Matrix transpose and fast transpose

Nicole Orzan

June 4, 2018

In this exercise we had been given a code (in fortran and c versions) that performs the transpose of a matrix; the aim was to measure its performance and then to modify it in order to write an optimized version of the code using the Fast Transpose algorithm explained in class.

The Fast Transpose Algorithm

The idea of the Fast transpose algorithm is to divide the original matrix in blocks of a given size, to transpose them and finally to find the exact position inside the transpose matrix where to put them. Here below you can find the piece of code containing the Fast Transpose algorithm I wrote in C language (the one wrote in fortran was similar).
Here A is the original matrix, AT the transpose one and Atemp is the block to be filled, transpose, and placed in the right position inside AT; numblock is the total number of blocks and BLOCKSIZE is the size of a single block; ib and jb are indices that run on the two block's dimensions. This algorithm works only with blocks that are submultiples of the mother matrix.

```
//finding starting point on the rows (ioff) and the columns(joff)
  for(int ib=0; ib<numblock; ib++){
    ioff=(ib)*BLOCKSIZE;
    for(int jb=0; jb<numblock; jb++){
      joff=(jb)*BLOCKSIZE;

//filling the block Atemp
      for(int j=0;j<BLOCKSIZE; j++){
        for(int i=0;i<BLOCKSIZE; i++){
        Atemp[j*BLOCKSIZE+i]=A[(j+joff)*MATRIXDIM+(i+ioff)];
        }
      }

//transposing the block
      for(int j=0; j<BLOCKSIZE; j++){
        for(int i=1; i<j-1; i++){
        Aswap=Atemp[j*BLOCKSIZE+i];
        Atemp[j*BLOCKSIZE+i]=Atemp[i*BLOCKSIZE+j];
        Atemp[i*BLOCKSIZE+j]=Aswap;
        }
      }

//putting the block in the right position inside AT
      for(int j=0; j<BLOCKSIZE; j++){
        for(int i=0; i<BLOCKSIZE; i++){
          AT[(i+ioff)*MATRIXDIM+(j+joff)]=Atemp[j*BLOCKSIZE+i];
        }
      }
    }
```

The following analysis will concern the C code, and was computed on my laptop, a lenovo ideapad 300 with Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz,4 processors, whose architecture is the following:

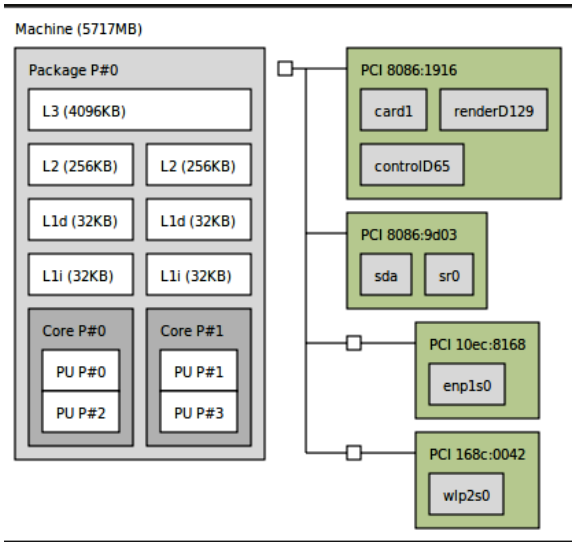

Figure 1: lstopo output of my laptop.

Code Analysis

First of all I ran the original code for matrices of dimensions 1024, 2048, 4096 and 8192. To built a complete study I analyzed the execution times performing 12 runs for each of the four matrices dimensions, then I computed the average times and the standard deviations of the samples. The results that I obtained are written in the following table.

Matrix Dimension	Mean Time	err (σ)
1024	0.02244	0.00055
2048	0.10865	0.00042
4096	0.42602	0.00208
8192	1.49798	0.01429

Then I analyzed the Fast transpose code, fixing the same four matrices sizes and varying the size of the blocks (Nb) used by the algorithm; I used power of two as blocks dimensions for each matrix size: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024.
You can find an image reporting the means and the errors of the execution times obtained from the Fast transpose algorithm in figure 2.

