

First Assignment GPU Computing - Report

Michele Minniti

Student id: 247168

michele.minniti@studenti.unitn.it

https://github.com/Sminnitex/Homework1_GPUComputing

1. Section 1: problem description

Our task is to perform a matrix transpose in C, basically the operation consists in invert rows and columns of our source Matrix. To formalize a little bit this concept, we can refer as the transpose of a matrix A as A^T ; considering m the dimension of the rows of the matrix A , and n the columns (for brevity I will use the notation $A[m][n]$ to refer to a generic matrix of dimensions m and n), the transpose operation will give us a new matrix $A^T[n][m]$.

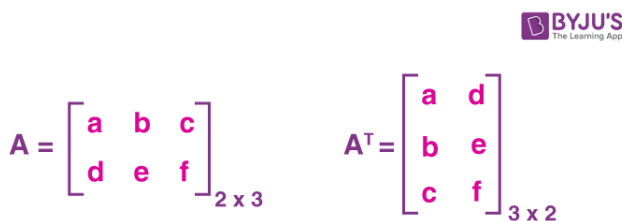

$$A = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}_{2 \times 3} \quad A^T = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}_{3 \times 2}$$

Figure 1. Simple example of a matrix transpose, taken from [1]

The goal is to develop an algorithm capable of doing this operation and analyze the performance considering different levels of optimization.

The code works taking as input the dimensions of our matrices, and giving as output the time of the operation. To visualize better the process we will increase gradually the dimension of our matrices and see how this changes the performance on the long run, on a basis of about 500 iterations. We will also compare the results of two different techniques to perform this task: the normal matrix transpose and the block matrix transpose. The experiments will be performed on the same hardware but on different conditions of our OS to see how this changes the performance, and then we will analyze the bandwidth graph. First of all, let's briefly see the pseudo-code of the normal matrix transpose and let's talk about it.

```
Function matrixTranspose(sourceMatrix, rows, columns):  
    transposeMatrix = newMatrix(columns, rows);  
    from i = 0 to rows :  
        from j = 0 to columns :  
            transposeMatrix[j][i] =  
                sourceMatrix[i][j];  
        endfor  
    endfor  
    return transposeMatrix;  
Algorithm 1: Matrix transpose algorithm
```

This is the base algorithm where we will perform the transpose simply performing over two for loops a change of coordinates, this will give us the possibility of performing the task easily even in the case of a non symmetric matrix. We will compare this operation to the Block Matrix Transpose algorithm,

which will be performed as it follows:

```
Function blockMatrixTranspose(sourceMatrix, rows, columns, blockSize):  
    transposeMatrix = newMatrix(columns, rows);  
    from ii = 0 to rows, ii ← ii + blockSize :  
        from jj = 0 to columns, jj ← jj + blockSize :  
            from i = ii to findMin(rows, ii + blockSize) :  
                from j = jj to findMin(columns, jj + blockSize) :  
                    transposeMatrix[j][i] =  
                        sourceMatrix[i][j];  
                endfor  
            endfor  
        endfor  
    endfor  
    return transposeMatrix;
```

Algorithm 2: Block Matrix transpose algorithm

This time to the function we will need to pass a new parameter, block size, and the operation will be divided in small blocks over the whole size of the matrix using 4 nested loops. To perform a fair comparison we will initialize our matrices with random float values and compare the time only of the transpose operation. At the end of each time check we will increase the rows and columns dimension of 1 unit for all the 500 iteration of our for loop.

2. Section 2: experimental results

The code will be performed on a specific hardware, my laptop which is an Acer Aspire E5 produced in 2017.

