# GPU Computing
## Homework 2: Matrix Transposition

Murtas Cristian
248025
cristian.murtas@studenti.unitn.it
GitHub Repository

## Abstract

*This work focuses on the implementation of different algorithms for matrix transposition on the CPU and on the GPU. First, a formal definition of the problem is provided and followed by the description of the pseudocodes. On the CPU side the performance of the algorithms has been evaluated using different optimization flags, while the GPU side has been evaluated using different kernel configurations. Through the experiments, the performance of the algorithms has been evaluated. In particular, the attention has been put on diverse metrics such as the effective bandwidth and the cache behavior (for the CPU case). In the end, the results have been analyzed and discussed.*

## 1. Problem Description

The goal of this homework is to implement a matrix transposition algorithm, where the transpose of a matrix $A$ is an operation that flips $A$ over its diagonal. If $A$ is of size $m \times n$ and $e$ is the element at row $i$ and column $j$, then the transpose of $A$ - also denoted as $A^{\mathrm{T}}$ - is a matrix of size $n \times m$, where $e$ is at row $j$ and column $i$.
Additionally, we are asked to measure the effective bandwidth of our implementation, also considering the usage of different optimization flags, such as: `-O0`, `-O1`, `-O2`, `-O3`. Furthermore, an analysis of the cache behavior related to the algorithm is required and, for this purpose, we are going to use valgrind.

### 1.1. Algorithms

Two algorithms have been implemented. The first one is a naïve approach, which consists in iterating over the *src* matrix, reading the data at position $src[i][j]$ and writing it at position $dest[j][i]$. The second algorithm is a more optimized version, which takes advantage of a block mechanism to reduce the number of cache misses.

**Algorithm 1** Naïve Matrix Transposition

```
1: src ← create_matrix(size)
2: for i = 0 to size do
3:     for j = 0 to size do
4:         dest[j * size + i] = src[i * size + j]
5:     end for
6: end for
```

**Algorithm 2** Matrix Transposition with Blocking

```
1: src ← create_matrix(size)
2: for i = 0 to size increase i by block_size do
3:     for j = 0 to size increase j by block_size do
4:         for k = i to i + block_size do    ▷ Loop over the current block
5:             for l = j to j + block_size do
6:                 dest[l * size + k] = src[k * size + l]
7:             end for
8:         end for
9:     end for
10: end for
```

## 2. Experimental Setup

### 2.1. Hardware

The entire experiment has been conducted on the Marzola cluster of the University of Trento, which has an Intel Xeon Silver 4309y [3] and NVIDIA A30 GPUs [5]. The code has also been tested on a local machine equipped with an AMD Ryzen 5 5600X [1] and an NVIDIA RTX 3070 [2]. For both setup, the information about the GPUs have been retrieved using the *cudaGetDeviceProperties* function [4].

## 2.2. Results

To provide a broader and precise analysis, the results have been collected executing both algorithms on matrices of different sizes, ranging from $4 \times 4$ to $4096 \times 4096$. Also, for each dimension, the algorithms have been executed multiple times (100) to further improve the measurements.
The effective bandwidth (*EB*) has been calculated using the formula below:

$$EB = \frac{(2 * dimension * dimension * 4)}{execution\,time} \left[\frac{GB}{s}\right]$$

The factor *2* is because we are both reading and writing the matrix; the *dimension* is the size of the matrix; *4* is the size of a single floating element in bytes and the *execution time* is the average time of the 100 executions.