We immediately notice that the fast transpose algorithm is faster then the original code only for a certain "band" of block sized, namely from 8 up. Indeed we can see that for smaller block sizes the use of the algorithm increases the time needed to transpose the matrix; this maybe happens because the algorithm has many extra lines of code then the original code, and since with small values of Nb it has to be performed a lot of times, using it is not more convenient.
We can see then that the curves reach a minimum for the block size=128; this happens because increasing the size of the blocks the algorithm has to be performed lesser times, and became convenient.
Why, then, the execution time increases after this value? The reason is that the block is not longer small enough to be contained inside the caches; indeed as you can see in figure 1 the size of the cache L2 in my laptop is 256 kB, and the small block of size Nb=128 can be contained inside it (8 byte * $(256)^2 \simeq 131$ kB) while a block of size Nb=256 can not. This leads to an increasing of the execution times, because of the larger request of data to the main memory.

To understand better what is happening I decided then to use the same algorithm on the same matrices but with the addition of compiler optimizations O0, O1, O2 and O3, to see what happened.

You can see an image of the curves obtained with the compiler optimization in figure 3 (I

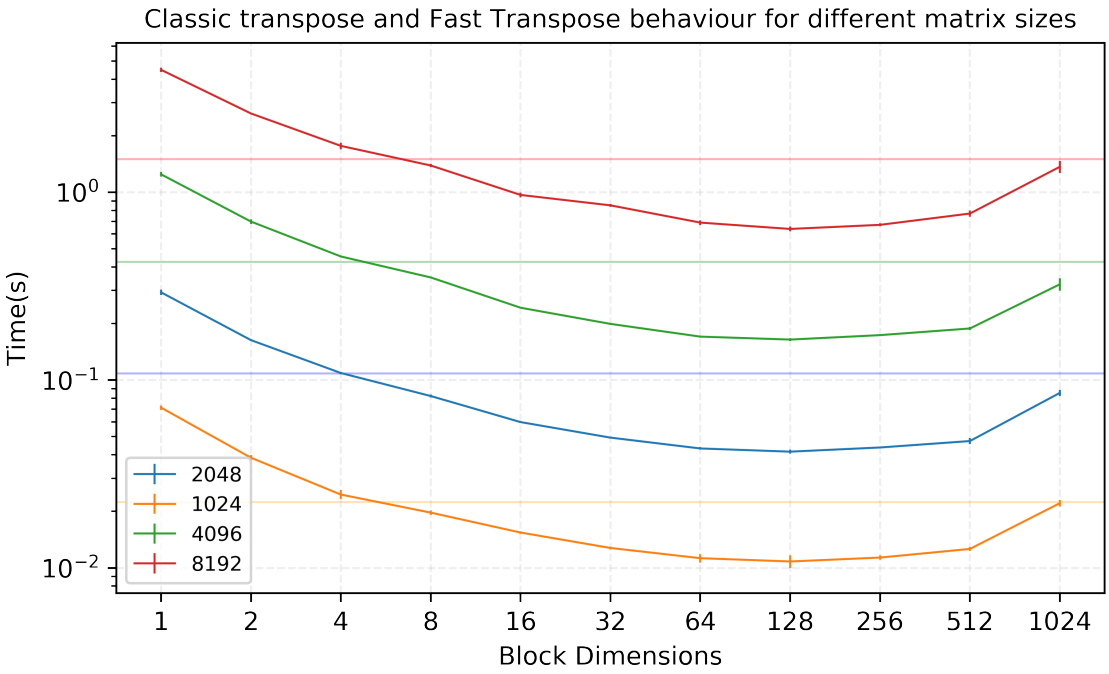

Figure 2: Mean computing times of original code (straight line) and the fast transpose one, for matrices of dimension 1024, 2048, 4096 and 8192 and blocks of sizes 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024; with error bars.

