Assignment 3: Cache and Performance

Authors:
Kevin de la Coba Malam Group 1391
Miguel Herrera Martínez Pair 2

1. Introduction

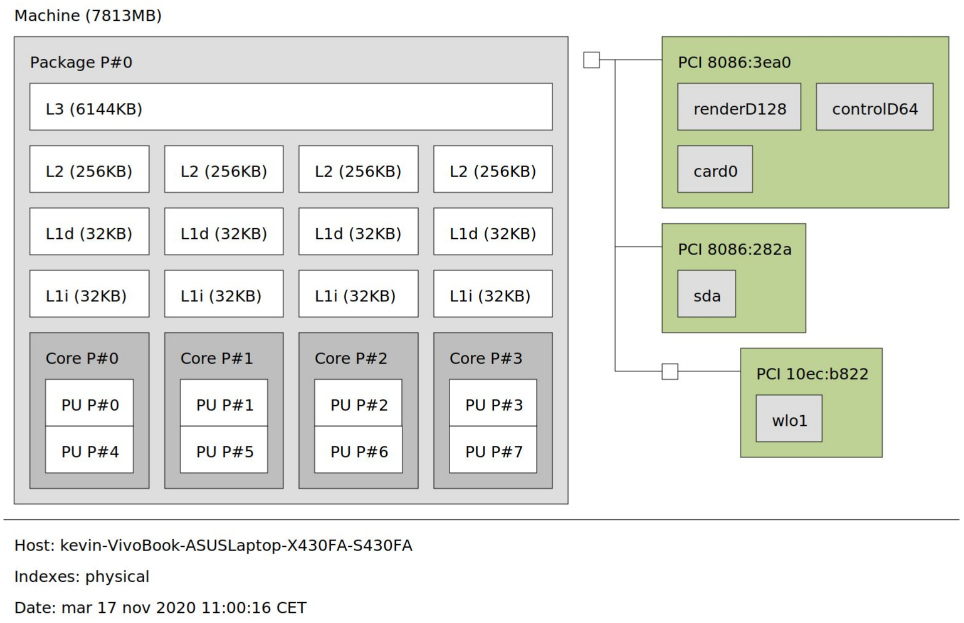
In this assignment we created a script which creates plots and files related with the performance of our CPU doing different tasks. These tasks are: adding all the elements of a matrix, but traversing the matrix by columns (slow) or by rows (fast), and doing matrix multiplications .

In the last part of the assignment we are asked to create one function which performs a multiplication of 2 matrices with the normal method (row by column) and another function to perform a multiplication of 2 matrices, but this time, the second matrix is going to be transposed before any multiplication.

In our case P is 6 because: We are pair 2, so 2mod7 = 2 + 4 = 6. P = 6

2. Exercises

a. System cache.



