GPU Computing

First Assignment - Report

Christian Dalvit

Student-ID: 249988

E-mail: christian.dalvit@studenti.unitn.it

April 27, 2024

1 Introduction

The goal of this homework is to implement an algorithm that transposes a non-symmetric matrix. Furthermore, different metrics of the algorithm should be measured and analyzed. In this report I describe the problem setting, algorithms and experimental results of my 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.

2 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 row-major memory layout of the matrix.

While implementing an algorithm that computes the transpose of a matrix is straightforward, comming 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.

2.1 Algorithms

In this homework three different algorithms for in-place matrix transposition are implemented. The first implementation, as shown in Figure 1, can be directly inferred from the mathematical definition. Transposition is performed by iterating over all entries above the matrix diagonal and swaping them with the corresponding entries below the diagonal. The first implementation is referred to as "naive" and it will serve as 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.

```
void naive_transpose(int size, int* mat){
    for(int i = 0; i < size; i++){
        for(int j = i+1; j < size; j++){
            swap(mat[i*size+j], mat[j*size+i]);
        }
    }
}
```

Figure 1: Pseudocode for the naive implementation.

```
void prefetch_transpose(int size, int* mat){
    for(int i = 0; i < size; i++){
        for(int j = i+1; j < size; j++){
            swap(mat[i*size+j], mat[j*size+i]);
            __builtin_prefetch(&mat[j*size+(i+1)], 1, 1);
            __builtin_prefetch(&mat[(i+1)*size+j], 1, 1);
        }
    }
}
```

Figure 2: Pseudocode for the naive implementation with prefetch.

The second implementation (Figure 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.

The third algorithm (Figure 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. Figure 3 depicts pseudocode for this algorithm. For a matrix sizes of 128 or smaller the algorithm performs a normal transpose operation (i.e. the `swap_small_matrix`-function call in Figure 3). Otherwise the matrix is splitted into four submatrices and the function is called recursively (i.e. the `else`-block in Figure 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 Figure 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 $\mathcal{O}(\frac{N^2}{B})$

```c
void transpose_block(int size, int *mat, int row_offset, int
    col_offset) {
    if (size <= 128) {
        swap_small_matrix(size, mat, row_offset, col_offset);
    } else {
        int m = size / 2;
        transpose_block(m, mat, row_offset, col_offset);
        transpose_block(m, mat, row_offset, col_offset+m);
        transpose_block(m, mat, row_offset+m, col_offset);
        transpose_block(m, mat, row_offset+m, col_offset+m);

        // Swap upper-right and bottom-left quadrants
        swap_quadrants(m, mat, row_offset, col_offset);
    }
}
```

Figure 3: Naive implementation with prefetch of in-place matrix transposition.

[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 determined empircally. During the homework the same algorithm was tested for threshold values 32, 64, 128, 256 and 512. The results showed that 128 was the algorithm worked (on the tested architectures) best with 128 as threshold.

3 Experiments

In this section the experimental results of the three algorithms are evaluated and compared.

3.1 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]. The experiments were conducted on two different architectures

- MacBook Air
- iMac

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.

3.2 Results

All human things are subject to decay. And when fate summons, Monarchs must obey.

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).