Dssc esercizio 4

Alberto Presta

January 2019

Introduction

The aim of the exercise is to use optimization techniques in order to improve a code used to perform a matrix transpose. Initially we will limit ourselves to evaluate the performance of a naive code that has been given to us; then we will try to improve it, dividing the initial matrix in blocks. We will use *perf* in order to evaluate the performance in terms of cache misses.

Naive code

The Naive code written by professor Girotto accepts the dimension of the matrix in input, then It fills the matrix in this way:

```
for( i = 0; i < MATRIXDIM * MATRIXDIM; i++ ){
    A[i] = (double) i;
  }
```

Then It calculates the transpose of the matrix in that way.[1]

```
for( i = 0; i < MATRIXDIM; i++ ){
    for( j = 0; j < MATRIXDIM; j++ ){
      AT[ ( j * MATRIXDIM ) + i ] = A[ ( i * MATRIXDIM ) + j ];
    }
  }
```

Also the code calculates the elapsed time necessary to perform the transpose. We have tried with different size and we reached the following results:

SIZE OF THE MATRIX(SQUARE MATRIX)	ELAPSED TIME (secs)
1000	0.0101
2400	0.1971
4096	2.26261
8192	10.0103
16000	24.2793
24000	38.1821

[1]It is a naive way, in the sense that it explicitly calculates the transposed, without any optimization technique.

We notice that the elapsed time increases more than linearly when the dimension of the matrix becomes greater. Now we will evaluate the performance of this code,especially in terms of cache misses: in fact we are forced to write elements in the transpose matrix in "columns wise" and it is no such a good thing for C. We used perf and this commands:

```
perf stat -e cpu-cycles:u,instructions:u,cache-misses:u,branches:u,
    branch-misses:u  ./trasposta 8192
```

...And we have the following results

```
  30039075602 cpu-cycles:u              #    0,000 GHz
        1946389470 instructions:u             #    0,06   insns per
    cycle 75243389 cache-misses:u
```

Now we want to analize the number of cache misses and cache hit: through the command *perf stat -e L1-dcache-load-misses ./trasposta 8192*. We have the following results:[2]

```
 1304901849 L1-dcache-loads
 103262903 L1-dcache-load-misses

```

Fast code

What we want to do is to optimize our code; but how? There are a lot of different techniques! The First (and the unique thing that we are going to do in this report) thing to do is to divide the computation of the transposed matrix in blocks in order to take advantage of locality and cache blocking. We have introduced a new input number in the code, which is the dimension of the blocks in which we want to divide the matrix. Then we have organized the computation in four nested loops:

1. The first two are used to define the different blocks.
2. The third one define the "traveler" index.
3. In the last one we do the transposition of the single block.

[2]We have calculated also the total number of L1-dcache-loads.

Here we have the code of this 4 nested loops:

```c
for(i=0;i<MATRIXDIM/BLOCKSIZE;i++){
   blocco_i=BLOCKSIZE*i; //inizio del blocco viaggiando verticalmente
   blocco_if=blocco_i + BLOCKSIZE; //fine del blocco viaggiando verticalmente
    for(j=0;j<MATRIXDIM/BLOCKSIZE;j++){
       blocco_j=BLOCKSIZE*j; //inizio del blocco viaggiando orizzontalmente
       blocco_jf=blocco_j + BLOCKSIZE; //fine del blocco viaggiando orizzontalmente
       for(int  k=blocco_i;k<blocco_if;k++){
            viaggiatore=MATRIXDIM*k; //viaggiator lungo i blocchi
            for(int h=blocco_j;h<blocco_jf;h++){
               AT[(viaggiatore+h)]=A[(MATRIXDIM*h+k)];
                                 }
                     }
          }
 }
```

First of all, we want to take a look of elapsed time: in this case we will change both the dimension of the matrix and the size of the blocks:

MATRIXDIM	BLOCKSIZE	ELAPSED TIME [secs]
4096	16	0.1818
4096	32	0.1687
4096	124	0.1625
4096	256	0.1800
4096	512	0.2680
4096	513	0.3265
4096	2048	0.2895
8192	16	0.7518
8192	64	0.6587
8192	124	0.7722
8192	256	1.7649
8192	512	1.8164
8192	4096	1.8229

As we can see, there is an INCREDIBLE IMPROVEMENT in terms of elapsed time: in fact, in the case of a matrix of size equal to 8192, time has decreased more than an order of magnitude if we consider blocks of size equal to 64! as we did in the previous section we want to control the performance in terms of cache misses and cache hits; we will consider these data:

```
MATRIXDIM = 8192
BLOCKSIZE = 124
```

We have the following results:

```
 2493366216 cpu-cycles:u
1951058883 instructions:u
10648586 cache-misses:u
```

We notice that the number of cache misses is decreased a lot (from 75243389 to 10648586): in particular we want to focus on the number of "L1d-cache-load-misses" and we have:

```
 102013408 L1-dcache-load-misses
 1218672317 L1-dcache-loads
```

also in this case the number of L1-dcache-load-misses has decreased compared to the naive code.

What is the best dimension of the blocks?

We have seen that it is not true that if we increase (or decrease) the size of the blocks the elapsed time increases (or decreases); we should find a sort of trade-off between the dimension of the matrix and the dimension of the blocks. In the graph below we will show how elapsed time varies changing the dimension of blocks with a 8192x8192 matrix.

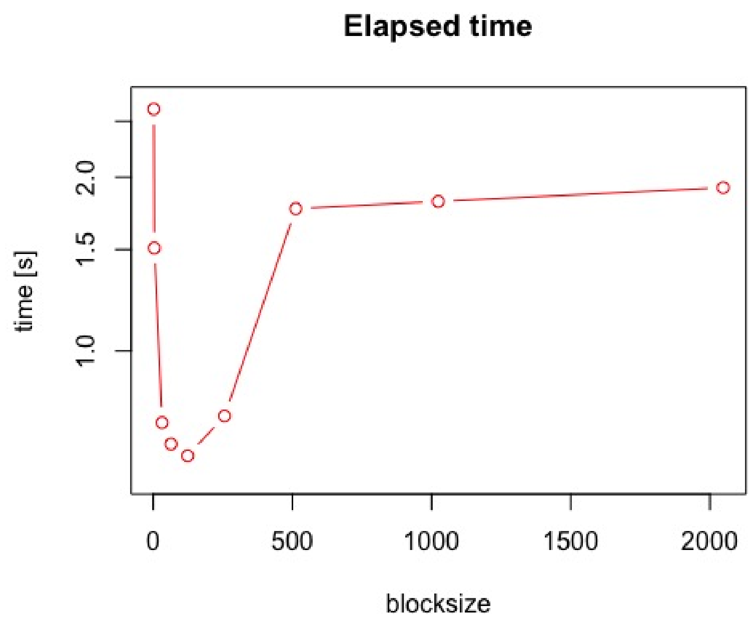

Figure 1: In this figure we can see how elapsed time change with the dimension of the blocks; the dimension of the blocks that we have take into account are the following: 2,4,16,32,64,124,256,512,1024.

We can observe that in this case we reached the best results for BLOCK-SIZE=64!
We can conclude that thanks to divide the matrix in blocks, we can improve our algorithm in terms of time and memory devices (we could see that, thanks to the fast code, we have saved a lot of memory).