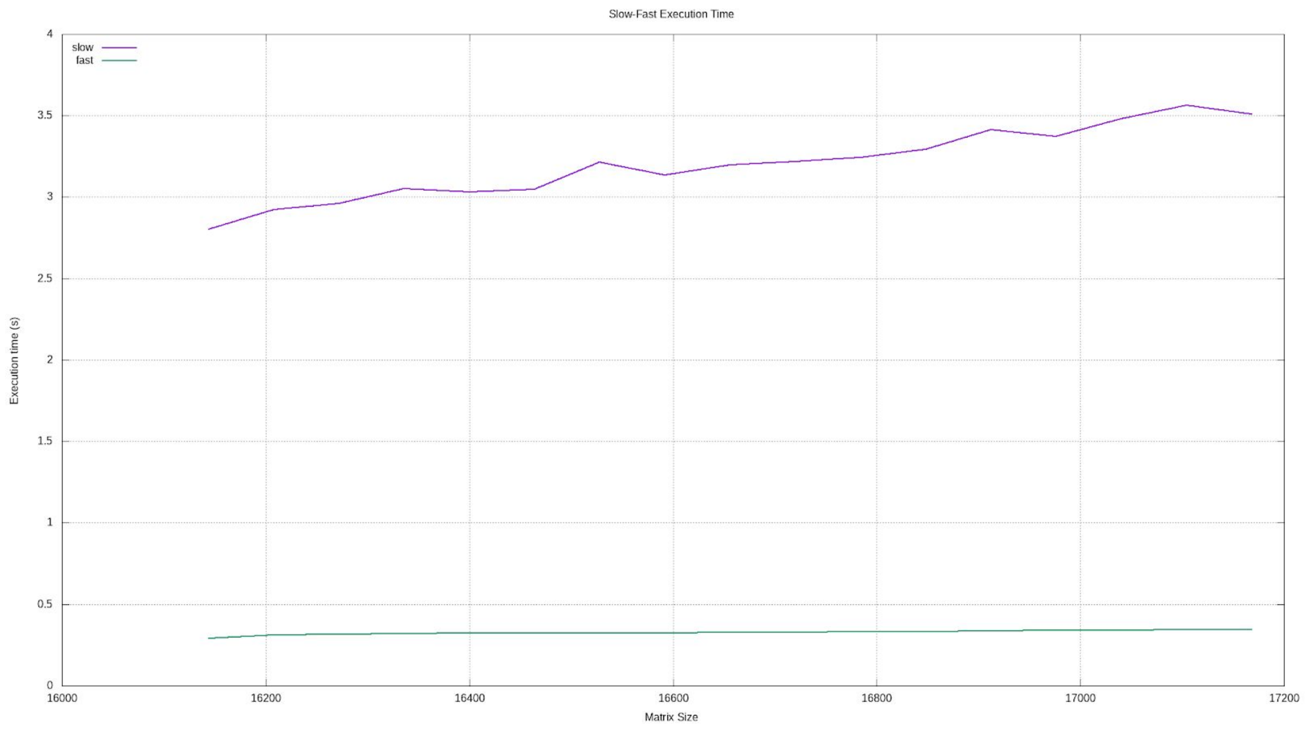
This is a map of my CPU. As we can see it has 4 cores, each of those has one level 1 cache for data and instructions, that cache is a N-Way associative cache, with N being 8, and with 32 KB for each one (data and instruction). Then I also have a level 2 cache of 256 KB and this one is a N-Way associative cache, with N being 4. Finally we have the last level with a cache common for all the cores of the processor, this cache has a size of 6144KB and it is a N-Way associative cache, with N being 12.

b. Cache and Performance

In our case, we obtain the execution time for the programs slow and fast using matrices of size 16144*16144 (10000 + 1024*6) up to 17168*17168 (10000 + 1024*(6+1)) in increments of 64.

Explain the reason why performance measurements must be taken multiple times for each program and matrix size.

We call each program n_times (15) and do the average. This is because if we don't measure performance several times we will not have a good sample. Consequently, the generated plot will have too many sharp edges which don't show us the real performance in one moment. Because if we generate again the plot in that moment we could have a close but different value compared to the previous one. For that reason, we run the program several times and then we calculate the average, as the value obtained actually represents the overall behavior of the program.



Why is the execution time for a small matrix similar, but different when the size is increased? How is the matrix stored in memory, per row (elements of the same row are stored consecutive in memory) or per column (elements of the same column are stored consecutive in memory)?

The matrix is stored in memory like this:

1 2 3 4
5 6 7 8
9 0 1 2

Memory: 1 2 3 4 5 6 7 8 9 0 1 2

So if we try to access the matrix by rows, we will traverse the memory in a linear way. We would have to just access the next element in memory in each iteration, as follows:

1 2 3 4 5 6 7 8 9 0 1 2

But if we try to access by columns we have to calculate everytime the position where the element is in the memory. First we would access 1, then the element 5 which is not next to 1 in memory and so on... The process of taking the right element would take some time.
So when the array is small, in the case where we have to calculate the sum by columns, there's less distance between each element, therefore we wouldn't have to traverse that much in the memory. But when it is bigger the distance between each element is increased, so now we have to traverse more in memory to get the right element.

c. Cache size and Performance

In our case, we obtain the cache misses for the programs slow and fast using matrices of size 5072*5072 (2000 + 512*6) up to 5584*5584 (2000 + 512*(6+1)) in increments of 64.

Are there any trend changes observed by varying the cache sizes for the slow program?

Yes, there are changes by varying the cache. If we have a small cache, it is less probable to find the data we are looking for, but if we have a bigger cache, it is more probable to have stored the data anywhere in the cache. So having a big cache means less misses. That's the general case.

