Homework 2: Transposition of a Non-Symmetric Matrix

Martina Scheffler, Matr. 249609, martina.scheffler@studenti.unitn.it

June 7, 2024

The code discussed in this paper and instructions to run it can be found on GitHub.

Abstract - CPU

The following work shows two different implementations of an algorithm to transpose a square, non-symmetric matrix whose dimensions are a power of two ($2^N \times 2^N$). The first implements the transposition naively by traversing the matrix and switching the row and column indices during assignment to a new matrix. The second algorithm is a cache-oblivious solution dividing the matrix into blocks of half its size and swapping them until they fit into the cache and can be efficiently transposed. After describing the pseudocode, the algorithms are compiled using the `-Ox` gcc compile flags and their execution times are compared. It can be shown that the block-based implementation outperforms the naive one for all flags except `-O0` which equals no optimizations. Finally, a deeper analysis of the cache misses for both algorithms using a $2^{12} \times 2^{12}$ matrix is provided which explains the superior performance of the block-based transposition.

1 Problem Statement

The goal of this work is to write C++ code implementing the transposition of a non-symmetric $N \times N$ matrix. Taking the transpose is a common mathematical operation which mirrors the values of a matrix along its main diagonal (for further details see e.g. [1]). Further, the program should take as input a parameter determining the size of the matrix, i.e. calling the program like `./transpose 10` should generate a matrix of dimension $2^{10} \times 2^{10}$ and transpose it. This means, that the matrices considered will always be square and of dimensions that are a power of two. Additionally, entries will always be of integer type, as it does not change the analyses.

2 Algorithms

In the following, two different algorithms for transposing a matrix - one a naive direct approach, the other block-based - will be introduced, their implementation discussed briefly and then compared regarding several performance measures.

2.1 Naive Solution

The simplest solution, which also works in case of non-square matrices is to simply loop over the matrix by rows and then by columns and switch the indices when assigning to the transposed matrix, as shown in Algorithm 1.

Algorithm 1 Simple Matrix Transpose ($N \times N$)

```
1: procedure SimpleTranspose(A)
2:     A^T ← Resulting Matrix
3:     for i ← 0, N − 1 do                    ▷ Rows
4:         for j ← 0, N − 1 do                ▷ Columns
5:             A^T[N · i + j] ← A[j · N + i]
6:         end for
7:     end for
8:     return A^T
9: end procedure
```

There also exists a modification of this algorithm if the matrix to transpose is square to perform transposition in-place, saving memory. This version however is known to exhibit poor performance due to the way it accesses the matrix elements and will therefore not be considered further (see [2] for details and pseudocode).

2.2 Block-Based Solution

Matrix transposition can also be done in a cache-oblivious fashion, meaning that the algorithm uses the cache optimally without previously knowing the cache size [3]. For this, the matrix is recursively divided into blocks of quarter its size ($N/2 \times N/2$) which are then transposed and swapped to implement

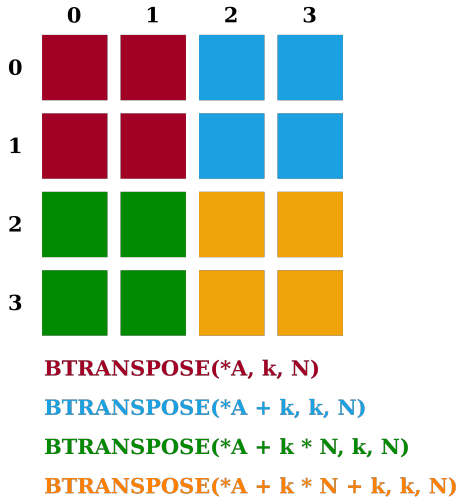
$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^T = \begin{pmatrix} A^T & C^T \\ B^T & D^T \end{pmatrix}. \tag{1}$$

The idea behind this is that a matrix , e.g. from Eq. (1) would be stored as `[A B C D]` in memory and can be efficiently used for calculations once it fits into a cache line all at once. This means that the algorithm becomes more cache-efficient as soon as the current blocks fit into a cache line, however if the cache line size is not known and B chosen too small, further divisions remain efficient as they are done in cache [4]. Since we have specified the matrix size to be a power of 2, the division can be done without problem. Upon reaching a previously defined block size B, the blocks themselves are transposed. After calling the transpose on the blocks, the blocks on the lower left and the upper right of the matrix are swapped as shown in Algorithm 2 [3].

