Second Assignment - Report

Christian Dalvit
Student-ID: 249988
Trento, Italy
christian.dalvit@studenti.unitn.it

I. Introduction

The objective of this homework is to implement an algorithm for transposing a non-symmetric matrix and to measure and analyze various metrics of the algorithm. This report provides a description of the problem setting, algorithms, and experimental results of the implementation. The code used for this homework is made available through a public Github repository. Details on how to run the code and reproduce the results can be found in the `README.md` file of the Github repository.

II. Problem Description

For a given matrix $A \in \mathbb{R}^{n\times m}$, the transpose of the matrix $A^T \in \mathbb{R}^{m\times n}$ is defined as

$$A^T_{ij} = A_{ji}$$

In this homework, matrices have dimensions of 2^N for $N \in \mathbb{N}$, so only square matrices are considered. As a result, the implemented algorithms don't need to accommodate changes in the output matrix's shape. For the purpose of this homework, we assume a row-major memory layout of the matrix. While implementing an algorithm that computes the transpose of a matrix is straightforward, coming up with an efficient implementation is quite tricky. In general, leveraging spatial and temporal locality can improve efficiency. Because each element of the matrix is accessed only once, temporal locality cannot be exploited for computing the transposed matrix [2], so spatial locality becomes the only source for improvement. The issue with leveraging spatial locality in matrix transposition is that data is accessed along rows but written along columns, potentially leading to poor cache performance. Algorithms that respect spatial locality in their memory access pattern can benefit from quicker access to cached data. In the following, the different algorithms implemented during this homework are described and their memory access pattern is discussed.

A. CPU Algorithms

In this homework, three different algorithms for in-place matrix transposition are implemented. Pseudocode for the algorithms can be found in the appendix. The first implementation, as shown in Algorithm 1, can be directly inferred from the mathematical definition. Transposition is performed by iterating over all entries above the matrix diagonal and swapping them with the corresponding entries below the diagonal. The first implementation will serve as the baseline implementation to measure performance improvements of other implementations. The main issue with the naive implementation's memory access pattern is the disjointed access from `mat[j*size+i]`, potentially causing poor cache performance, especially with larger matrices where `mat[j*size+i]` might not be present in cache and requires loading from memory.

Algorithm 1: Naive implementation

```
input: Size $n$, Matrix $M$
for $i$=0 ... $n$ do
    for $j$=$i$+1 ... $n$ do
        swap($M_{i,j}$, $M_{j,i}$);
    end
end
```

The second implementation (Algorithm 2) tries to improve performance by prefetching the memory addresses needed in the next iteration. This should reduce cache-miss latency by moving data into the cache before it is accessed [6]. The built-in function `__builtin_prefetch` can be used to perform prefetching. `__builtin_prefetch` takes as arguments the address to be prefetched and two optional arguments *rw* and *locality*. Setting *rw* to 1 means preparing the prefetch for write access and setting *locality* to 1 means that the prefetched data has low temporal locality [6]. The second implementation is referred to as "prefetch"-implementation.

Algorithm 2: Prefetch implementation

```
input: Size $n$, Matrix $M$
for $i$=0 ... $n$ do
    for $j$=$i$+1 ... $n$ do
        swap($M_{i,j}$, $M_{j,i}$);
        builtin_prefetch($M_{i+1,j}$, 1, 1);
        builtin_prefetch($M_{j,i+1}$, 0, 1);
    end
end
```

The third algorithm (Algorithm 3) implements a recursive pattern for matrix transposition. It uses the fact that

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

Where A, B, C and D are submatrices. Note that the submatrices B and C get swapped. The idea behind this algorithm