And for the fast program?

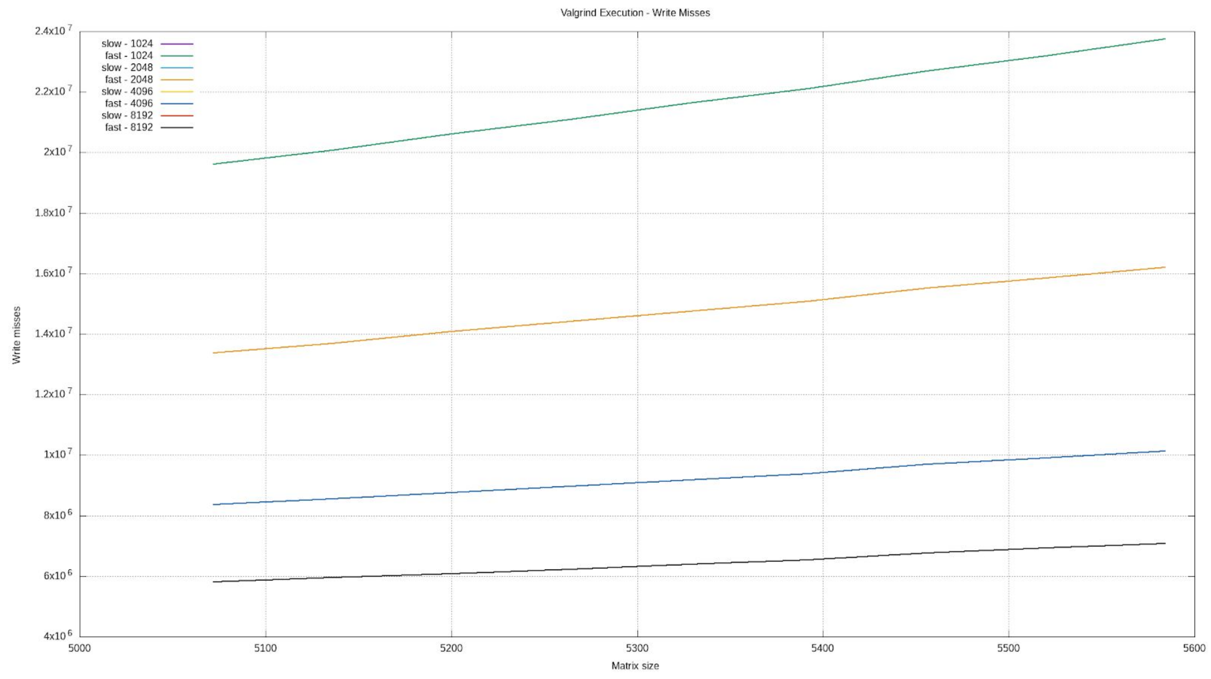
For the fast program, the data is stored and accessed in a sequential way. So when a block of memory is stored in the cache it is more probable that if, for example we have an array with 1--> 2 --> 3, these three numbers are all stored in the same block and therefore there are less misses.