**2.2.1. Naïve Algorithm.** The obtained results are in line with the expectations. Indeed, the executions with `-O0` don't show any significant difference because the compiler doesn't apply any optimization. Meanwhile, the executions with `-O1`, `-O2` and `-O3` show a significant improvement in the performance, on all the tested platforms. This is probably due to the fact that the compiler is able to apply some optimizations, such as loop unrolling.
Another interesting aspect is that both the Ryzen 5 and the Xeon start to show a decrease in the performance when the matrix size is greater than $64 \times 64$, as shown in figure 1.
To understand this behavior, the cache miss rate has been analyzed using valgrind (table 1).

| **Ryzen 5 5600X** | | |
|---|---|---|
| **Dimension** | **I1 miss rate** | **D1 miss rate** |
| 64 | 0.01% | 0.4% |
| 128 | 0.00% | 4.0% |
| **Xeon 4309y** | | |
| **Dimension** | **I1 miss rate** | **D1 miss rate** |
| 64 | 0.01% | 0.1% |
| 128 | 0.00% | 3.6% |

**Table 1. Cache miss rate of the naïve algorithm**

**2.2.2. Blocked Algorithm.** The idea behind the blocked algorithm is to reduce the number of cache misses by exploiting the locality of the data. But in order to benefit from this the block size must be chosen with care, since it also depends on the size of the cache line. Different dimensions have been used to test the algorithm and find the best one. In this step, we used a matrix of $1024 \times 1024$ and set the optimization flag to `-O3`. The results for the used platforms are shown in table 2, where the highlighted rows indicate the most appropriate block size for the given processor.
Subsequently, as shown in figure 2, the results are pretty evident. Indeed, the blocked algorithm outperforms the naïve one on both platforms.
The cache miss rate analysis highlights how the blocked algorithm is able to exploit the temporal locality of the data. The results of the analysis are shown in table 3.

| **Ryzen 5 5600X** | |
|---|---|
| **Blocksize** | **Bandwidth (GB/s)** |
| 2 | 4.85 |
| 4 | 7.49 |
| 8 | 3.47 |
| 16 | 3.62 |
| 32 | 3.33 |
| 64 | 2.75 |
| **Xeon 4309y** | |
| **Blocksize** | **Bandwidth (GB/s)** |
| 2 | 2.04 |
| 4 | 3.83 |
| 8 | 4.80 |
| 16 | 2.45 |
| 32 | 2.59 |
| 64 | 2.72 |

**Table 2. Results of the blocked algorithm with different block sizes (**$1024 \times 1024$**, `-O3`)**

| **Ryzen 5 5600X** | | | |
|---|---|---|---|
| **Algorithm** | **Dimension** | **I1 m.r.** | **D1 m.r.** |
| Naïve | 1024 | 0.00% | 4.0% |
| Blocked | 1024 | 0.00% | 1.4% |
| **Xeon 4309y** | | | |
| **Algorithm** | **Dimension** | **I1 m.r.** | **D1 m.r.** |
| Naïve | 1024 | 0.01% | 3.7% |
| Blocked | 1024 | 0.00% | 1.0% |

**Table 3. Comparison of the cache miss rate between the naïve and the blocked algorithm, `-O3`**

# 3. Conclusion

To conclude, if provided with an adequate block size, the blocked algorithm is able to achieve an higher effective bandwidth than the naïve one. These results are concordant with the outcomes of the cache analysis.

