GPU Computing

First Assignment

Report

Christian Dalvit

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 [1], 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.

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

Figure 1: Naive implementation of in-place matrix transposition. Here `size` is the dimension of the matrix and `mat` is a pointer to the matrix.

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

Figure 2: Naive implementation of in-place matrix transposition. Here `size` is the dimension of the matrix and `mat` is a pointer to the matrix.

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.

The second implementation tries to improve performance by prefetching the values needed in the next iteration. Since it is known which memory addresses are accessed in the next iteration, these addresses can be prefetched in order to improve performance. For prefetching the

3 Experiments

3.1 Setup

3.2 Results

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

References

[1] 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.