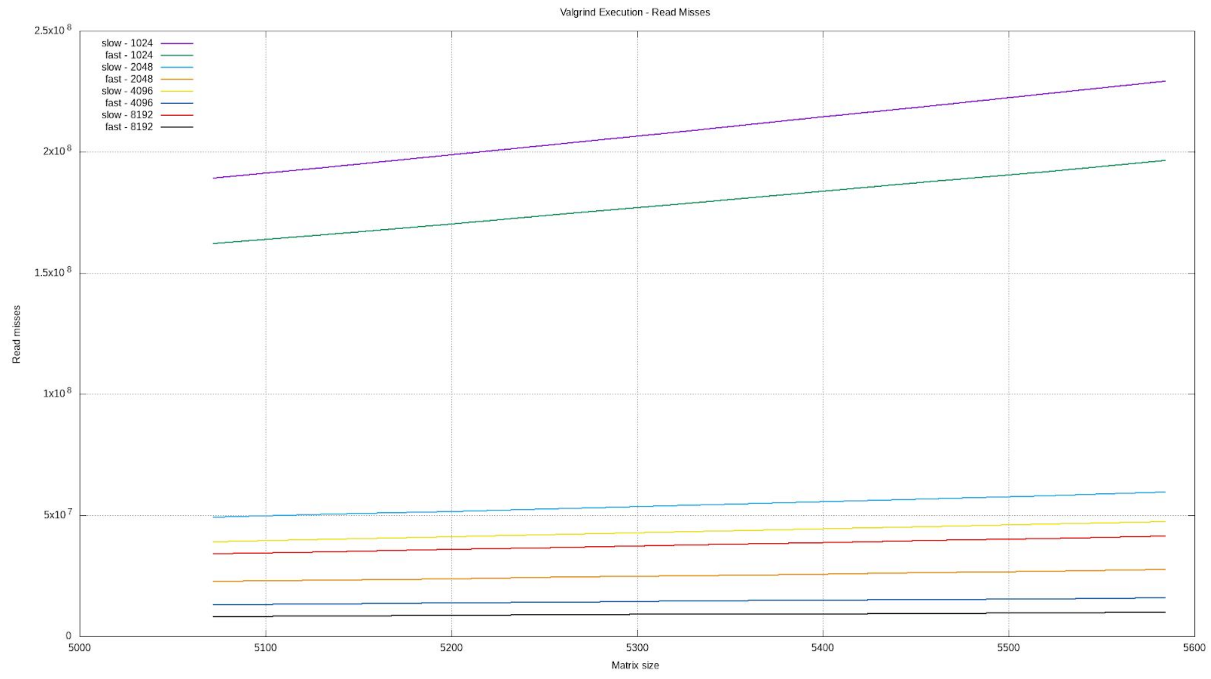
But for the slow program we don't access the memory in a sequential way, we may access the element 2 and then the element 6 (for example), so it is less probable that the 6 is in the same block of the cache. Therefore, there are going to be more cache misses for those non-sequential accesses.
The fast program is always going to have less read misses but when we refer to write misses, both have the same misses because we are writing in the same way in both of the programs.

Does the trend change when you look at a specific cache size and compare the slow and fast program?

Apparently it doesn’t, the misses grow when the matrix size grows too, and it does in a linear way.

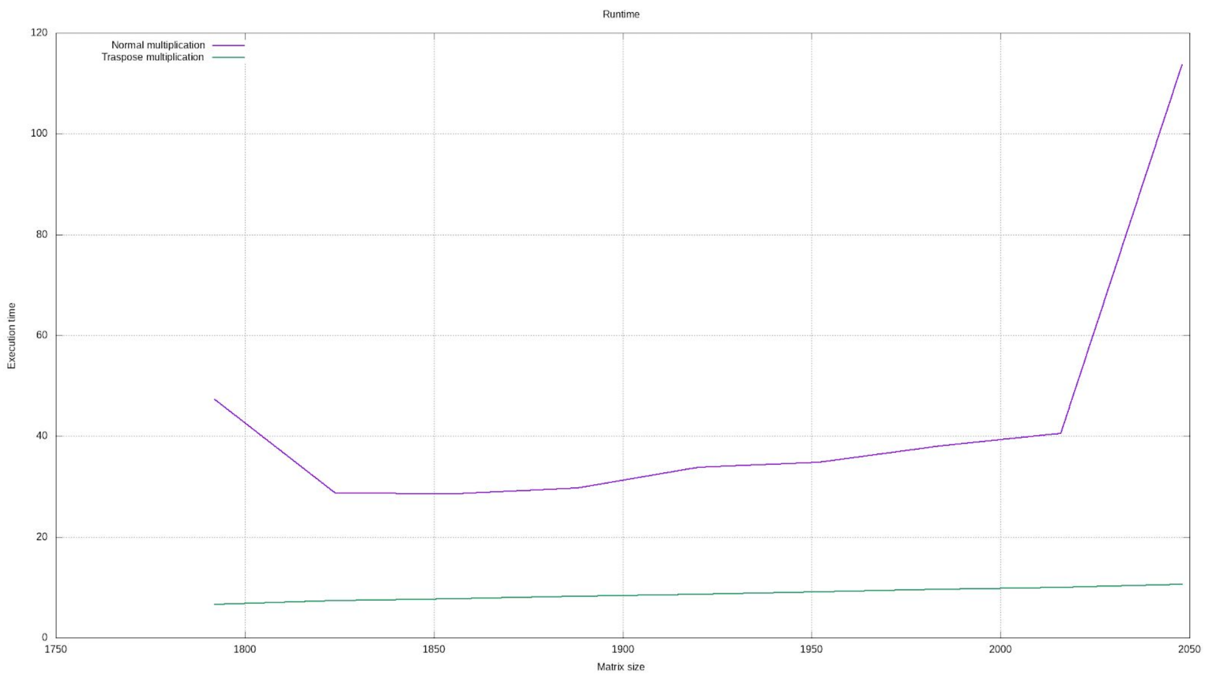
Observation: In the write misses we can see that there are only 4 lines instead of 8, this is because the write misses are the same for both programs because in both of them we write in the same way.

