Deliverable report 2

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

***Abstract*—This deliverable proposes improvements on the implementation shown in the previous one.**
Two different implementations will be shown, the first simply addressing the more glaring issues of the old kernel through the use of shuffle operations and other CUDA features. The second one will rely heavily on shared memory to increase the kernel performance on structured matrices with higher local density of non-zeros.

***Index Terms*—Sparse Matrix, SpMV, CUDA, Parallelization, Storage Format, Shared Memory, Shuffle Operation**

I. Introduction

The implementation proposed in the previous deliverable mapped each row of the matrix to a single warp. The threads in each warp then iterated through the row with a stride equal to the warp size to guarantee coalesced memory access to some of the data structures in used. However, due to the indirect access to the vector x used in the multiplication, many of the accesses were not coalesced. The solutions in this deliverable will try to attenuate this problem to improve performance.

The results will be compared with the ones from the previous work, as well as the state of the art solutions to the SpMV problem.

II. Problem Statement

The two main CUDA features that are gonna be explored in this deliverable are shuffle operations and shared memory, but some other smaller optimization will be used as well.

A. Shared Memory

Shared memory is a region of memory present on the GPU which can be accessed with latency orders of magnitude lower than global memory. This makes shared memory a perfect instrument to build user-defined caches (the use we're going to make of it), since data can be loaded from global memory once into shared and then more quickly be accessed from there.

The two main drawbacks of shared memory are the size and bank conflicts. The first is an issue because the size of shared memory is relatively small, forcing the developer to decide which bits of data are worth transferring to shared memory and which ones are not.
Shared memory is divided into banks which can be accessed simultaneously by different threads. If more threads try to access the same bank however (a conflict), the operations get serialized, lowering the effective throughput. It is important to avoid this scenario whenever possible. [1]

B. Shuffle Operations

Warp shuffle functions are used to exchange and share variables between threads within the same warp.
This allows for single variables to be exchanged simultaneously for all threads without relying on shared or global memory. [2]

III. Methodology and Contributions

In this section I'm going to briefly present the best implementation from the previous deliverable (warp-per-row), with its strengths and weaknesses. This is gonna be followed by an improved version that makes minimal changes and uses the shuffle functions to improve performance. The third version of the kernel will be a further improvement based on shared memory.

A. warp-per-row

Algorithm 1 warp-per-row

```
1:  procedure KERNEL(vals, xs, ys, vec, nnz, nrows, result, buffer)
2:      id ← block_id · block_dim + thread_id
3:      wid ← thread_id/warp_size
4:      lane ← thread_id%warp_size
5:      row ← block_id · block_size/warp_size + wid
6:      buffer[tid] ← 0    ▷ Always initiated to 0 for safety
7:      if row < nrows then
8:          start ← ys[row]
9:          end ← ys[row + 1]
10:         sum ← 0
11:         //Iterate through the assigned row
12:         for i = start + lane; i < end; i+ = 32 do
13:             sum+ = vals[i] · vec[xs[i]]
14:         end for
15:         buffer[id] ← sum
16:     end if
17:     // Reduction
18:     for s = 1; s < warp_size; s = s^2 do
19:         __syncthreads()
20:         if id&(s^2 − 1) == 0 then
21:             buffer[tid]+ = buffer[tid + s]
22:         end if
23:     end for
24:     if lane == 0 ∧ row < nrows then
25:         ret[row] ← buffer[id]
26:     end if
27: end procedure
```

In this implementation we have a whole warp (32 threads in this case) working on each row of the matrix. Each thread calculates the partial sum of the elements that are assigned to it and stores it in a common buffer. A reduction of the elements in the buffer is then performed cooperatively by all the threads in the warp. Finally the thread with the lowest id moves the total sum in the result vector.

The access to the $vals$ and xs vectors is coalesced thanks to the stride being equal to the warp size, but the indirect access to the dense vector is not, tanking performance.

The reduction used to sum the partial results is performed cooperatively, making it faster than a linear scan through the buffer, but it still requires repeated access to a global memory buffer, which has high latency.

B. Improved

Algorithm 2 improved

```
1:  procedure KERNEL(vals, xs, ys, vec, nrows, result)
2:      //Same variable initialization
3:      if row ≥ rows then                          ▷ Early return
4:          return
5:      end if
6:      start ← ys[row]
7:      end ← ys[row + 1]
8:      sum = 0
9:      for i = start + lane; i < end; i+ = 32 do
10:         sum+ = vals[i] · __ldg(&vec[xs[i]])
11:     end for
12:     //Reduction
13:     for off = 16; off > 0; off = √off do
14:         mask ← 0xffffffff
15:         sum+ = __shfl_down_sync(mask, sum, off)
16:     end for
17:     if lane == 0 then
18:         ret[row] ← buffer[id]
19:     end if
20: end procedure
```

IV. System Description and Experimental Set-up

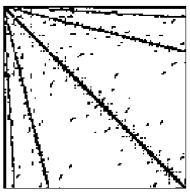
Each run has been performed with 3 warm-up cycles and 10 timed runs, which are then averaged. The timing of the CPU implementation was measured with the `gettimeofday` function from the standard library, while the GPU kernels were timed using CUDA events.

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

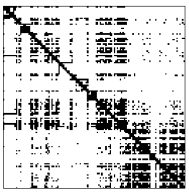
A. System Description

All the code has been ran on the Baldo cluster, with an AMD EPYC 9334 32-CORE Processor and a NVIDIA A30 GPU.

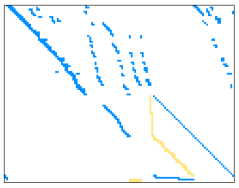
The CUDA version being used is 12.5.0 and the code has been compiled with NVCC 12.5. The theoretical memory bandwidth for the GPU is around 933GB/s.



(a) Delaunay Graph



(b) Stanford-Berkeley Dataset



(c) LP-Ganges Dataset

Fig. 1: Datasets distributions

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.

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

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.

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.

[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

	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

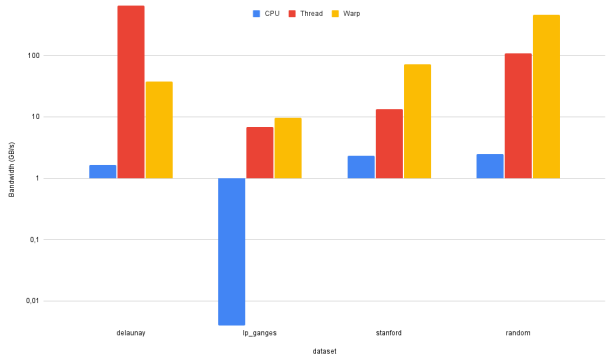


Fig. 2: Bandwidths of the algorithms on the different datasets

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 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] M. Harris. Using shared memory in cuda c/c++. [Online]. Available: https://developer.nvidia.com/blog/using-shared-memory-cuda-cc

[2] Warp shuffle functions. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/