sorted order, we can take advantage of spatial locality while iterating over the matrix.

B. Parallelization

The first approach presented will focus on the row-wise parallelization of the multiplication. This choice was made because the result in each cell of the result matrix is independent from every other cell, so there won't be any conflict or synchronization issue when computing them in parallel.
Each thread will independently iterate through it's assigned row of the matrix, multiply the elements by the corresponding ones in the vector, sum them, and store the result.

A further improvement on this technique will further parallelize the multiplication along each row by allocating multiple threads to it.

C. Cache Usage

This deliverable already showed how spatial locality is well taken advantage of during the scan of the matrix. The access to the vector, however, is mostly random, making cache hits more rare.
One of the implementations that will be shown sacrifices some of the spatial locality of the matrix to gain temporal locality on the vector. By dividing the multiplication procedure into smaller chunks, we can theoretically get better cache usage in the case highly structured matrices.

III. Methodology and Contributions

In this section I'm gonna illustrate four different implementations for the multiplication. The first one is a straightforward multiplication on the CPU, made faster by the CSR representation of the matrix. The second one tries to take better advantage of the cache by accessing the data in a different pattern. The third one works like the first, but the computation for each line is done simultaneously on the GPU. Finally, the fourth one is a further improvement on the third. In this implementation, instead of each row being assigned a single thread, it will get 32.

A. Naive CPU

This is the simplest approach which iterates through the nonzero values, uses the *x* index to indirectly access the vector, and accumulates the result for each line.

B. CPU - chunked

This implementation attempts to increase temporal-locality when accessing the vector, sacrificing some of the spatial-locality in the access to the matrix. The algorithm iterates through the vector in chunks, it then iterates through the rows and computes the product between the chunk of vector and the corresponding chunk in the line, adding the partial sum to

Algorithm 1 Naive implementation on the CPU

Require: The input vectors $vals$ and xs of size nnz, the vector ys of size $ncols$, and the vector vec to multiply the matrix with.

```
1: procedure MULTIPLY(vals, xs, ys, vec, nnz, nrows)
2:     result ← array[nrows]
3:     for i in [0 ... nrows) do
4:         result[i] ← 0
5:         for j in [ys[i] ... ys[i + 1]) do
6:             result[i] += vals[j] ∗ vec[xs[j]]
7:         end for
8:     end for
9:     return result
10: end procedure
```

the total for said row.
This approach requires the storage of the current index for each row and introduces the chunk size as an hyperparameter (8 should be a good value, ensuring the whole chuck of the vector remains cached).

Algorithm 2 Chunked CPU implementation

```
1: procedure MULTIPLY(vals, xs, ys, vec, nnz, nrows, ncols)
2:     result ← array[nrows]
3:     rc ← array[nrows]                          ▷ Row Counters
4:     for i in [0 ... nrows) do
5:         rc ← ys[i]    ▷ Counters start at the beginning of the row
6:     end for
7:     for i in [0 ... nrows) do
8:         result[i] ← 0
9:     end for
10:    for i = 0; i < ncols; i+ = span do
11:        for r in [0 ... nrows) do
12:            line_end ← ys[r + 1]
13:            while rc[r] < line_end and i ≤ xs[rc[r]] < i + span do
14:                result[r]+ = vals[rc[r]] ∗ vec[xs[rc[r]]]
15:                rc[r] + +
16:            end while
17:        end for
18:    end for
19:    return result
20: end procedure
```

C. Naive GPU

The GPU implementation uses CUDA to parallelize the product along the rows. The algorithm works exactly as **Algorithm 1**, with the difference that its ran on $nrows$ threads, and each thread only performs the inner loop.

The parameters are the same, with the exception of the additional return vector. While the CPU implementation allocated the vector inside the function, the GPU kernel receives a global array where to store the results.

Algorithm 3 Naive implementation on the GPU

```
1: procedure KERNEL(vals, xs, ys, vec, nnz, nrows, result)
2:     i ← thread_id + block_id · block_size
3:     if i >= nrows then                          ▷ Early return
4:         return
5:     end if
6:     result[i] ← 0
7:     for j in [ys[i] ... ys[i + 1]) do
8:         result[i] += vals[j] ∗ vec[xs[j]]
9:     end for
10: end procedure
```

D. Improved GPU

An improvement on the first GPU implementation consists in allocating one warp per row instead of a single thread. The threads in the warp iterate over the row with a stride equal to the warp size and store their partial sum in a buffer. A reduction is then performed to sum together the partial sums, and the thread with the lowest id finally moves the result into the result vector.

The buffer is a vector stored in global memory. A further improvement would be moving it to shared memory, but it is out of the scope of this deliverable.

IV. System Description and Experimental Set-up

Each run has been performed with 3 warm-up cycles and 10 timed runs, which are then averaged. The timing of the CPU implementation was measured with the `gettimeofday` function from the standard library, while the GPU kernels were timed using CUDA events.

Additionally, the GPU kernels have been tested with different block sizes, ranging from 32 to 1024 and doubling each time.

A. System Description

All the code has been ran on the Baldo cluster, with an AMD EPYC 9334 32-CORE Processor and a NVIDIA A30 GPU.
The CUDA version being used is 12.5.0 and the code has been compiled with NVCC 12.5. The theoretical memory bandwidth for the GPU is around 933GB/s.

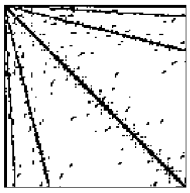
B. Dataset description

Each of the algorithms was ran on 4 different kind of matrices, with different sizes, types of data, and levels of organization.
Three of the matrices have been taken from SuiteSparse Matrix Collection, while one is generated at runtime with a completely random distribution.