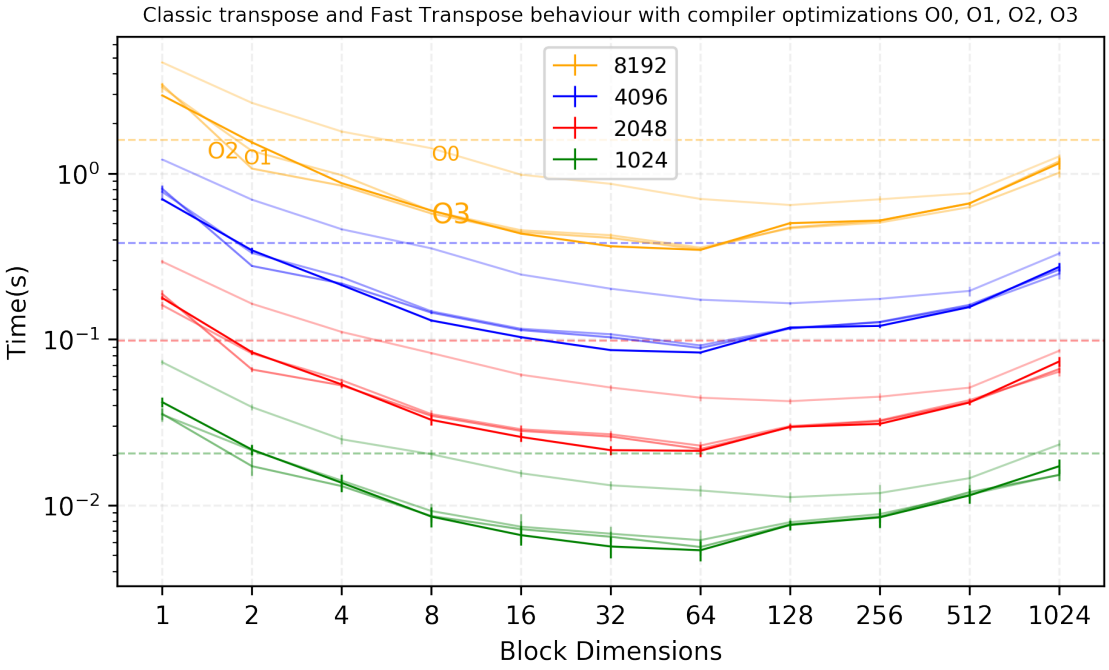

Figure 3: Mean computing times of original code and fast transpose one with the addition of compiler optimizations O0, O1, O2 and O3 (the latter is the most colorful one), for matrices of sizes 1024, 2048, 4096 and 8192, with error bars.

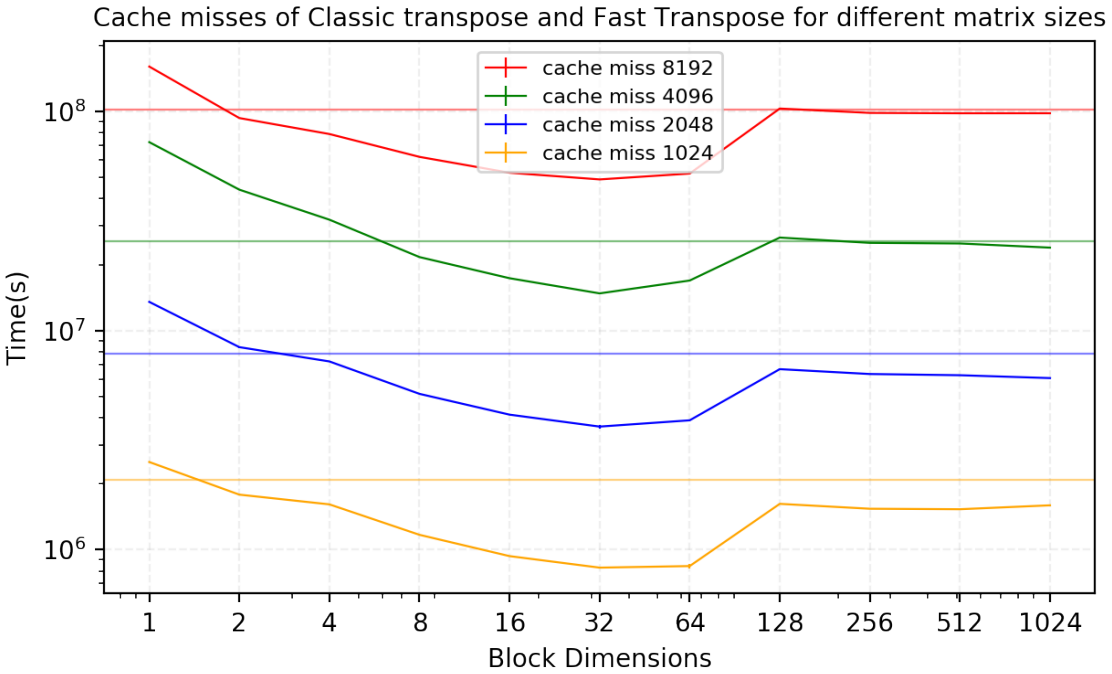

Figure 4: Cache misses of the original code and the fast transpose code, for matrices of dimension 1024, 2048, 4096 and 8192 and blocks of sizes1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024; with error bars.

know that this image is a mess but I didn't know how to put it in an better way... I hope it is understandable!). Here the straight dashed lines represent the execution time of the original code, while the curves with a slight color represent the optimizations O0, O1 and O2. The colorful curve represents the optimization O3. As we expected going form O0 to O3 the execution times curve goes little down.
It seems that the minimum of the curves moves, as we turn on compiler optimizations, form the value of blocks of size 128 to blocks of size 64. Maybe what is happening is that the compiler makes more effective the use of the caches.

