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:         result[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 kernel

```
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:         result[row] ← buffer[id]
19:     end if
20: end procedure
```

The biggest improvement in this version of the kernel is the dropping of the global memory buffer. The reduction is performed through the repeated use of the `__shfl_down_sync` operation instead, whose working is better visualized in **Fig. 1**.

In the first step, the first $warp\_size/2$ threads sum their partial sums with those of the thread $warp\_size/2$ to the right. This process is repeated until the first thread gets the total sum. This approach would not be possible if we used more threads, since shuffle functions only operate within single warps.

Since this loop always takes a constant number of loops ($log_2(warp\_size)$) it could be manually unrolled to gain more performance. Writing it like this however makes it more readable. Furthermore, if the `-O3` flag is set in the compiler, it can automatically unroll loops of this size.

Another improvement on the base implementation is the use of `__ldg()` when accessing the vector. This function informs the compiler that the data being accessed is read-only, and can thus be stored in the read-only cache. [3] Caching portions of the vector makes the access to contiguous parts of it faster, which can be beneficial when working with highly structured matrices.

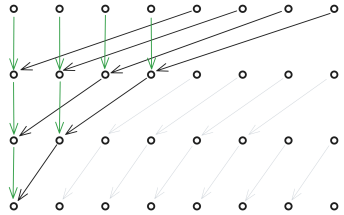


Fig. 1: Steps of a reduction of eight elements.

The last improvement made here is flagging all pointer parameters with the `__restrict__` keyword [4]. This gives more information to the compiler, informing it that there is no aliasing between the pointers and allowing it to perform better optimizations.

C. Adding Shared Memory

The easiest and most efficient way of improving the kernel with shared memory would be to let each block load the entire vector into shared memory. This is however made impossible by the limited size of the shared memory. Another approach would involve only loading the most frequently used elements of the vector into shared memory, increasing performance during the multiplication but requiring a large overhead to determine the parts of the vector to transfer.

The approach I chose in the end is one based on tiling. A portion of the shared memory is reserved to each warp, and the rows of the matrix are divided in *tiles* of constant size. For each non-empty tile of the row, the threads cooperatively load the corresponding tile of the vector into shared memory. The multiplication is then performed as usual for the elements of the tile, before moving to the subsequent one.

After running through the whole row, a reduction like that explained for **Algo. 2** is performed, and the result is saved to the output vector.

This solution targets heavily matrices with an high degree of locality. If data is too sparse or homogeneously distributed, it is likely that each tile of the vector will only contain few useful items. Tiles are also 1 tall, meaning that threads in one row cannot take advantage of elements cached in shared memory by the rows above or below it.

The best case scenario for this kernel then is a matrix in which elements appear as horizontal clusters, as will be discussed in the **Sec. V**.

IV. System Description and Experimental Set-up

Like in the first deliverable, 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.

Algorithm 3 shared memory kernel

```
define TILE 128
procedure KERNEL(vals, xs, ys, vec, nrows, result)
    //Same variable initialization
    if row ≥ rows then return                          ▷ Early return
    end if
    extern __shared__ data_t smem[]
    s_vec = smem + wid * TILE     ▷ Each warp gets its tile of the shared memory
    start ← ys[row]
    end ← ys[row + 1]
    sum = 0
    if start ≥ end then                                ▷ The row is empty
        if lane == 0 then result[row] ← 0
        end if
    end if
    cur ← start     ▷ index of the current element in the row being examined
    while cur < end do
        tile_start ← __ldg(&xs[cur])
        //Get the tile containing the current item
        tileBase ← tile_start/TILE * TILE ▷ Exploit rounding
        tileEnd ← min(tileBase + TILE, end)
        tileW ← tileEnd − tileBase                     ▷ Tile width
        //Load the tile into shared memory
        for t = lane; t < tileW; t+ = warp_size do
            s_vec[t] ← __ldg(&vec[tileBase + t])
        end for
        __syncwarp()
        //Process the elements that fall in the tile
        j ← cur + lane
        while j < end do
            c ← __ldg(&xs[j])
            if c ≥ tileEnd then break
            end if
            sum+ = vals[j] · s_vec[c − tileBase]
            j+ = 32
        end while
        //Find the next element past the tile (min of js)
        next ← j
        for off = 16; off > 0; off = √off do
            next ← min( next, shfl_down(next, off) )
        end for
        cur ← next
        __syncwart()
    end while
     //Perform reduction as in previous kernel
end procedure
```

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

The relevant tests from the previous deliverable have been re-run to eliminate any variable that may have occurred since and make the comparison more accurate.

cuSPARSE has been benchmarked with the same number of runs and pre-runs, but giving the library control over the number of threads and blocks used. To make the comparison more fair the matrix is provided to cuSPARSE in the CSR format as well, and the algorithm used is `CUSPARSE_SPMV_ALG_DEFAULT`.

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.

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

Some relevant considerations can be made by looking at **Table. I** and **Fig. 2**

- LP_Ganges is by far the smallest, both in terms of size and number of elements
- Standford-Barkeley is the most structured, with an high degree of locality of the non-zeros
- The Delaunay also has clear patterns in it, but with few horizontal streaks

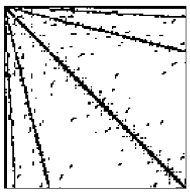
V. Experimental Results

The experimental results broadly mirror the expectations set in **Sec. III**. The full data can be seen in **Table II** but can be better visualized in **Fig. 3**.

