Matrix transpose and fast transpose

Nicole Orzan

December 19, 2017

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.

Code Analysis

Original Code

I ran the original code for matrices of dimensions 1024, 2048 and 4096. To do a complete study, I analyzed the execution times performing 12 runs for each of the three matrices dimensions, and 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	0.0224	5.5e-4
2048	0.1086	4.2e-4
4096	0.4260	2.08e-3

Fast Transpose

The idea of the Fast transpose algorithm is to divide the original matrix in blocks of a given size, to transpose them and then to find the exact position inside the transpose matrix where to put them. In the appendix A you can find a piece of code containing the Fast Transpose algorithm I wrote in C language; the one wrote in fortran was similar.
The following analysis will concern only 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.

I decided then to analyze the written code fixing the same three matrices sizes, named 1024, 2048 and 4096, and varying for those ones the size of the blocks used by the algorithm: I used 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024 as blocks dimensions for each matrix size.
I took then 12 runs for every combinations of these, and I computed mean execution times and standard deviations, which you can find in the table in figure 2. You can find also three figures reporting the mean execution time obtained from the Fast transpose algorithm for each block size, in figures 1, 2 and 3.
We can see from the three figures that the fastest case are the ones with block sizes equal to 64 and 128.

Cache misses

In a second part we had to measure the cache misses, using the command:
perf stat -r L1-dcache-load-misses ./myprog.x
To compute cache misses of the provided code, as before I ran the code 12 times and I get the mean and the standard deviation, which you can read in the table below.
You can find also three figures reporting the mean cache misses obtained from the Fast transpose algorithm for each block size, in figures 4, 5 and 6.
We can see from the three figures that, in each of the three matrix sizes, for the block size=32 we have the lowest cache miss.

	1024 means	err (σ)	2048 means	err (σ)	4096 means	err (σ)
1	0.0713	5.8e-4	0.293	2.24e-3	1.2406	0.0108
2	0.0386	3.6e-4	0.163	4.8e-4	0.6982	6.27e-3
4	0.0246	4.0e-4	0.109	4.3e-4	0.4552	8.5e-4
8	0.0196	1.4e-4	0.082	4.4e-4	0.3521	1.12e-3
16	0.0154	7e-5	0.059	2.2e-4	0.2428	5.8e-4
32	0.0127	7e-5	0.049	2.2e-4	0.1992	8.1e-4
64	0.0112	1.7e-4	0.043	2.0e-4	0.1704	6.8e-4
128	0.0108	2.5e-4	0.041	3.2e-4	0.1643	8.5e-4
254	0.0113	9e-5	0.043	7e-5	0.1735	5.9e-4
512	0.0126	9e-5	0.047	5.3e-4	0.1883	11.7e-3
1024	0.0221	2.6e-4	0.085	1.0e-3	0.3235	7.50e-3

Table 1: Mean computing times and standard deviations of the fast transpose code for matrices of dimensions 1024, 2048 and 4096, using blocks of dimensions 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.

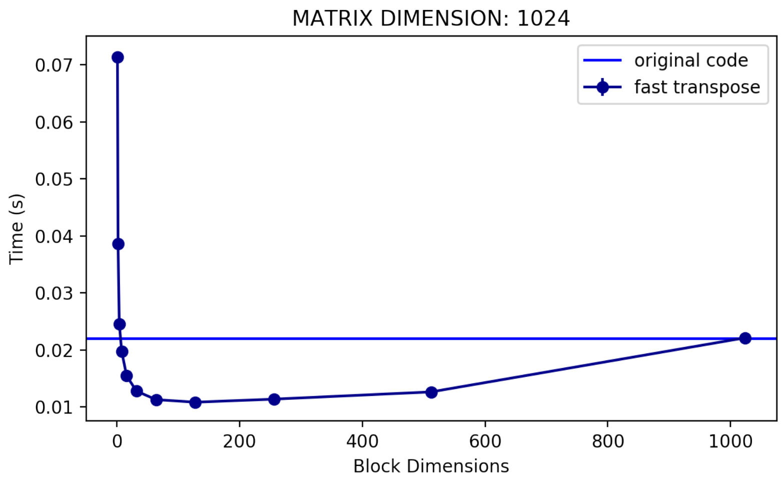

Figure 1: Mean computing times of original code (straight line) and the fast transpose one for a matrix of dimension 1024x1024, with blocks of dimensions 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, with error bars.