Cache misses

In a second moment I measured the cache misses of the code execution using the command:
perf stat -r L1-dcache-load-misses ./myprog.x
First of all I computed cache misses of the original transpose code on my laptop performing 12 runs for each size. You can read the results in the table below.

Matrix Dimension	Cache Misses	err (σ)
1024	2.0883e6	1.6783e4
2048	7.8483e6	1.3766e4
4096	2.5516e7	6.9674e3
8192	1.0178e8	5.1678e4

Then I took the values of the cache misses from the Fast Transpose code, for the same matrices and block sizes as before. In figure 4 you can observe the mean cache misses obtained from the algorithm for each block size.
We can notice from the image that the curves show a minimum around the value of block size=32 for each case.

This value so, unlike what I expected, is different from the one of the execution timings. This can be connected in some way to the size of the cache line of my laptop (which is 64 bytes) and

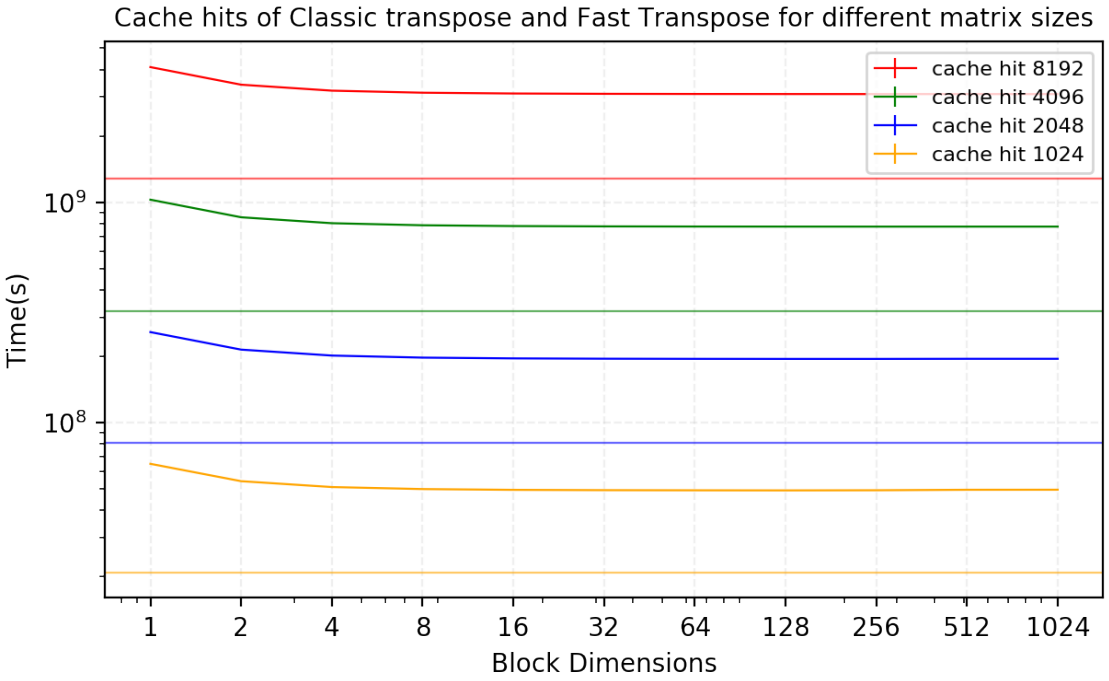

Figure 5: Cache hits of the original code and the fast transpose code, for matrices of dimension 1024, 2048, 4096 and 8192 and blocks of sizes1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024; with error bars.

to the fact that the compiler, by default, is using some optimization to make the execution faster, even if it leads to some cache misses (for example, it can be that it is loading something needed right after the operation that is performing in that moment).