The laptop is equipped with an Intel Core i5-7200U CPU. This CPU has 2 cores and 4 threads, 3MB of cache and 34 GB/s of max bandwidth. Then the PC is equipped with 12GB of RAM, and Windows 11 as OS (even if for convenience the code will run on WSL, in order to use the Valgrind software). Valgrind is a tool that we will use to analyze the cache behavior, specifically with the command-tool cache-grind. So let's take a look to the plot of the time to perform the transpose with the normal and block technique, changing the dimension of the matrices:

In the plot is analyzed also the performance using four different optimization levels for GCC compiler that goes from -O0 (No optimization), to -O3 (aggressive optimization). As we can see the results are pretty interesting, in fact not only the normal transpose is often faster than the block transpose, but analyzing the statistics we can see how in both cases the -O3 optimization is not the best overall. In fact in the block transpose the -O1 optimization is more consistent in the results having a lower standard deviation, and a lower minimum peak, however the median and mean of the time used is slightly higher, this can be easily explained noticing how in the -O3

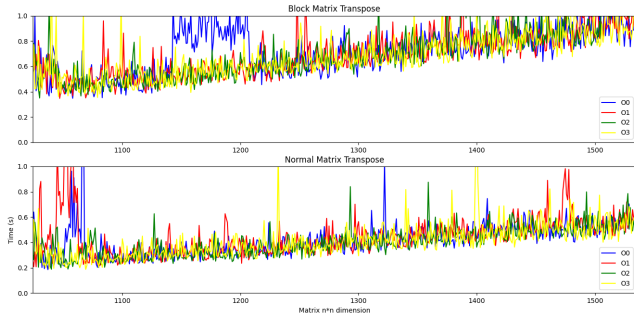


Figure 2. Performance measure of the code executed from my laptop

run there are way more outliers in the results, with spike of time consuming transpose. In the normal transpose the -O2 optimization is easily the fastest, with better statistics in everything except the global minimum value compared to the -O3. Let's try to make a sense of these results taking a look to the cachegrind file, on figure 3.

Function-file summary

Dr	Dwr	Dlwr	Dlwr	Dr	function:file	Dlwr	Dlwr
27,070,892,000 (27.08, 27.08)	2 (0.1%, 0.1%)	2 (0.1%, 0.1%)	6,795,122,688 (19.08, 19.08)	2,848 (0.0%, 0.0%)	2,844 (0.0%, 0.0%)		
2,545,171,000 (26.08, 26.08)	0 (0.0%, 0.0%)	0 (0.0%, 0.0%)	0 (0.0%, 0.0%)	random:???			
22,090,852,864 (22.5%, 26.1%)	3 (0.2%, 0.3%)	1 (0.2%, 0.3%)	10,135,961,600 (29.08, 49.08)	131,876,724 (08.1%, 36.1%)	53,325,483 (97.1%, 97.1%)		
690,840,512 (0.0%, 35.81)	849,390,336 (12.1%, 31.83)	52,453,984 (0.8%, 48.83)	TransposedMatrix(float*, float*, int, int, int)???				
17,817,197,856 (18.2%, 48.3%)	3 (0.2%, 0.5%)	9 (0.2%, 0.5%)	6,795,122,688 (19.08, 49.08)	512 (0.0%, 36.1%)	511 (0.0%, 97.1%)		
1,698,780,672 (17.3%, 53.7%)	0 (0.0%, 93.8%)	0 (0.0%, 93.8%)	random:???				
15,296,275,772 (15.6%, 83.9%)	9 (0.5%, 1.0%)	9 (0.5%, 1.0%)	5,949,823,756 (17.4%, 86.8%)	84,342 (0.1%, 36.1%)	43,384 (0.1%, 97.1%)		
1,099,444,497 (17.3%, 71.6%)	53,243,705 (5.9%, 99.7%)	52,762,429 (48.8%, 97.7%)	main:???				
8,439,983,360 (8.7%, 92.6%)	0 (0.0%, 1.0%)	0 (0.0%, 1.0%)	2,545,171,000 (7.4%, 34.1%)	512 (0.0%, 36.1%)	4 (0.0%, 97.1%)		
1,698,780,672 (17.3%, 53.7%)	0 (0.0%, 99.7%)	0 (0.0%, 99.7%)	random(1)???				
4,246,951,680 (4.2%, 56.9%)	1 (0.1%, 1.1%)	1 (0.1%, 1.1%)	849,390,336 (2.1%, 36.8%)	0 (0.0%, 36.1%)	0 (0.0%, 97.1%)		
849,390,336 (8.9%, 86.4%)	0 (0.0%, 99.7%)	0 (0.0%, 99.7%)	rand:???				
2,816,611,848 (2.8%, 99.8%)	1,300 (79.1%, 80.4%)	1,293 (79.2%, 80.3%)	1,048,158,127 (3.1%, 99.8%)	2,247,652 (1.1%, 99.7%)	1,377,341 (2.1%, 99.7%)		
126,756,608 (1.1%, 99.8%)	2,816,611 (0.1%, 100.0%)	2,243,759 (2.1%, 99.8%)	???:???				