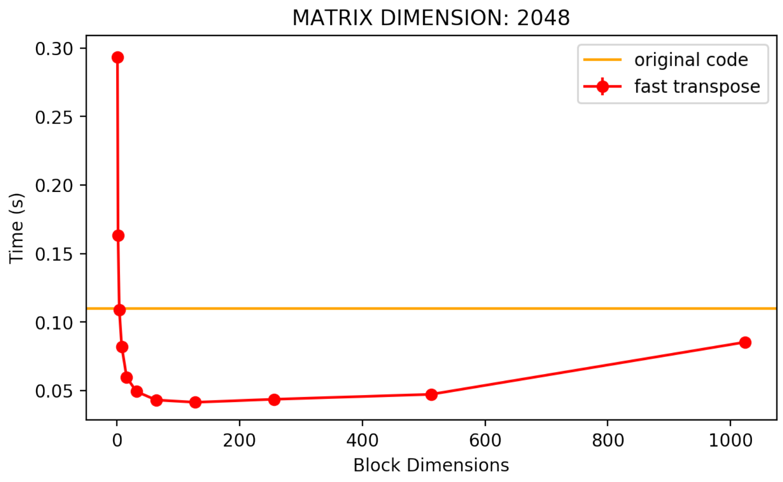

Figure 2: Mean computing times of original code (straight line) and the fast transpose one for a matrix of dimension 2048x2048, with blocks of dimensions 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, with error bars.

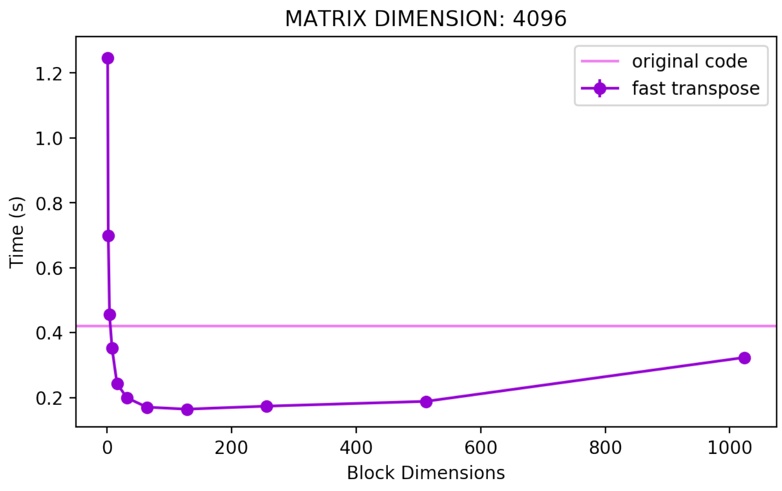

Figure 3: Mean computing times of original code (straight line) and the fast transpose one for a matrix of dimension 4096x4096, with blocks of dimensions 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, with error bars.

	1024 cachemiss	err (σ)	2048 cachemiss	err (σ)	4096 cachemiss	err (σ)
1	2.5105e7	1.4643e3	1.3511e7	5.8267e3	7.2437e7	1.6804e4
2	1.7814e6	4.8196e3	8.3964e6	2.1525e4	4.3958e7	5.1516e4
4	1.608e6	5.8872e2	7.2248e6	7.1822e3	3.2062e7	1.7950e4
8	1.1168e5	5.5493e2	5.1388e6	5.4577e3	2.1642e7	2.1076e4
16	9.3346e5	6.7739e2	4.1298e6	4.6835e3	1.7327e7	1.1090e4
32	8.2702e6	1.6526e3	3.6427e6	2.0598e4	1.4775e7	1.0060e7
64	8.4079e5	7.7386e4	3.8911e6	8.9111e3	1.6913e7	4.0284e4
128	1.6170e6	8.2492e2	6.6622e6	1.1303e4	2.6568e7	2.0775e4
254	1.5359e6	6.5381e2	6.3300e6	5.6089e3	2.5128e7	1.5068e4
512	1.5281e6	7.6503e2	6.2503e6	6.2640e3	2/4965e7	2.1047e4
1024	1.5922e6	2.5391e3	6.0651e6	4.9287e3	2.3902e7	1.9256e4

Table 2: Mean computing times and standard deviations of the fast transpose code for matrices of dimensions 1024, 2048 and 4096, using blocks of dimensions 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.