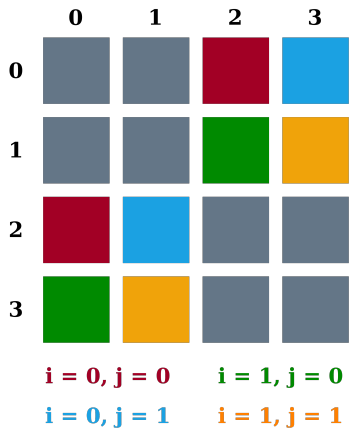
Algorithm 2 Block Matrix Transpose ($N \times N$)

```
1:  procedure BTranspose(*A, n, N)
2:      if n ≤ B then                          ▷ B is block size
3:          TRANSPOSE(*A)
4:      else
5:          k ← N/2                            ▷ Split into N/2 × N/2

                                               ▷ Call BTRANSPOSE recursively
6:          BTRANSPOSE(*A, k, N)
7:          BTRANSPOSE(*A + k, k, N)
8:          BTRANSPOSE(*A + k · N, k, N)
9:          BTRANSPOSE(*A + k · N + k, k, N)

                                               ▷ Swap upper right and lower left blocks
10:         for i ← 0, k − 1 do
11:             for j ← 0, k − 1 do
12:                 swap(A[i][j + k], A[i + k][j])
13:             end for
14:         end for
15:     end if
16:     return A
17: end procedure
```

Two parts of Algorithm 2 are shown in Fig. 1. Lines 5-9 where the method is called recursively are shown in Fig. 1a and lines 10-14 where the lower left and upper right blocks are swapped are shown in Fig. 1b. As the second plot looks a bit counter-intuitive at first, it is important to note that the swapping of the blocks happens after they are transposed, giving the overall correct ordering of the entries.



(a) Recursive Execution of BTRANSPOSE



(b) Swapping of Upper Right and Lower Left Blocks

Figure 1: Visualization of Algorithm 2

3 Experimental Setup

The code will be tested and analyzed on the Marzola computation cluster of the University of Trento. It is equipped with an Intel Xeon Gold 6238R CPU running at 2.2 GHz clock frequency [5]. To show portability and reproducibility, the code is also executed on the author's laptop which has an Intel Core i7-8550U CPU running at 1.8 GHz [6]. The results of these second tests can be found in the additional resources folder of the Github repository.

4 Results

The code will be tested on the cluster described in Section 3. During the tests the execution time of the system with respect to the specified matrix dimensions will be analyzed. Also, the effect of different compile flags will be shown and the cache behavior of the different implementations will be analyzed.