Figure 3. File summary of a cachegrind file read with cg annotate, is about block transpose with flag

-O0

We can read the file on figure 3 as it follows: we basically have three different instructions, and these are Instruction executed, Data read and Data writes. For each of the instructions we have three measurements: the total number of time an instruction is called, the cache misses on the first level, and the cache misses on the last level. The two percentage in brackets refer to the contribution of a measure referred to the whole program, or for the function. So considering the first percentage in the image: the entire code is 100% of the instructions executed, and our block transpose function represents 26% of the calls of our program. In absolute number it has been read over 30 billion times and has a very low amount of cache misses (about 0.3%) of first and last level. However the number of cache misses for reading and writing operation on Data is relevant, and slow down the whole program.

Now if we try to take a look to a normal transpose cachegrind output we will notice something important.

As we can easily see comparing the two methods it immediately catches the eyes that the block transpose has been called way more times than the normal transpose. This could depend on the complexity of the function, that in the block case uses more parameters and more instruction, needing four nested loop instead of two. But what it matters is the number of cache misses, and we can see how both function struggles a bit, resulting in a good number of misses on L1 and LL. The percentage of misses on L1 cache is always greater than the misses on LL, this should not be surprising since the L1 cache is smaller than the LL; the access on L1 is constant and the

Function-file summary

Dr	Dwr	Dlwr	Dlwr	Dr	function:file	Dlwr	Dlwr
27,070,892,000 (27.08, 27.08)	2 (0.1%, 0.1%)	2 (0.1%, 0.1%)	6,795,122,688 (19.08, 19.08)	2,848 (0.0%, 0.0%)	2,844 (0.0%, 0.0%)		
2,545,171,000 (26.08, 26.08)	0 (0.0%, 0.0%)	0 (0.0%, 0.0%)	0 (0.0%, 0.0%)	random:???			
22,090,852,864 (22.5%, 26.1%)	3 (0.2%, 0.3%)	1 (0.2%, 0.3%)	10,135,961,600 (29.08, 49.08)	131,876,724 (08.1%, 36.1%)	53,325,483 (97.1%, 97.1%)		
690,840,512 (0.0%, 35.81)	849,390,336 (12.1%, 31.83)	52,453,984 (0.8%, 48.83)	TransposedMatrix(float*, float*, int, int, int)???				
17,817,197,856 (18.2%, 48.3%)	3 (0.2%, 0.5%)	9 (0.2%, 0.5%)	6,795,122,688 (19.08, 49.08)	512 (0.0%, 36.1%)	511 (0.0%, 97.1%)		
1,698,780,672 (17.3%, 53.7%)	0 (0.0%, 93.8%)	0 (0.0%, 93.8%)	random:???				
15,296,275,772 (15.6%, 83.9%)	9 (0.5%, 1.0%)	9 (0.5%, 1.0%)	5,949,823,756 (17.4%, 86.8%)	84,342 (0.1%, 36.1%)	43,384 (0.1%, 97.1%)		
1,099,444,497 (17.3%, 71.6%)	53,243,705 (5.9%, 99.7%)	52,762,429 (48.8%, 97.7%)	main:???				
8,439,983,360 (8.7%, 92.6%)	0 (0.0%, 1.0%)	0 (0.0%, 1.0%)	2,545,171,000 (7.4%, 34.1%)	512 (0.0%, 36.1%)	4 (0.0%, 97.1%)		
1,698,780,672 (17.3%, 53.7%)	0 (0.0%, 99.7%)	0 (0.0%, 99.7%)	random(1)???				
4,246,951,680 (4.2%, 56.9%)	1 (0.1%, 1.1%)	1 (0.1%, 1.1%)	849,390,336 (2.1%, 36.8%)	0 (0.0%, 36.1%)	0 (0.0%, 97.1%)		
849,390,336 (8.9%, 86.4%)	0 (0.0%, 99.7%)	0 (0.0%, 99.7%)	rand:???				
2,816,611,848 (2.8%, 99.8%)	1,300 (79.1%, 80.4%)	1,293 (79.2%, 80.3%)	1,048,158,127 (3.1%, 99.8%)	2,247,652 (1.1%, 99.7%)	1,377,341 (2.1%, 99.7%)		
126,756,608 (1.1%, 99.8%)	2,816,611 (0.1%, 100.0%)	2,243,759 (2.1%, 99.8%)	???:???				