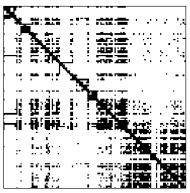
A. Improved Kernel

The improved version of the kernel, making use of shuffle functions and the `__ldg` intrinsic beats the baseline in all tests, as expected. The implementation is still very close to the one from the first deliverable, with the only changes being clear improvements.
Not relying on the buffer when performing the reduction has proven to be a big step ahead.

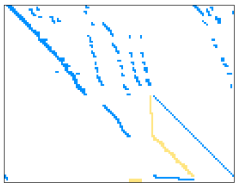
[1] https://www.cise.ufl.edu/research/sparse/matrices/LPnetlib/lp_ganges
[2] https://www.cise.ufl.edu/research/sparse/matrices/DIMACS10/delaunay_n23
[3] http://www.cise.ufl.edu/research/sparse/matrices/Kamvar/Stanford_Berkeley



(a) Delaunay Graph



(b) Stanford-Berkeley Dataset



(c) LP-Ganges Dataset

Fig. 2: Datasets distributions

	Delaunay				LP_Ganges			
	old	shuffle	shared	cuSPARSE	old	reduction	shared	cuSPARSE
threads per block	128	256	256	-	512	1024	64	-
mean time (ms)	7.478	2.09	2.092	0.29	0.01	0.009	0.012	0.014
deviation (ms)	0.003	0.001	0.001	0.001	0	0.001	0.001	0.001
bandwidth (GB/s)	40.386	144.487	144.359	1040.253	7.254	8.463	6.128	5.114
FLOPS	6731019	24081095	24059877	173375384	1382533	1612955	1168002	976735

	Stanford				random			
	old	reduction	shared	cuSPARSE	old	reduction	shared	cuSPARSE
threads per block	1024	256	256	-	256	256	64	-
mean time (ms)	0.774	0.125	0.126	0.086	0.059	0.025	0.342	0.027
deviation (ms)	0.136	0	0	0.001	0.001	0	0.001	0
bandwidth (GB/s)	89.031	551.717	547.227	828.254	86.572	206.414	14.938	190.657
FLOPS	19607203	121503540	120514895	182404945	20289007	48375031	3500824	44682282

TABLE II: Experimental results

B. Outlier

The graph clearly shows how the bandwidth of all kernels is much lower one the lp_ganges dataset and does not follow the general trend, this being likely due to the very small size of the matrix. This makes it so that the overheads introduced by each approach become relatively more significant, lowering the performance per non-zero element.

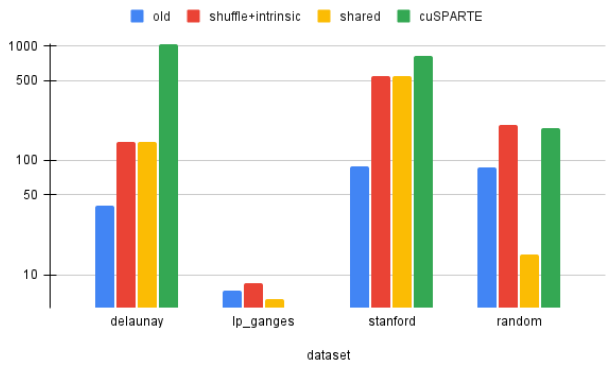


Fig. 3: Bandwidths of the algorithms on all different datasets

This abnormality is visible even in the cuSPARSE results, where the library designed around very large matrices struggles with such a small dataset.

C. Shared Memory

As it was previously hypothesized, the performance of the kernel using shared memory is directly tied to how structured the matrix is.
The random matrix is the worse case scenario for this approach since the data will on average be more spread out. The Delaunay dataset is more structured, with lines clearly visible in **Fig. 2a**. The results are better than those of the random matrix, but are still limited due to the varying directions of the lines of data.
Finally, the best results for this approach come from the Stanford-Berkeley dataset. Except the major diagonal being filled with non-zeros, most of the other ones are clustered in square patterns. This is one of the best cases for this kernel, since each time a tile gets loaded into shared memory there is a good chance most of it's elements are gonna be used in the multiplication.

VI. Conclusions

The use of shuffle functions was proved to be a big improvement whenever performing a reduction between elements in the same warp, both due to their speed and their inherent synchronicity.

This deliverable did not prove a definitive and clear-cut gain in performance whenever using shared memory. This is because its use is non-trivial, and there are many improvements to the algorithm that will be discussed in **Sec. VI-A**.
The experimental validate however the assumption initially made that the performance would be directly tied to the locality of data in the matrix and its pattern of distribution.

A. Future works

The kernel making use of shared memory can be improved in different ways, only a few of which I am going to cite.

1) The load of the operations could be better balanced. As of now, each row gets the same amount of threads regardless of it's number of non-zero elements. More threads could be directed instead to the rows containing more elements, but this would require better planning and take synchronization into account.
2) The tiles could be expanded on the vertical axis. If a warp took up multiple rows for example (or implementing other access patterns) it could load a rectangular tile into shared memory, taking advantage of data locality along both axis instead of one.
3) Lastly, as an extension of point 2, the locality of the matrix could be artificially inflated by reordering the rows. An initial rearrangement of the rows could be done at the start of the operation to bring together the rows that frequently access the same parts of the vector. After the multiplication has been done this operation can be reverted to get the correct result.

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/index.html#warp-shuffle-functions

[3] Read-only data cache load function. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#read-only-data-cache-load-function

[4] Cuda programming guide: __restrict__. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/#restrict