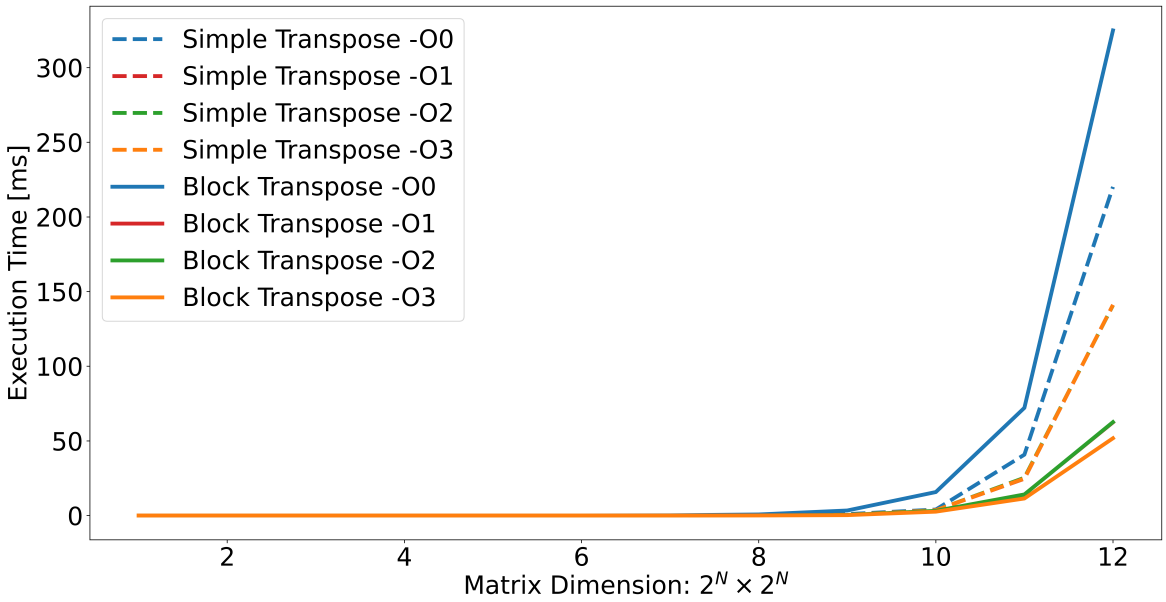


Figure 2: Effect of Different Compile Flags on the Execution Times

4.1 Compile Flags

The following gcc compile flags for optimization will be used (explanations are an excerpt from [7]):

> `-O0`: Default. Reduce compilation time and make debugging produce the expected results.
>
> `-O1`: The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
>
> `-O2`: GCC performs nearly all supported optimizations that do not involve a space-speed trade-off. As compared to '-O1', this option increases both compilation time and the performance of the generated code.
>
> `-O3`: Optimize yet more.

The two algorithms are then executed with matrix dimensions ranging from $2^1 \times 2^1$ to $2^{12} \times 2^{12}$. The block size is chosen to be $B = 32$ for Algorithm 2. The resulting execution times are shown in Fig. 2. It can be seen that roughly up to a dimension of $2^9 \times 2^9$, the execution times remain similar regardless of the implementation and the compile flags. After that, they begin to diverge, with the block matrix transposition outperforming the naive transpose for all compile flags except `-O0`. As for this flag, no optimization is performed, the greater number of operations needed for the second algorithm creates too big a performance overhead, thus increasing execution times. For the other flags (`-O1`, `-O2`, `-O3`), the main difference in execution time is created by the cache-efficiency of the second algorithm as explained in Section 2.2.

4.2 Cache Behavior

To further prove this, the cache behavior is analyzed using Valgrind [8] for a single execution of the transposition algorithms using a $2^{12} \times 2^{12}$ matrix, $B = 32$ and compilation using the `-O3` flag. This size is chosen as Fig. 2 shows a significant divergence in execution times for this size which justifies a closer analysis to find the reason for this behavior.

I1mr	ILmr	D1mr	D1mw	DLmr	DLmw
0.14	0.14	99.9	49.94	99.94	49.96
0.68	0.71	0.06	0.06	0.00	0.00

Table 1: Cache Miss Percentages for a $2^{12} \times 2^{12}$ Matrix. First Row: Algorithm 1, Second Row: Algorithm 2

The resulting cache miss percentages are shown in Table 1. I1mr and ILmr are the instruction fetch misses at the L1 and the last level cache respectively. High miss percentages in these values would indicate that the instructions frequently used during execution do not fit well into the cache, however as the percentages are below one for both algorithms, this is unproblematic. The main difference between the two implementations becomes obvious in the cache misses for the data.

D1mr and D1mw are the read and write misses at the L1 cache whereas DLmr and DLmw are the read and write misses for the last level cache. It can be seen that the second algorithm has a far lower number of L1 cache misses and no last level cache misses at all, while the first algorithm has a read miss rate of almost a 100% and a write miss rate of almost 50% including at the last level cache which means that it has to access the access the much slower main memory, leading to a significant loss of performance.