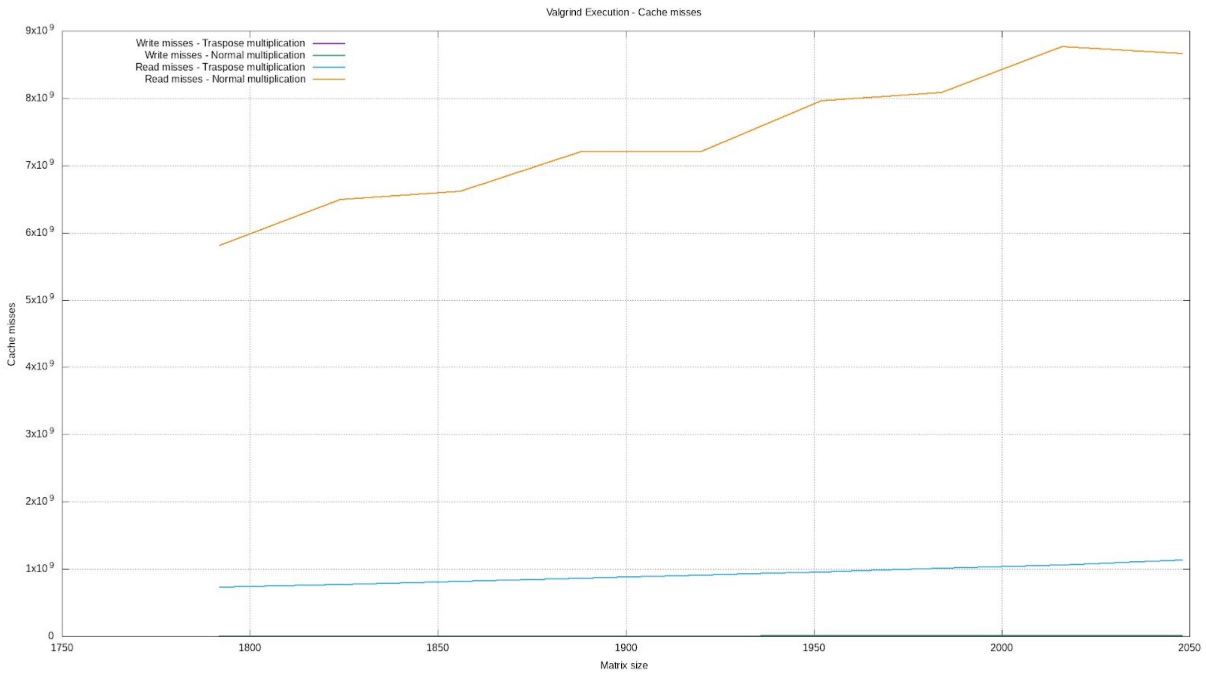




d. Cache and Matrix multiplication

In our case, we obtain the execution time and cache misses for the programs multiplication and mul_trasp using matrices of size 1792*1792 (256 + 256*6) up to 2048*2048 (256 + 256*(6+1)) in increments of 32.





Are there any changes in the trend when varying the sizes of the matrices?

No, but, regarding the execution time, we can see that after the size is 2016*2016, the time grows up very fast.

What are the observed effects?

We can see that using a transposed multiplication is more efficient than using the normal one except for reading. This is because in the transposed multiplication we multiply row by row, which means that the data we are looking for is more likely to appear as we said in **2.b**.
We can also observe that the write misses compared to the read misses are very low.