Homework 1: Transposition of a Non-Symmetric Matrix

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

April 26, 2024

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

Abstract

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:     $A^T \leftarrow$ Resulting Matrix
3:     for $i \leftarrow 0, N-1$ do                ▷ Rows
4:         for $j \leftarrow 0, N-1$ do            ▷ Columns
5:             $A^T[N \cdot i + j] \leftarrow A[j \cdot N + i]$
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

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

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].

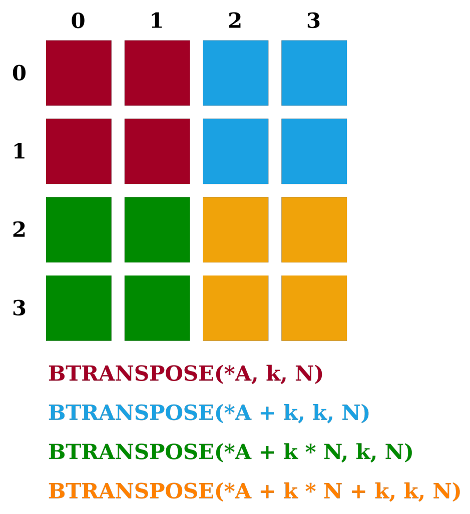
Algorithm 2 Block Matrix Transpose ($N \times N$)

```
1:  procedure BTranspose(*A, n, N)
2:      if $n \leq B$ then                          ▷ B is block size
3:          TRANSPOSE(*A)
4:      else
5:          $k \leftarrow N/2$                       ▷ Split into $N/2 \times N/2$

                                  ▷ 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 $i \leftarrow 0, k-1$ do
11:             for $j \leftarrow 0, k-1$ 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
```

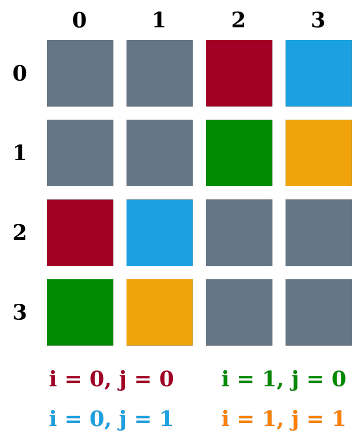
Two parts of Algorithm 2 are shown in Fig. 1. Lines 5-9 where the method is called recursively are shown in Fig. 1a and lines 10-14 where the lower left and upper right blocks are swapped are shown in Fig. 1b. As the second plot looks a bit counter-intuitive at first, it is important to note that the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

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.


0
1
2
3
BTRANSPOSE(*A, k, N)
BTRANSPOSE(*A + k, k, N)
BTRANSPOSE(*A + k * N, k, N)
BTRANSPOSE(*A + k * N + k, k, N)

(a) Recursive Execution of BTRANSPOSE



(b) Swapping of Upper Right and Lower Left Blocks

Figure 1: Visualization of Algorithm 2

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.

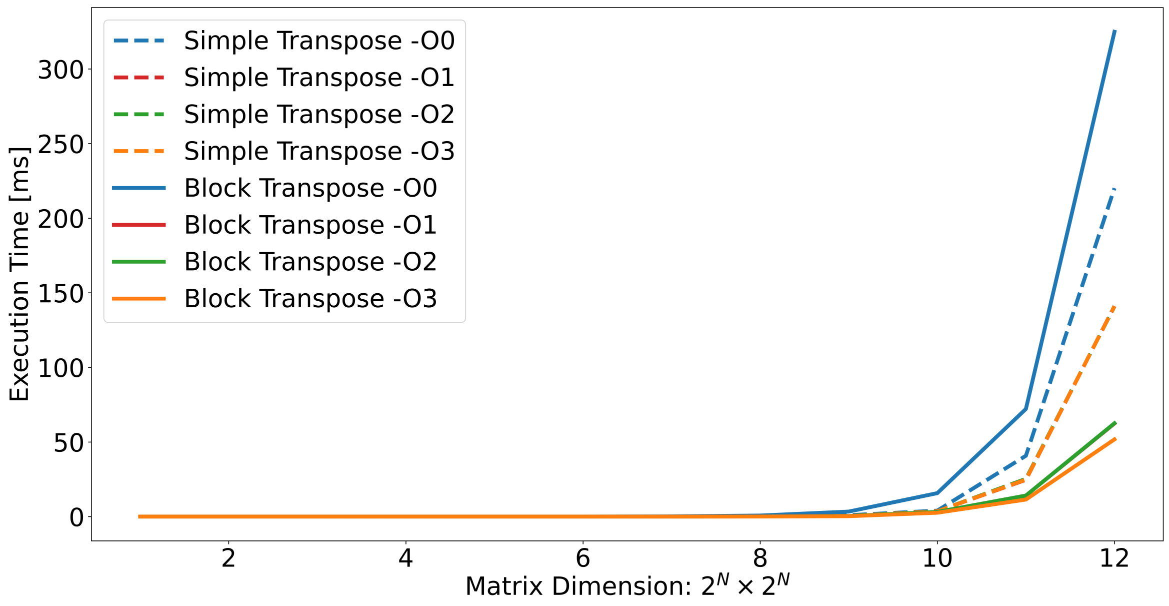


Figure 2: Effect of Different Compile Flags on the Execution Times

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

[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).