5 Conclusion and Future Work

To sum up, the paper presents two methods for transposing a square matrix whose dimensions are a power of two, a naive and a cache-oblivious block-based one. As was shown in Section 4, the block-based transition clearly outperforms the naive solution when using at least the `-O1` compile flag due to its more efficent cache behavior. However, the algorithm still has room for improvement as so far it is run sequentially. A version using parallel transposition of the smaller blocks the matrix is decomposed into during transposition could greatly increase performance and should be investigated in the future.

References - CPU

[1] Wikipedia. "Transpose." (2024), [Online]. Available: `https://en.wikipedia.org/wiki/Transpose` (visited on 03/24/2024).

[2] Wikipedia. "In-place matrix transposition." (2024), [Online]. Available: `https://en.wikipedia.org/wiki/In-place_matrix_transposition#Square_matrices` (visited on 04/13/2024).

[3] S. Slotin. "Algorithmica. cache-oblivious algorithms." (2022), [Online]. Available: `https://en.algorithmica.org/hpc/external-memory/oblivious/` (visited on 04/18/2024).

[4] Wikipedia. "Cache-oblivious algorithm." (2024), [Online]. Available: `https://en.wikipedia.org/wiki/Cache-oblivious_algorithm#Examples` (visited on 04/18/2024).

[5] Intel. "Intel xeon gold 6238r processor." (2024), [Online]. Available: `https://www.intel.com/content/www/us/en/products/sku/199345/intel-xeon-gold-6238r-processor-38-5m-cache-2-20-ghz/specifications.html` (visited on 04/18/2024).

[6] Intel. "Intel core i7-8550u processor." (2024), [Online]. Available: `https://www.intel.com/content/www/us/en/products/sku/122589/intel-core-i78550u-processor-8m-cache-up-to-4-00-ghz/specifications.html` (visited on 04/18/2024).

[7] Free Software Foundation Inc. "Gcc-8.5.0 documentation." (2018), [Online]. Available: `https://gcc.gnu.org/onlinedocs/gcc-8.5.0/gcc.pdf` (visited on 04/12/2024).

[8] Valgrind Developers. "Cachegrind." (2023), [Online]. Available: `https://valgrind.org/info/tools.html#cachegrind` (visited on 04/15/2024).

Abstract - GPU

The following part shows three different implementations of matrix transposition using CUDA kernels on an Nvidia GPU. The first uses a naive approach and the global memory, the second and third both use the shared memory to transpose smaller tiles of the matrix. After the pseudocode is shown, the kernels' effective bandwidths are analyzed and their sensitivity to the choice of certain parameters explained. Finally, their execution times are compared to the ones obtained using two different algorithms on the CPU to find out when the usage of the GPU is advantageous and when transposition should be done on the CPU instead.

6 Problem Statement

In the following, the problem from Section 1 will be tackled using a GPU instead of a CPU for computations. For this, three different CUDA kernels will be implemented. Afterwards, their performance will be evaluated by means of the effective bandwidth and compared to the maximum possible bandwidth of the GPU.

7 CUDA Kernels

In the folllowing, three different CUDA kernels will be shown. While the method for transposing the matrix changes, the code calling the kernels remains the same and is shown in Algorithm 3, where `KERNEL` depends on the used method. Furthermore, a warmup kernel (as suggested in [1], but just executing one's own kernel would also suffice) will be used before the timed runs of the kernels to avoid errors in the performance measurements due to startup overhead. Also, each kernel will be run `NUM_REP`, e.g. 10 times, and the overall execution time divided by `NUM_REP` to compensate for small fluctuations. Finally, to guarantee correctness, the resulting transposed matrix is compared with one obtained via the `cuBLAS` library using `cublasSgeam` [2].