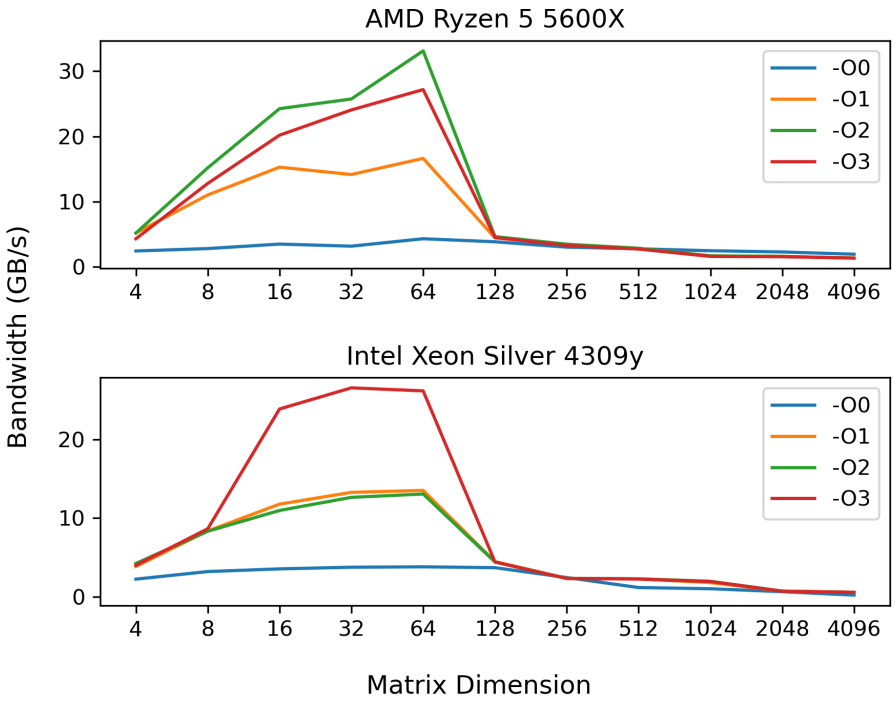


**Figure 1. Comparison of the naïve matrix transposition algorithm**

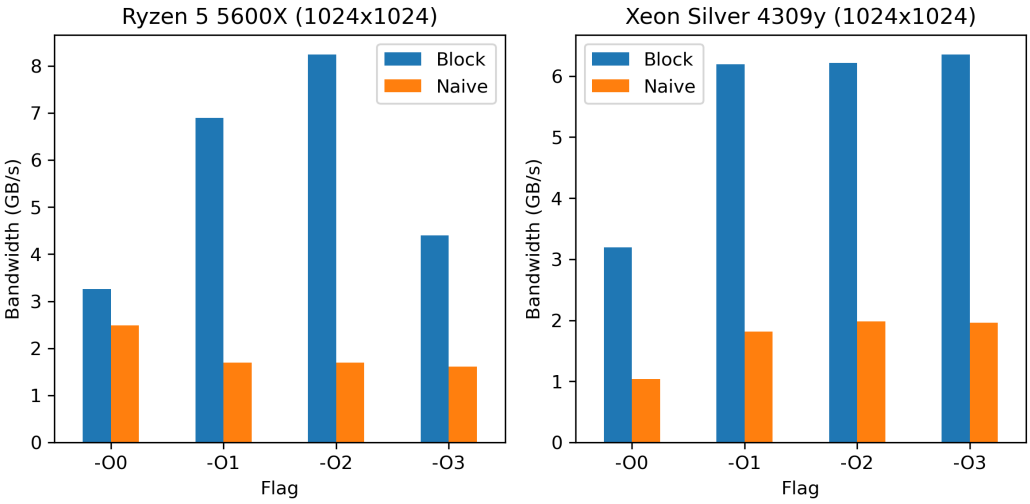


**Figure 2. Comparison of the blocked algorithm against the naïve algorithm**

A future direction to further improve the performance could be to parallelize the algorithms. A possible solution could ideally employ the usage of one thread for each cell of the matrix. Of course, this approach would require a high number of threads, therefore being impractical for large matrices. Instead, a better solution could be to divide the matrix into blocks and assign a thread to each of them.

# Part 2: CUDA Implementation

## 4. CUDA Algorithms

During this section, the three proposed algorithms are going to be introduced along with their pseudocode. For all the implementations, the kernel configuration has been calculated as follows:

dim3 block(TILE_DIM, TILE_DIM, 1)

dim3 grid(side / TILE_DIM, side / TILE_DIM, 1)

### 4.1. Naïve Transposition

In the *NaiveTranspose* algorithm, the element at position $[row][col]$ of the source matrix is read and written into the destination matrix at position $[col][row]$. Note that while the accesses to the *src* matrix are coalesced, the accesses to the *dst* matrix are not. Indeed, the write operations to *dst* have a stride equal to the width of the matrix.