is that it splits the matrix into four sub-matrices until the submatrices fit into the cache. Then the submatrices get transposed and the quadrants get swapped. Algorithm 3 depicts pseudocode for this algorithm. For a matrix size of 128 or smaller the algorithm performs a normal transpose operation (i.e. the `swap_small_matrix`-function call in Algorithm 3). Otherwise, the matrix is split into four submatrices and the function is called recursively (i.e. the **`else`**-block in Algorithm 3). After the transposition of the submatrices, the upper-right and bottom-left quadrants need to be swapped (i.e. the `swap_quadrants`-function call in Algorithm 3).
This algorithm exploits spatial locality as it divides the matrix into sub-matrices that can fit into the cache. The third algorithm also has a reduced I/O complexity of $\mathcal{O}(\frac{N^2}{B})$ [5], where N is the size of the matrix and B the size of the blocks (i.e. the size of matrices where standard transposition is performed). It then transposes each submatrix, which can be performed more efficiently as the whole submatrix is present in cache. This algorithm also works quite well for large matrices, because they are always reduced to submatrices of sizes that fit into cache. The threshold of 128 for performing standard matrix transposition, was selected because the MacBook Air has an L1 data cache of 65KB (as described in Section III). Since each matrix element is an integer having a size of 4 bytes, a submatrix of dimension 128 needs $128 * 128 * 4 = 65536$ bytes. Therefore one submatrix fits in the L1 data cache, which should result in better performance. During the homework, the same algorithm was tested for threshold values 32, 64, 128, 256 and 512. The results indicated that the algorithm performed best with a threshold of 128 on the tested architectures.

Algorithm 3: Recursive implementation

```
input: Size $n$, Row offset $x$, Column offset $y$, Matrix $M$
if $n \leq 128$ then
    swap_small_matrix($n$, $M$, $x$, $y$);
else
    $m \leftarrow \frac{n}{2}$;
    Recursive call of this algorithm;
    transpose_block($m$, $M$, $x$, $y$);
    transpose_block($m$, $M$, $x$, $y + m$);
    transpose_block($m$, $M$, $x + m$, $y$);
    transpose_block($m$, $M$, $x + m$, $y + m$);
    Swap upper-right and bottom-left quadrants;
    swap_quadrants($m$, $M$, $x$, $y$);
end
```

B. GPU Algorithms

TODO!

III. CPU Experiments

A. Setup

The three algorithms were implemented in C. Each algorithm was compiled with optimization levels `-O0`, `-O1`, `-O2`, `-O3`. Each resulting binary was evaluated 50 times for matrix sizes between 2^8 and 2^{14}. Cache data was collected using Valgrind [3]. Each binary was benchmarked with the Cachegrind tool for matrix size 2^{14}. The experiments were conducted on two different architectures

- MacBook Air (2020) having a M1 chip with 3,2 GHz and 8 cores, 8GB of RAM, 131KB L1 instruction cache, 65KB L1 data cache, 4.2 MB L2 cache and a cacheline size of 128 byte.
- iMac (2011) having a Intel Core i5 with 2.5GHz and 4 cores, 8GB of RAM, 64KB L1 cache, 1MB L2 cache, 6MB L3 cache and a cacheline size of 64 byte.

Unfortunately Valgrind is not officially supported for ARM-based Apple computers [4] and Open-Source projects working on compatibility for M1 processors are still in the experimental phase [1]. Therefore it is not possible to provide cache performance data for the MacBook Air experiments.

B. Results

The execution times of the different algorithms are compared in Figure 1. The third algorithm (`oblivious128` in Figure 1) achieved the highest speedup with optimization flags. It showed a speedup of 7.81 on the MacBook Air and 5.7 on the iMac when comparing non-optimized code to code with `-O3` enabled for matrix size 2^{14}. On the MacBook Air, the third algorithm gained an additional speedup of 1.46 for matrix size 2^{14} using `-O2` and `-O3` optimizations compared to `-O1` optimization. The second algorithm (`naive_prefetch` in Figure 1) performs slightly better for large matrices with some optimization enabled, resulting in a speedup of 2.11 on the MacBook Air and 1.73 on the iMac for matrix size 2^{14}. Enabling optimization flags can be disadvantageous for execution time performance (see `naive` and `naive_prefetch` implementations for matrix size 2^{10}).

The effective bandwidth of the different algorithms is compared in Figure 2. It's evident across all graphs that the effective bandwidth decreases as the matrix size increases. This trend arises because the matrix size grows exponentially with a base of 2, while the execution time grows exponentially with a base of roughly 10, as illustrated in Figure 1. Consequently, the execution time increases more rapidly than the matrix size, leading to a decline in effective bandwidth.

