GPU Computing

Homework 1: Matrix Transposition

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

April 26, 2024

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^{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: $src \leftarrow create\_matrix(size)$
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: $src \leftarrow create\_matrix(size)$
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

1. **Desktop PC**
   - **CPU**: AMD Ryzen 5 5600X
     - **Cores**: 6
     - **Threads**: 12
     - **Base Clock**: 3.7 GHz
     - **L1 Cache**: 64 KB (per core)
     - **L2 Cache**: 512 KB (per core)
     - **L3 Cache**: 32 MB
   - **RAM**: 16 GB DDR4
   - **OS**: Ubuntu 22.04 (on WSL2)
   - **Compiler**: G++ 11.4.0

2. **Marzola Cluster**
   - **CPU**: Intel Xeon Silver 4309y
     - **Cores**: 4
     - **Threads**: 8
     - **Base Clock**: 2.80 GHz
     - **L1 Cache**: 32 KB (per core)
     - **L2 Cache**: 1 MB (per core)
     - **L3 Cache**: 12 MB
   - **RAM**: 257 GB
   - **OS**: Rocky Linux 8.7
   - **Compiler**: G++ 8.5.0

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 4×4 to 4096×4096. Also, for each dimension, the algorithms have been executed multiple times (100) to further improve the measurements.
The effective bandwidth has been calculated using the formula below:

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

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 64×64, 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

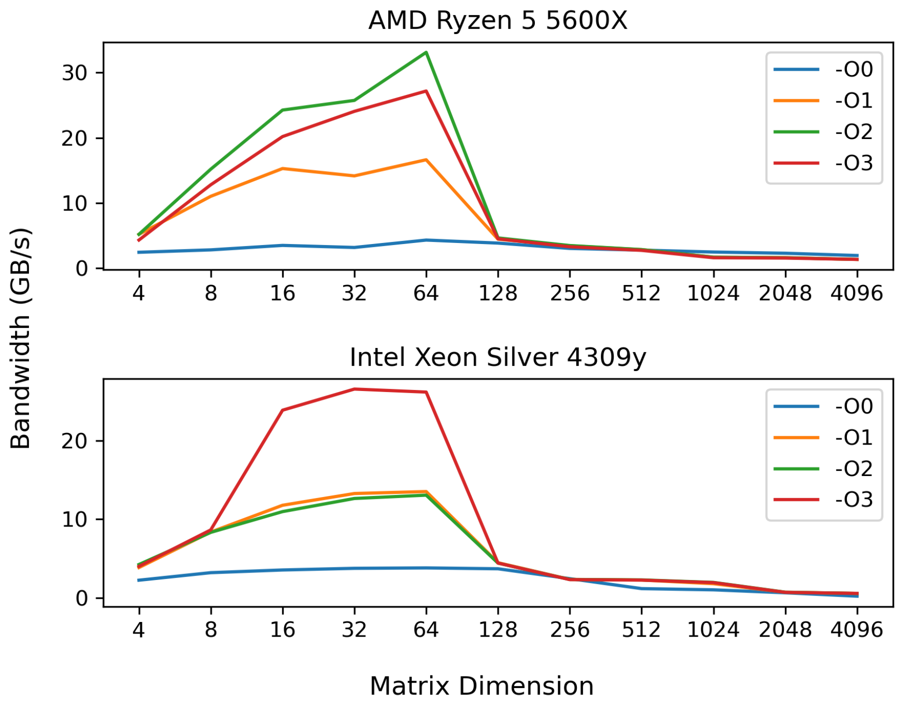

Figure 1: Comparison of the naïve matrix transposition algorithm

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 1024×1024 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 (1024×1024, `-O3`)

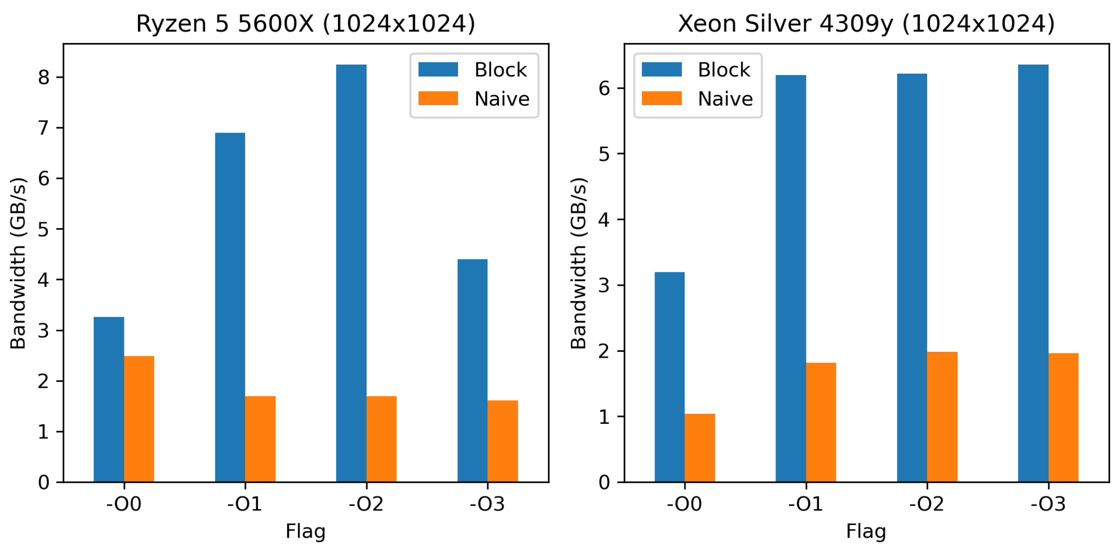

Figure 2: Comparison of the blocked algorithm against the naïve algorithm

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.
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.