**Algorithm 3** Naive Matrix Transposition

```
1: procedure NAIVETRANSPOSE(src, dst, width)
2:     row ← blockIdx.x × blockDim.x + threadIdx.x
3:     col ← blockIdx.y × blockDim.y + threadIdx.y
4:     if row ≤ width && col ≤ width then
5:         dst[row × width + col] = src[col × width + row]
6:     end if
7: end procedure
```

### 4.2. Transposition with Coalesced Memory

The transposition with coalesced memory exploits the usage of the shared memory to store a tile of the matrix, avoiding the large strides through the global memory.
A downside of this approach is that the usage of shared memory could lead to memory bank conflicts.

### 4.3. Transposition with Coalesced Memory and No Bank Conflicts

As mentioned above, a wrong access pattern to the shared memory could result in bank conflicts, which can slow down the execution of the kernel. To avoid this conflict, the shared memory can be padded with an additional column.

**Algorithm 4** Matrix Transpose with Shared Memory

```
1: procedure TR_COALESCED(src, dst, width)
2:     tile[TILE_DIM][TILE_DIM]
3:     row ← blockIdx.x × TILE_DIM + threadIdx.x
4:     col ← blockIdx.y × TILE_DIM + threadIdx.y
5:     if row ≤ width && col ≤ width then ▷ copy to shared memory
6:         tile[tIdx.y][tIdx.x] = src[col × width + row]
7:     end if
8:     __syncthreads()
9:     row ← blockIdx.x × TILE_DIM + threadIdx.x
10:    col ← blockIdx.y × TILE_DIM + threadIdx.y
11:    if row ≤ width && col ≤ width then
12:        dst[col × width + row] = tile[tIdx.y][tIdx.x]
13:    end if
14: end procedure
```

**Algorithm 5** Matrix Transpose with Shared Memory with no Bank Conflicts

```
1: procedure TR_COALESCED(src, dst, width)
2:     tile[TILE_DIM][TILE_DIM + 1]
3:     row ← blockIdx.x × TILE_DIM + threadIdx.x
4:     col ← blockIdx.y × TILE_DIM + threadIdx.y
5:     if row ≤ width && col ≤ width then ▷ copy to shared memory
6:         tile[tIdx.y][tIdx.x] = src[col × width + row]
7:     end if
8:     __syncthreads()
9:     row ← blockIdx.x × TILE_DIM + threadIdx.x
10:    col ← blockIdx.y × TILE_DIM + threadIdx.y
11:    if row ≤ width && col ≤ width then
12:        dst[col × width + row] = tile[tIdx.y][tIdx.x]
13:    end if
14: end procedure
```

