Lab Assignment : Cachelab
Computersystemen

Micha de Groot (10434410)
Eelco van der Wel (10670033)

October 12, 2016

1 Introduction

In this report the various ways to improve the caching hits in transposing several matrices will be discussed. For this lab, three differently sized matrices will be transposed, each with different requirements. The lab can be split in three as follows:

- implement a $32 * 32$ matrix transposition with less than 300 cache misses.
- implement a $64 * 64$ matrix transposition with less than 1,300 cache misses
- implement a $61 * 67$ matrix transposition with less than 2,000 cache misses.

The different successful and unsuccessful approaches for optimal caching will be clarified in the following sections.

2 Progress report

In the attempt to improve and optimize the transpose of the matrices, some overlapping strategies were used. The most general method applied was the blocking method. This method uses the fact that data is cached per block and that the number of cache hits can be improved by reading and writing the whole cache block before moving on to the next part of the matrix.

2.1 Blocking

By transposing the matrix in smaller blocks, the number of cache misses can be reduced significantly. This section gives a small example of a $4 * 4$ matrix with a blocked transposition.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

A is divided into blocks.
$A^1 = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}$ $A^2 = \begin{bmatrix} 3 & 4 \\ 7 & 8 \end{bmatrix}$
$A^3 = \begin{bmatrix} 9 & 10 \\ 13 & 14 \end{bmatrix}$ $A^4 = \begin{bmatrix} 11 & 12 \\ 15 & 16 \end{bmatrix}$

Each block is transposed:
$A^{1T} = \begin{bmatrix} 1 & 5 \\ 2 & 6 \end{bmatrix}$ $A^{3T} = \begin{bmatrix} 9 & 13 \\ 10 & 14 \end{bmatrix}$
$A^{2T} = \begin{bmatrix} 3 & 7 \\ 4 & 8 \end{bmatrix}$ $A^{4T} = \begin{bmatrix} 11 & 15 \\ 12 & 16 \end{bmatrix}$

Finally, the blocks are transposed to the A^T matrix, resulting in a transposed A matrix.
$A = \begin{bmatrix} A^1 & A^2 \\ A^3 & A^4 \end{bmatrix}$ $A^T = \begin{bmatrix} A^1 & A^3 \\ A^2 & A^4 \end{bmatrix}$

$$A^T = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

In this particular example, the size of a block is 2. In the code, this is referred to as `block_size`, and will be different for each matrix size. Note that, for the implementation used for the $32 * 32$ and $64 * 64$ matrices, the block size has to be a factor of the matrix size. For instance, in case of the square 32 matrix, the blocksize can only be 2, 4, 8, or 16. Any other size will result in overflowing rows and columns, and will throw errors. For the unevenly sized matrix, an additional step is done, described in section 2.6

2.2 Diagonal elements

When transposing a matrix, the elements on the diagonal do not move. This property can be used to reduce the number of operations in the inner for loops, effectively reducing the number of cache misses. This step is implemented by checking if the column and row are the same in the most inner for loop. If they are, the element can be transposed in one for loop higher.

2.3 Order of for loops

There is a choice to be made for the order of the for loops. 2 options are possible: for each row, for each column (row major order) and the other way around (column major order). This order is important, because of the way the numbers are stored in the cache. Initially, the experiments are implemented in row major order.

2.4 $32 * 32$ matrix

The first attempted matrix size is 32x32. A naive implementation was given in the exercise, and improved by implementing blocking (section 2.1).

Block size	**# cache misses**
4	487
8	343
16	1183

A block size of 8 almost gives the required 300 or less misses, but not quite. The next step is to implement the diagonal trick. This gives a result on the 32*32 matrix of **287 misses**: enough to pass this step.

As an additional experiment, the order of rows and columns in the for loop are changed, as described in 2.3. Instead of row major order, column major order is used. However, this resulted in a higher number of cache misses: 347.

2.5 $64 * 64$ matrix

Next, the $64 * 64$ matrix is transposed. For this matrix a similar approach was used as with the $32 * 32$ matrix since it is also a square matrix which lengths are a power of two. The same set of blocks was tested with different results.

Block size	**# cache misses**
4	1747
8	4643
16	4687

This clearly shows a drastic deterioration of the caching accuracy of almost a factor of four. Another interesting observation is that not 8 but 4 the more optimal block size is. This gives **1747** misses, which is still worse than required.

The same swap of rows and columns as in 2.4 was tested. This behaved in a similar way and increased the number of cache misses to **1867**

Subsequently the thought was entertained to first divide the $64 * 64$ matrix into four $32*32$ matrices. These matrices would then be transposed according to the method in 2.4, for which it is already known it can be done efficiently. The four transposed matrices would then be returned into the total $64 * 64$ matrix.

Several sizes for the inner block and the outer block were tested. The inner block size variation gave the same variation as without an outer block and the outer block size variation made no difference.

2.6 $61 * 67$ matrix

Finally, the $61 * 67$ matrix is transposed. In order to make the uneven sized matrix work with the blocking method, an additional check has to be added to the inner loop. Aside from checking if the number of elements in each row/column is inside the block, a check has to be performed to see if an element is still within the matrix. This will result in a higher number of cache misses, but is necessary to make the blocking mechanism work for an uneven sized matrix.

As with the previous matrices, different block sizes were tried on the $61 * 67$ matrix. The diagonal trick is included in the measurements.

Block size	**# cache misses**
4	2307
8	2241
16	1812

With a block size of 16, the required less then 2000 misses is reached.

3 Discussion

In this report several successful and less successful ways to improve the caching of a matrix transpose were discussed. For various matrices different results for similar methods were observed. As discussed the blocking method was most widely applied. The most notable observation here was that a different block size was optimal for different matrices, even when said matrix was an exact multiple of another matrix. This could be explained by the fact that the corners furthest away from the diagonal gave different cache misses depending on the block size.

What gave a steady improvement in each case was assigning the diagonal of the matrix outside of the inner for loop, decreasing the number of assignments in the inner loop.

Using a double block structure gave no improvement in the way it was implemented. However, this could lead to an improved caching rate when applied in a different manner. If the first ans last quadrant were transposed in this way, the caching rate of these two parts of the transpose function is expected to be quite good. A different efficient method has then to be found for the remaining two quadrants.

Concluding it can be said that the caching transposing a matrix can be improved, but the improvements will mostly have to be customized for each matrix.