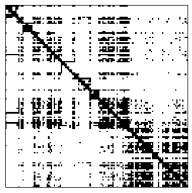
[1]https://www.cise.ufl.edu/research/sparse/matrices/LPnetlib/lp_ganges
[2]https://www.cise.ufl.edu/research/sparse/matrices/DIMACS10/delaunay_n23
[3]https://www.cise.ufl.edu/research/sparse/matrices/Kamvar/Stanford_Berkeley

name	rows	columns	nonzeros	(%)	type
lp_ganges[1]	1309	1706	6937	0.3106%	real
delaunay_n23[2]	8388608	8388608	50331568	7.152546e-5%	binary
Stanford_Berkeley[3]	683446	683446	7583376	0.001624%	binary
random	30000	20000	6001585	1.000264%	real

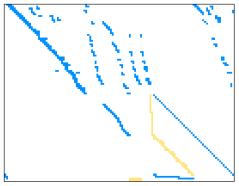
TABLE I: Test matrices



(a) Delaunay Graph



(b) Stanford-Berkeley Dataset



(c) LP-Ganges Dataset

Fig. 1: Datasets distributions

V. Experimental Results

The results immediately show that my assumptions on the chunked CPU implementation were wrong. Each run of that algorithm take up to double the time of the naive implementation on small matrices, and goes past the 5 minutes of allotted time on larger matrices.
The implementation aimed at taking advantage of temporal locality when reading the vector, sacrificing some of the spatial locality when reading the matrix. This trade-off, combined with the added complexity in iterating over the rows multiple times and handling the *row_counters* vector, has proven not to be worth it.

	Delaunay			LP_Ganges			stanford			random		
	CPU	thread	warp	CPU	thread	warp	CPU	thread	warp	CPU	thread	warp
threads per block	-	256	128	-	1024	128	-	64	128	-	1024	64
mean time (ms)	185.34	0.466	8.013	26.7	0.016	0.012	29	5.202	0.959	19.586	0.447	0.104
deviation (ms)	2.288	0.001	0.003	1.567	0	0.001	0.065	0.249	0.017	0.017	0.001	0.001
bandwidth (GB/s)	1.629	647.73	37.69	0.004	6.873	9.646	2.342	13.239	71.791	2.464	108	462.88
FLOPS	271560	107955022	6280994	519	846801	1188493	515870	2915663	15810505	612323	26843502	115047352

TABLE II: Experimental results

Table II shows the results of the three working algorithms on each dataset. The GPU implementations have been tested with different block sizes, but only the ones with the best results are shown.

The bandwidth of each algorithm, as shown in **Figure 2**, mostly follows a clear trend, with the increasing amount of parallelization leading to better performances.

The only outlier in this trend is the Delaunay dataset, with the reason most likely being the first row of the graph. Looking at the distribution of the values in **Figure 1a**, it is visible that

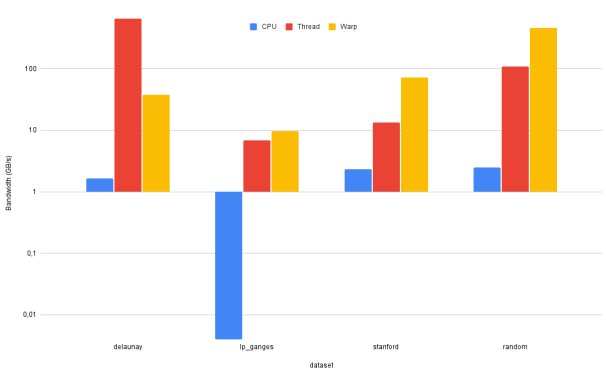


Fig. 2: Bandwidths of the algorithms on the different datasets

the first row of the graph is dense. This fact can possibly bring the second algorithm to take better advantage of the cache, beating the advantage that the third one would have had.

Lastly, *cachegrind* has been used to measure the amount of cache misses. On the random matrix the naive CPU implementation gets 0.6% D1 miss rate and 0.1% LL miss rate. These results confirm that simply using the CSR format ensures a good usage of the cache.

VI. Conclusions

The use of the CSR format to store the matrix in memory has proven to be a useful tool to optimize the SpMV operation. The ordering of the elements based on their position in the matrix allows for a sequential access that is predictable and that can take good advantage of the cache.

Parallelization was shown to be orders of magnitude faster than the sequential operation on all datasets, with varying results on the kind of parallelization that yield the better performance.

Lastly, the attempt to find a better memory access pattern was unsuccessful. The trade-offs made when using the chunk based approach were likely too disadvantageous to bring any improvement.

A. Future works

As discussed earlier, there is a trend of improvement when moving from the first GPU implementation to the second one, with the exception of the results taken on the Delaunay Graph dataset. Further testing on different kinds of matrices could help determine whether this was a one-of occurrence or a common one.

Another improvement would be to use more threads for each row of the matrix being processed. This can however complicate the synchronization between threads if the implementation is not well thought out

Lastly, the use of global memory to store the buffer with the partial results of operations is an obvious slowdown. Future implementations would use shared memory and a better reduction algorithm to add together the partial sums.

REFERENCES

[1] Y. Zhuo, X.-L. Wu, J. P. Haldar, T. Marin, W. mei W. Hwu, Z.-P. Liang, and B. P. Sutton, "Chapter 44 - using gpus to accelerate advanced mri reconstruction with field inhomogeneity compensation," in *GPU Computing Gems Emerald Edition*, ser. Applications of GPU Computing Series, W. mei W. Hwu, Ed. Boston: Morgan Kaufmann, 2011, pp. 709–722. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123849885000449