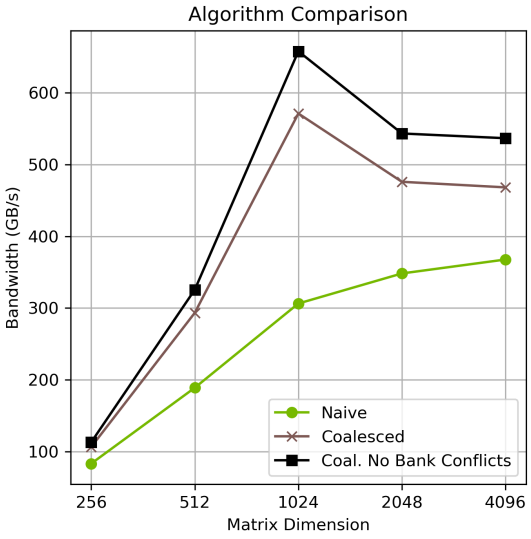


**Figure 3. Comparison of the different algorithms**

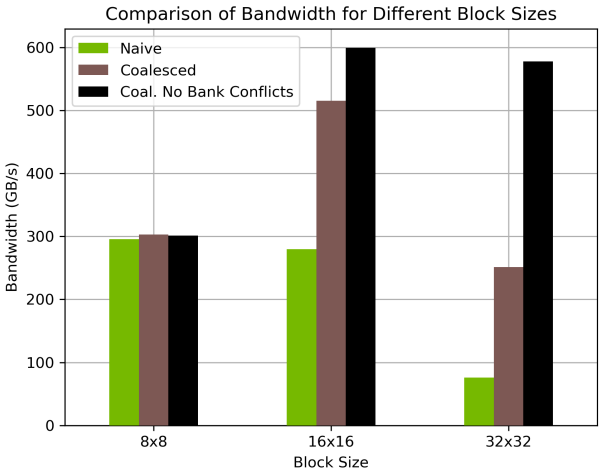


**Figure 4. Comparison of the different block sizes**

## 5. Results

The bandwidth measurements have been performed on the NVIDIA A30 mentioned in section 2.1. The code was compiled with CUDA 12.2.
The peak bandwidth used for comparison has been retrieved using the *cudaDeviceProp* structure [4] and the obtained value is 933.1 GB/s.

To evaluate the performance, the algorithms have been tested using various dimensions of the matrix ($256 \times 256$ to $4096 \times 4096$) and the execution of the kernels has been repeated 100 times to improve the accuracy of the measurements. Moreover, the kernel configuration has been set accordingly as stated in section 4, using a block size of $16 \times 16$. As depicted in figure 3, due to the use of the shared memory, both coalesced algorithms outperform the naïve one.

Additionally, an analysis has been conducted to evaluate the performance of the algorithms with respect to different block sizes. The dimension of the matrix has been set to $1024 \times 1024$.
In figure 4, the outcome shows that the $16 \times 16$ block size is the most efficient for all the algorithm. With a block size of $32 \times 32$ the behavior is more interesting. In particular, the algorithm with no bank conflicts is not influenced by its size, while the simple coleasced algorithm shows a significant decrease in the performance. This is probably due to a high number of bank conflicts, resulting in a lot of accesses to the shared memory that need to be serialized. Even the naïve algorithm drops in performance, likely because of the high number of accesses to the global memory.

## 6. Conclusion

To wrap up, three kernels - that represent different approaches to the matrix transposition problem - have been implemented. This work showed how to exploit the shared memory to coalesce the accesses to the global memory, attempting to enhance the performances.
From the previous results (section 5), we can conclude that both coalesced algorithms benefit from the shared memory, achieving a better effective bandwidth than the naïve one. Also, the block size plays an important role in tuning the performance.

Coming towards the end, it is worth to mention the speed up gain with respect to the CPU implementation, discussed in the first homework. Although some small matrices are faster to transpose on the CPU, the GPU is definitely able to surmount the CPU when dealing with larger matrices. Despite that, there's still room for improvement. Indeed, the algorithms could be further optimized by leveraging *Warp-level primitives* for efficient inter-thread communication. Lastly, since the most recent generations of GPU are equipped with *tensor cores*, a possible future direction could be to exploit them to achieve better performances.

## References


[1] AMD. Ryzen 5 5600x specifications. https://www.techpowerup.com/cpu-specs/ryzen-5-5600x.c2365, 2020.

[2] GigaByte. Rtx 3070 specifications. https://www.gigabyte.com/Graphics-Card/GV-N3070GAMING-OC-8GD-rev-20/sp#sp, 2020.

[3] Intel. Intel xeon silver 4309y specifications. https://www.intel.com/content/www/us/en/products/sku/215275/intel-xeon-silver-4309y-processor-12m-cache-2-80-ghz/specifications.html, 2021.

[4] NVIDIA. Cuda documentation: cudagetdeviceproperties. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html#group__CUDART__DEVICE_1g1bf9d625a931d657e08db2b4391170f0, 2020.

[5] NVIDIA. Nvidia a30 datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/data-center/products/a30-gpu/pdf/a30-datasheet.pdf, 2022.