Cache data, as depicted in Figure I in the appendix, reveals distinctive behaviors among implementations. Specifically, the `naive` and `naive_prefetch` implementations show similar cache patterns, while the `oblivious128` implementation stands out with different cache behavior. Comparing cache data across implementations, it's evident that the `oblivious128` implementation requires approximately twice as many instructions, data reads, and data writes as the `naive` implementation. The variance could be a result of

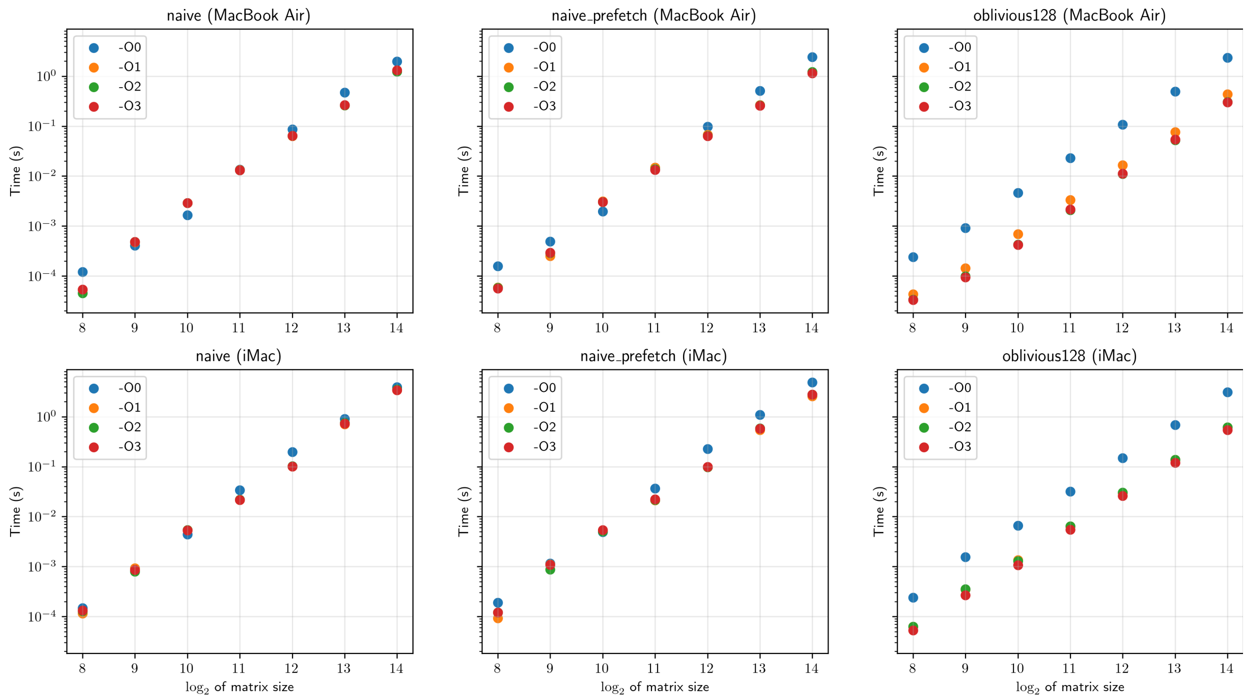

Fig. 1. Comparison of the mean execution time of various algorithms for matrix sizes ranging from 2^8 to 2^{14} for the MacBook Air architecture (upper row) and iMac architecture (bottom row). The y-axis denotes the execution time in seconds and is depicted on a logarithmic scale.