Algorithm 3 Host Code

1: N ▷ Matrix dimension along one axis
2: TD ▷ Blocks of TD · BR threads
3: BR ▷ Each transposes a TD · TD tile

4: $nBlocks \leftarrow (\frac{N}{TD}, \frac{N}{TD}, 1)$
5: $nThreads \leftarrow (TD, BR, 1)$
6: KERNEL$\lll nBlocks, nThreads \ggg (A, A^T)$
▷ Original and Transposed Matrix

7.1 Naive Solution

The kernel for the first algorithm can be implemented as shown in Algorithm 4 (taken from [3]).

Algorithm 4 Simple Transpose Kernel

1: **procedure** SIMPLEKERNEL(A, A^T)
2: $x \leftarrow blockIdx.x \cdot TD + threadIdx.x$
3: $y \leftarrow blockIdx.y \cdot TD + threadIdx.y$
4: $w \leftarrow gridDim.x \cdot TD$

5: **for** $i \leftarrow 0; i < TD; i \mathrel{+}= BR$ **do**
6: $A^T[x \cdot w + (y+i)] \leftarrow A[(y+i) \cdot w + x]$
7: **end for**
8: **end procedure**

7.2 Coalesced Transpose via Shared Memory

A more efficient solution is shown in Algorithm 5 (taken again from [3]), where the threads in one block use a common tile in the shared memory. First, each thread copies elements from the original matrix into a column of the tile. After all threads are done copying, they access the tile rowwise but copy into a column in the resulting matrix, thereby transposing the matrix. As also explained in [3], the tile's y-dimension is increased by one to avoid memory bank conflicts. A visualization of this algorithm is shown in [4, p. 10].

Algorithm 5 Coalesced Transpose Kernel

1: **procedure** COALESCEDKERNEL(A)
2: $t[TD][TD+1]$ ▷ shared memory tile
3: $x \leftarrow blockIdx.x \cdot TD + threadIdx.x$
4: $y \leftarrow blockIdx.y \cdot TD + threadIdx.y$
5: $w \leftarrow gridDim.x \cdot TD$

6: **for** $i \leftarrow 0; i < TD; i \mathrel{+}= BR$ **do**
7: $t[tIdx.y+i][tIdx.x] \leftarrow A[(y+i) \cdot w + x]$
8: **end for**
9: $\_\_syncthreads()$ ▷ Wait for end of copying

10: $x \leftarrow blockIdx.y \cdot TD + threadIdx.x$
11: $y \leftarrow blockIdx.x \cdot TD + threadIdx.y$
12: **for** $i \leftarrow 0; i < TD; i \mathrel{+}= BR$ **do**
13: $A^T[(y+i) \cdot w + x] \leftarrow t[tIdx.x][tIdx.y+i]$
14: **end for**
15: **end procedure**

7.3 Coalesced Diagonal Transpose

To further improve Algorithm 5, and avoid the problem of partition camping explained in [4], a small change is made to the kernel. `blockIdx.x` and `blockIdx.y` are replaced with `bIdx_x` and `bIdx_y` as shown in Algorithm 6, graphics and further explanations for this can be found in [4, p. 16ff.].

Algorithm 6 Diagonal Transpose Kernel

```
1: procedure DiagonalKernel(A, A^T)
2:     bIdx_y ← blockIdx.x
3:     bIdx_x ← (bIdx.x + bIdx.y)%gridDim.x
4:     ...                ▷ Continue with Algorithm
5: end procedure
```

8 Experimental Setup

The kernels will be executed on the University of Trento's cluster equipped with a Nvidia A30 GPU [5]. Its maximum bandwidth can be derived as

$$\frac{1215 \cdot \frac{3072}{8} \cdot 2}{10^9} \approx 933\text{GB/s}. \tag{2}$$

To evaluate the performance, the effective bandwidth in GB/s of the implementations can be calculated as

$$EBW = \frac{(B_r[Bytes] + B_w[Bytes])/10^9}{t_{\text{routine}}[s]} \tag{3}$$

where B_r and B_w are the bytes read and written respectively and t_{routine} is the execution time of the procedure in seconds. For our square matrices of type integer, $(B_r + B_w) = (N \cdot N \cdot 4 \cdot 2)$ where $N \cdot N$ is the size of the matrix, the size of an integer is 4 Bytes and the factor 2 includes the read and the write.

