Zoheir El Houari

Matriculation: a12044027

Assignment 1: Cache Optimization

Introduction:

The goal of this assignment is to develop faster algorithms for the matrix transposition problem, and also understand the poor cache behavior of the naïve solution, also for the rest of the assignment, we are going to have the assumption that both of our matrices SRC, DST respectively are stored in the memory in a row-major order.

- Problems of the naïve solution:

  Our matrices size is **n*n = 16384*16384** which means that a whole row of matrix SRC wouldn't fit entirely in memory.

  In the case of matrix SRC, it is stored in row major order and we are traversing it also in a row major order, there for we only get 1 cache miss whenever we fetch a new row and each block is read at most once, this means that at most cases for matrix SRC we would result in ***n/number of blocks*** cache misses.

  On the other hand, the matrix DST is stored in row-major order, but for our transposition problem we need to traverse it in a column-major order. This means that each row of the matrix needs to be fetched many times, this means that we would have at most ***n*n*** cache misses.

  Therefore the naïve solution has a slow running time, it needs to performs O(n*n) I/Os operation.

Cache-Aware Out-of-Place Matrix Transposition:

The main idea behind the Cache-aware algorithm is to work in so called tiles, which are sub matrices of our big matrices SRC and DST.

So the first thing we do is we partition the matrices into tiles, these tiles have a size of (**Blocksize** * **Blocksize**) where block size is going to be chosen so that 1 tiles fit completely in the cache, then what we can do is:

- We can read the elements of the tile, which fits completely in the cache
- Swap the elements of both tiles
- After that we write the elements of both tiles

This means that we are doing a lot of work without the need for additional I/Os

Which is why in my implementation I used 4 nested for loops:

- The first two loops are used to iterate over the tiles which is why the step size of the loops is **Blocksize**.
- The inner two loops are used to perform the transpose copy of a single tile

For the choice of the **Blocksize**, I used the following logic:

> After running the command ***getconf -a | grep CACHE*** I noticed that the alma server has 4 cache lines with each of them having a capacity 64 Bytes which means that the full capacity of the cache lines is 256 Bytes. Therefore, to occupy the cache fully, my block size needed to be of size 8*8
>
> Since our matrices contains only integers we all know that:
>
> > 1 integer = 4 bytes
> >
> > 8 integers = 32 bytes
> >
> > 8 * 8 integers = 256 bytes

This means that one tile would fit completely in the cache. And there for we do a total of

O((n*n) / number of tiles) I/Os operations.

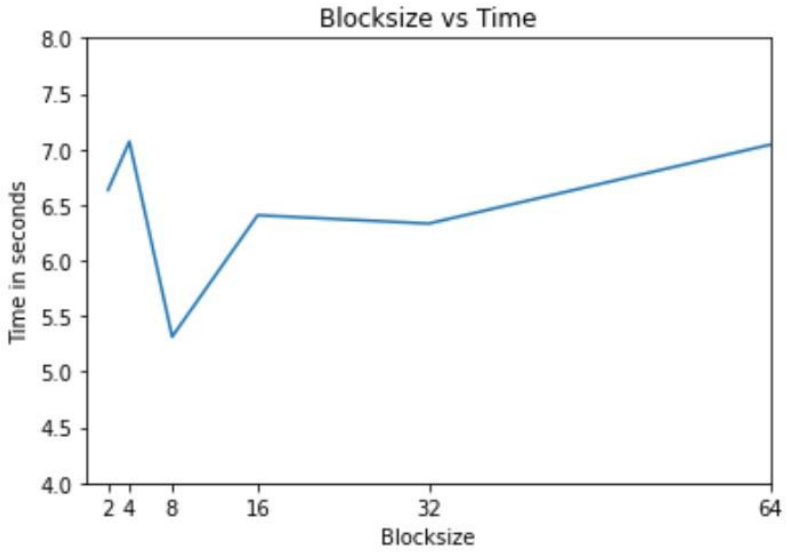


Figure 1

From figure one we can clearly see that the cache aware algorithm performs best when the block size is 8*8 which the case is when the tile fits completely in the cache containing 64 integers (256 Bytes).