Figure 4. Cachegrind output read with cg annotate of a normal matrix transpose with flag -O1

misses very frequent.

The misses are over 90% in the Data read section both for the block and normal transpose, at every optimization flag. This possibly means that my architecture is a bottle neck for the program. However is quite interesting to compare this results with other, obtained by the same hardware but using, instead of the WSL, a Linux system on the bare metal. The performance obtained are consistently better from every point of view, that is related to the better use of memory access: while WSL has to communicate with windows in order to obtain the data from my C program, a Linux OS can manage everything on it's own, and the new graph in this case are the following:

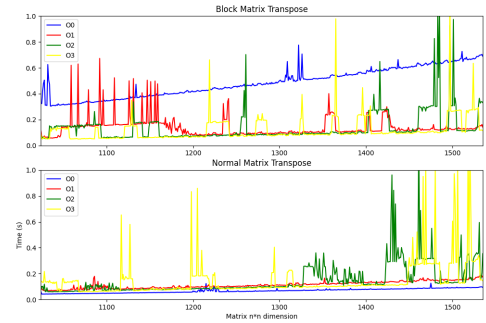


Figure 5. Performance of the same code on a Linux OS

The results are still influenced by many outliers, but this time is more clear what levels of optimization perform better than the other, and this time analyzing the stats we can see how the block matrix transpose works way better with the aggressive optimization, while the normal transpose perform better without optimization. Analyzing these cachegrind files we can see how this changes are reflected with a percentage of cache misses in writing data that drops especially for the block transpose. So the communication between OS was the real bottleneck in the application and the memory is managed way better by the Linux OS.

Now let's try to calculate the bandwidth to have more data, we will perform this analysis on the Linux results since they are more consistent and the communication between OS seems to make the code perform worse. As we know the formula to calculate the bandwidth is:

$$effectiveBandwidth = ((Dr + Dw)/10^9)/t \quad (1)$$

So as we could expect seeing the time metrics the bandwidth confirms that this code is more efficient for the normal matrix transpose, and in particular for the one -O0 optimization flag. The block matrix transpose, despite being less efficient has a good growth in performance for higher optimization flags.

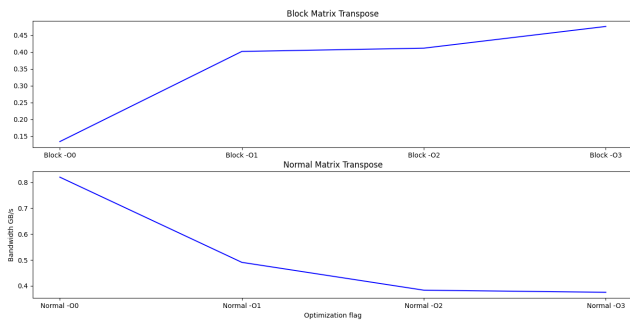


Figure 6. *Bandwidth analysis*

So it's clear that the block algorithm doesn't perform better than the normal algorithm, but as we have analyzed this seems to be related to the amount of cache misses that is in percentage consistently higher than in the normal algorithm. Despite that, not using a GPU to use parallelization our work between blocks for sure has an impact and could improve the performance of our code. The next step could be to adapt this algorithm and test it in a more suited hardware to do this job.

References

- [1] [Online]. Available: <https://byjus.com/maths/transpose-of-a-matrix/>.