For each of the three kernels shown in Section 7, the effective bandwidth will be calculated for matrix sizes from $2^1 \times 2^1$ to $2^{12} \times 2^{12}$. While doing this, the parameters `TILE_DIMENSION (TD)` and `BLOCK_ROWS (BR)` will be varied to find the optimal configuration for each kernel and matrix size. For `TD`, the used values will be $TD = \{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$ for Algorithm 4 and $TD = \{2, \ldots, 64\}$ for Algorithm 5 and Algorithm 6 while minding the constraint $TD \leq N$. The difference for the second and third kernel comes from the fact that the size of the shared memory is limited and would be exceeded for all values of `TD` above 64. For `BR`, the values used will be $BR = \{1, 2, 4, 8, 16, \ldots, TD\}$, so powers of two up until `TD` is reached, all the while minding the constraint $TD \cdot BR \leq 1024$, since the maximum number of threads that can be created in a block is 1024.

9 Results

In the following, a comparison of effective bandwidths between the different kernels, a brief remark on parameter choice and a comparison of execution times between CPU and GPU will be given.

9.1 Effective Bandwidths

For each matrix dimension, the maximum effective bandwidth of all three kernels is shown in Fig. 3. Also, the maximum bandwidth calculated in Eq. (2) is shown in red. It can be seen that the simple kernel performs significantly worse than the coalesced and diagonal ones for matrices of dimensions above $2^8 \times 2^8$, while those two remain very similar to each other. An example using concrete values is given in Table 2.

Kernel	EBW [GB/s]	% of Max. BW
`Simple`	356.76	38 %
`Coalesced`	781.12	84 %
`Diagonal`	780.65	84 %

Table 2: Comparison of Effective Bandwidths for a $2^{12} \times 2^{12}$ Matrix

9.2 Impact of Parameters

All three of the algorithms' performances are dependent on the choice of the parameters `TD` and `BR`. For example, using a $2^{12} \times 2^{12}$ matrix and Algorithm 5, a choice of $TD = 2, BR = 1$ leads to an effective bandwidth of 17.16 GB/s, while $TD = 64, BR = 8$ leads to 781.12 GB/s, an increase of 4452%. This shows just how sensitive the algorithms are to the choice of parameters and how vital testing for the best configuration is.

9.3 Comparison with CPU

To determine when it is useful to offload matrix transposition onto the GPU, the execution times of the algorithms on the CPU and the GPU are plotted in Fig. 4. It can be seen that starting from matrices of dimension $2^9 \times 2^9$, the execution times begin to diverge in favor of the GPU. While one also has to consider the additional transfer time to and from the GPU, it can be definitely said that transposition on the GPU is not worth it for matrices smaller than $2^9 \times 2^9$.