The naïve algorithm had a running time of 25 seconds, while the cache aware algorithm resulted in a speedup almost 5x, the best performance was 5.314 seconds.

Cache-Oblivious Out-of-Place Matrix Transposition:

When we use the z curve order to traverse the matrix it helps the cache performance, because it allows the cache lines to represent rectangular tiles, therefore increasing the chance that adjacent accesses are in the cache.

In my implementation I used a function called ***Compact1By1(int z)*** which does de-interleave the bits of z values again in order to get the original coordinates from the Morton code which I got from a resource online(see the link below).

Cache-Optimized Out-of-Place Matrix Transposition:

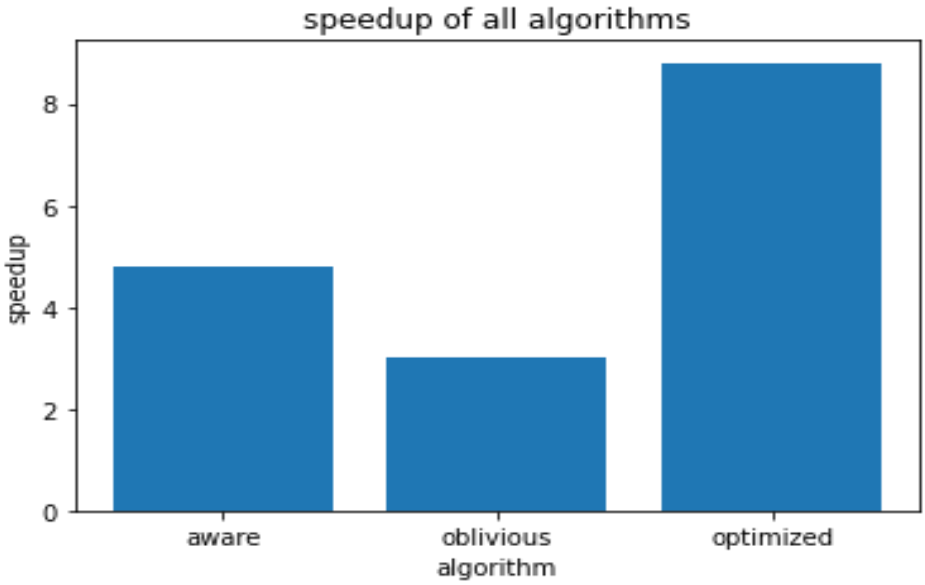
The main idea behind the optimized algorithm solution is to combine both the aware and oblivious solutions. The concept is to work with tiles of size (Blocksize*Blocksize) which we know that it would fit completely in the cache lines of Alma from the cache aware algorithm.

We then traverse the entries of each tile in row-major order while using the z-order curve to define the order by which tile would be visited next.

The cache optimized method resulted in 10x the speedup of the naïve algorithm.

My implementation was done using 3 nested for loops:

- The first loop iterate over the values of z (Morton code) it has a step size of (Blocksize*Blocksize) in order to arrive at the 1st index of the next tile.
- The inner 2 for loops are for traversing each tile in a row-major order and swapping the respective entries.



We can see clearly for the bar chart above, that the desired speedup of 3x was achieved In both the aware and the oblivious algorithms, the aware algorithm had a speedup of almost 5x. The optimized version performed much better than expected, it resulted in a speedup of 8.8x which is higher than the desired speedup of 6x.

References:

- https://en.wikipedia.org/wiki/Z-order_curve
- https://www.youtube.com/watch?v=huz6hJPl_cU&ab_channel=TomNurkkala
- https://fgiesen.wordpress.com/2009/12/13/decoding-morton-codes/