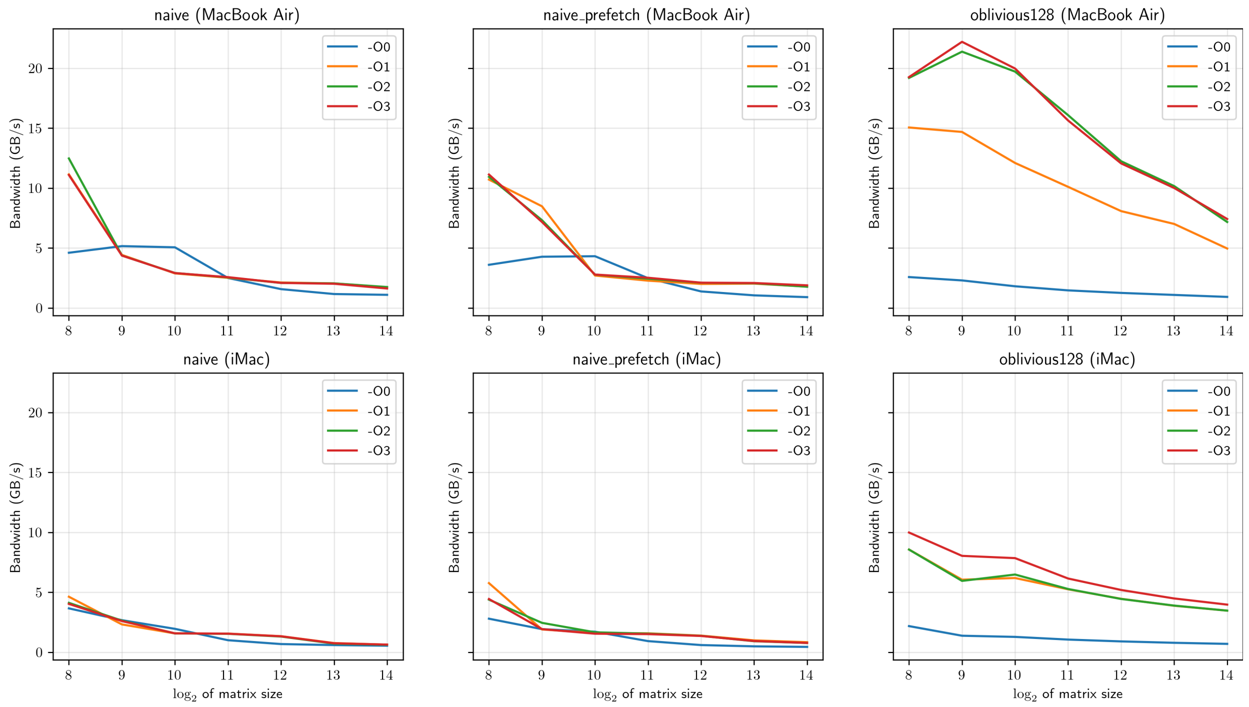

Fig. 2. Comparison of the mean effective bandwidth of different algorithms for matrix sizes between 2^8 and 2^{14} for the MacBook Air architecture (upper row) and iMac architecture (bottom row).

the extra instructions required for transposing submatrices and copying quadrants in the `oblivious128` algorithm, leading to a greater overall instruction count. Another notable finding is that the last-level data cache read misses are significantly lower for the `oblivious128` implementation. This may be due to the improved fit of submatrices utilized in the `oblivious128` algorithm within the cache.

TABLE I
SUMMARIZED CACHE DATA FOR DIFFERENT IMPLEMENTATIONS WITH `-O3` OPTIMIZATION

Name	Ir	Dr	DLmr	Dw	D1mw	DLmw
`naive`	20.6	5.1	0.14	2.4	0.016	0.016
`oblivious128`	41.1	10.1	0.041	4.7	0.033	0.033

[a]Values are given in billions

IV. GPU EXPERIMENTS

TODO!

A. Setup

TODO!

B. Results

TODO!

V. CONCLUSION

After analyzing various algorithms and metrics, it's evident that utilizing blocks can enhance the performance of matrix transposition algorithms. This was demonstrated through the better execution time and effective bandwidth of compiler-optimized versions of the `oblivious128` implementation, which incorporates a form of blocking. The third algorithm also presents promising directions for parallelization, as each submatrix can be processed independently, offering straightforward potential for parallel execution.

REFERENCES

[1] Louis Brunner. *Valgrind for macOS*. 2024. URL: https://github.com/LouisBrunner/valgrind-macos/issues/56#issuecomment-1971933471 (visited on 04/27/2024).

[2] Siddhartha Chatterjee and Sandeep Sen. "Cache-efficient matrix transposition". In: *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No. PR00550)*. IEEE. 2000, pp. 195–205.

[3] Valgrind Developers. *Valgrind*. 2024. URL: https://valgrind.org/ (visited on 04/27/2024).

[4] Valgrind Developers. *Valgrind Documentation*. 2024. URL: https://valgrind.org/docs/manual/dist.news.html (visited on 04/27/2024).

[5] Sergey Slotin. *Algorithmica*. 2022. URL: https://en.algorithmica.org/hpc/external-memory/oblivious/ (visited on 04/27/2024).

[6] GCC Team. *GCC Documentation*. 2023. URL: https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html (visited on 04/27/2024).