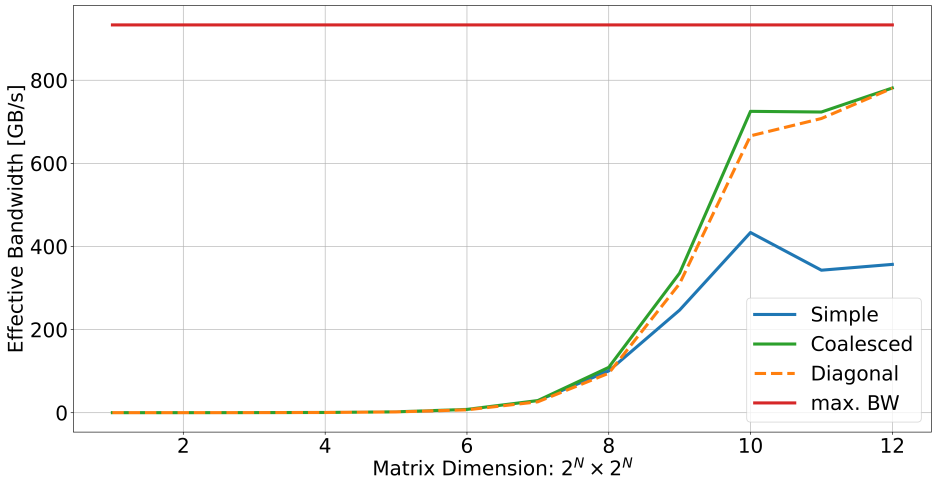


Figure 3: Effective Bandwidths for Different Kernels and Matrix Dimensions

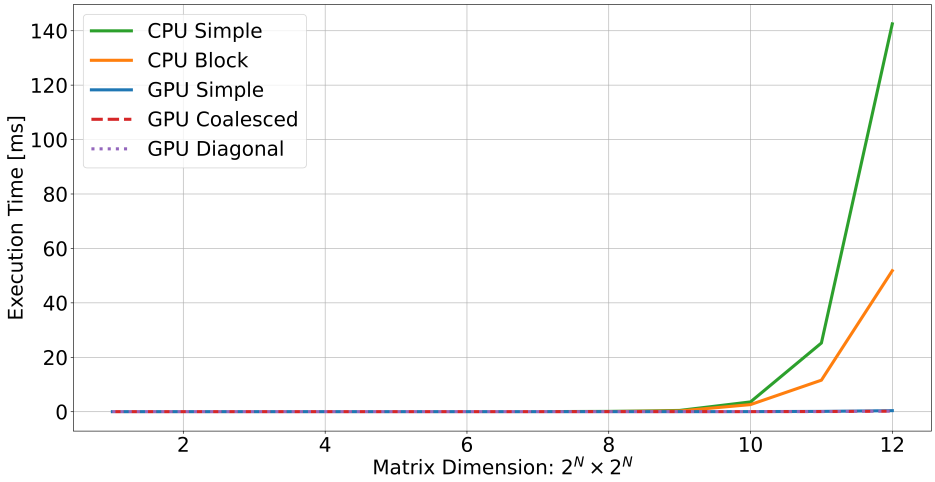


Figure 4: Comparison of Execution Times between the CPU and the GPU

Kernel	Execution Time [ms]
`CPU Simple`	142.5
`CPU Block`	51.8
`GPU Simple`	0.38
`GPU Coalesced`	0.17
`GPU Diagonal`	0.17

Table 3: Comparison of Execution Times between CPU and GPU for a $2^{12} \times 2^{12}$ Matrix

10 Conclusion

To sum up, the paper shows two methods for matrix transposition on the CPU in the first part and three more methods on the GPU in the second. As can be concluded from Section 9.3, offloading the computation onto the GPU is only worth it once the matrix reaches a certain size and further testing for the transfer time to/from the GPU is necessary. However, once transposition on the GPU is done, Algorithm 5 is the best choice as shown in Section 9.1, given of course the right choice of parameters as mentioned in Section 9.2.

References - GPU

[1] T. McKerche et al. "Professional CUDA C Programming." (2014), [Online]. Available: `https://www.cs.utexas.edu/~rossbach/cs380p/papers/cuda-programming.pdf` (visited on 06/05/2024).

[2] Nvidia. "cuBLAS." (2012-2024), [Online]. Available: `https://docs.nvidia.com/cuda/cublas/#cublas-t-geam` (visited on 06/05/2024).

[3] M. Harris. “An Efficient Matrix Transpose in CUDA C/C++.” (2013), [Online]. Available: https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc/ (visited on 05/15/2024).

[4] G. Ruetsch et al. “Optimizing Matrix Transpose in CUDA.” (2010), [Online]. Available: https://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/transpose/doc/MatrixTranspose.pdf (visited on 05/18/2024).

[5] Nvidia. “Nvidia A30 Tensor Core GPU.” (2022), [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/data-center/products/a30-gpu/pdf/a30-datasheet.pdf (visited on 05/15/2024).