Matrix Dimension	Cache Misses	err (σ)
1024	2.0883e6	7.5e3
2048	7.8483e6	4.58e4
4096	2.5516e7	5.00e3

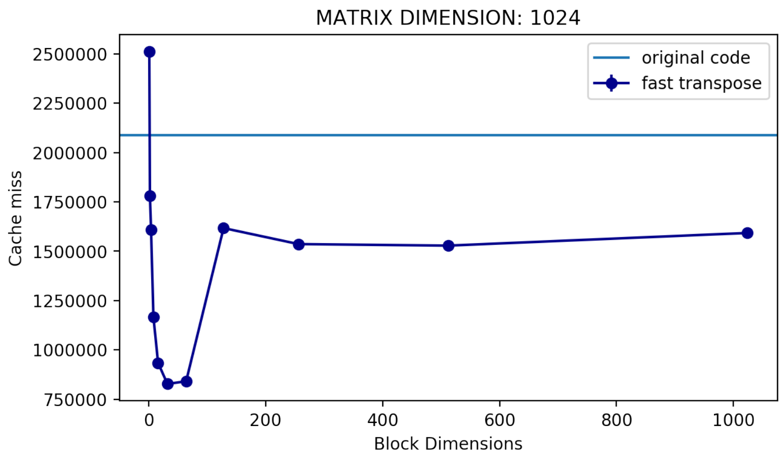

Figure 4: Cache misses of the original code and the fast transpose code, for a matrix of dimension 1024x1024 and blocks of dimensions 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024; with error bars.

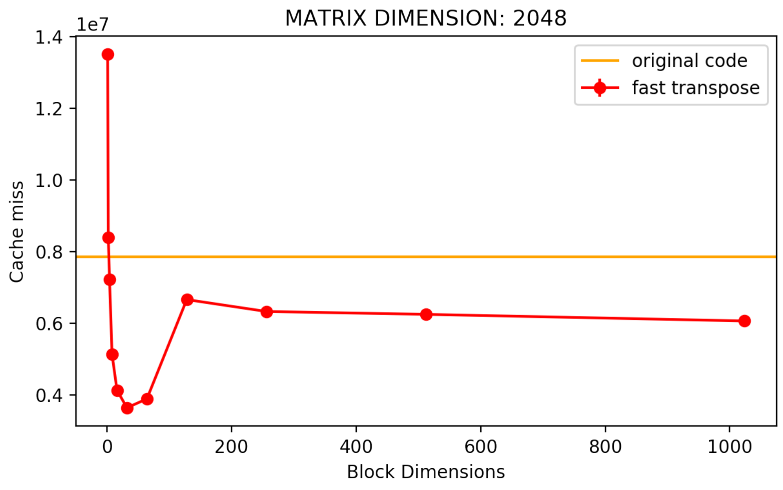

Figure 5: Cache misses of the original code and the fast transpose code, for a matrix of dimension 2048x2048 and blocks of dimensions 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024; with error bars.

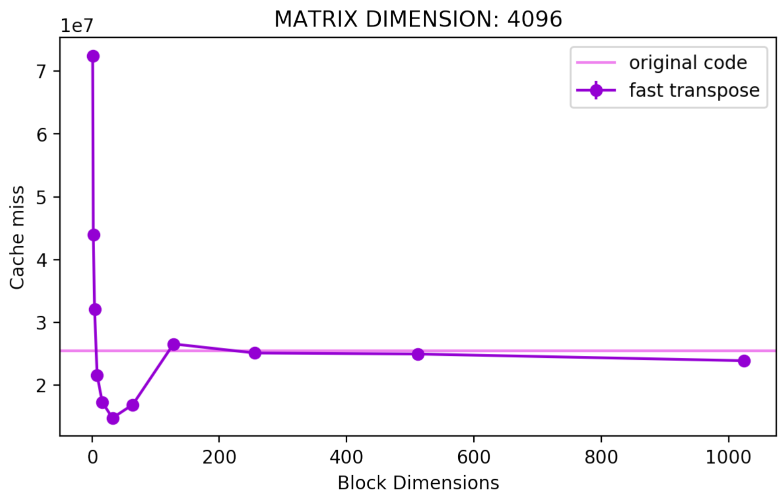

Figure 6: Cache misses of the original code and the fast transpose code, for a matrix of dimension 4096x4096 and blocks of dimensions 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024; with error bars.

Appendice

A Fast Transpose Algorithm

This piece of code is the implementation I made of the fast transpose algorithm in C language. 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.

```
for(int ib=0; ib<numblock; ib++){
  ioff=(ib)*BLOCKSIZE;
  for(int jb=0; jb<numblock; jb++){
    joff=(jb)*BLOCKSIZE;
    for(int j=0;j<BLOCKSIZE; j++){
      for(int i=0;i<BLOCKSIZE; i++){
      Atemp[j*BLOCKSIZE+i]=A[(j+joff)*MATRIXDIM+(i+ioff)];
      }
    }

    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;
      }
    }

    for(int j=0; j<BLOCKSIZE; j++){
      for(int i=0; i<BLOCKSIZE; i++){
        AT[(i+ioff)*MATRIXDIM+(j+joff)]=Atemp[j*BLOCKSIZE+i];
      }
    }
  }
```