Universidad Autónoma de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática

COMPUTER ARCHITECTURE

PRACTICE 3. CACHE AND PERFORMANCE.

Daniel Mohedano Rodríguez
Silvia Sopeña Niecko

03/12/2020

Contents

List of Figures

Exercise 0

With the command *cat /proc/cpuinfo* we found the following information:

- There are 8 processors numbered 0 to 7.
- Each one has a cache size of 6144KB.
- The cache alignment is 64B.
- They use 39b physical addresses and 48b virtual addresses.

With *dmidecode* we found the following information regarding each level of cache:

- There are 3 levels of cache.
- The 3 levels operate with write back.
- Level 1 is 256KB 8-way set associative.
- Level 2 is 1024KB 4-way set associative.
- Level 3 is 6144KB 12-way set associative.

With *getconf -a | grep -i cache* we found the following information:

- The line size for every level of cache is 64B.
- And more information that is more clearly transmited through the output of *lstopo*.

As it can be seen in 1, all the information described earlier comes together in the diagram. The 256KB of level 1 cache are distributed between the 4 cores of the computer. And for each core it is equally distributed between level 1 instruction cache and level 1 data cache (each of 32KB). The 1024KB of level 2 cache are also distributed between the 4 cores. Finally, the level 3 cache is common for all cores.

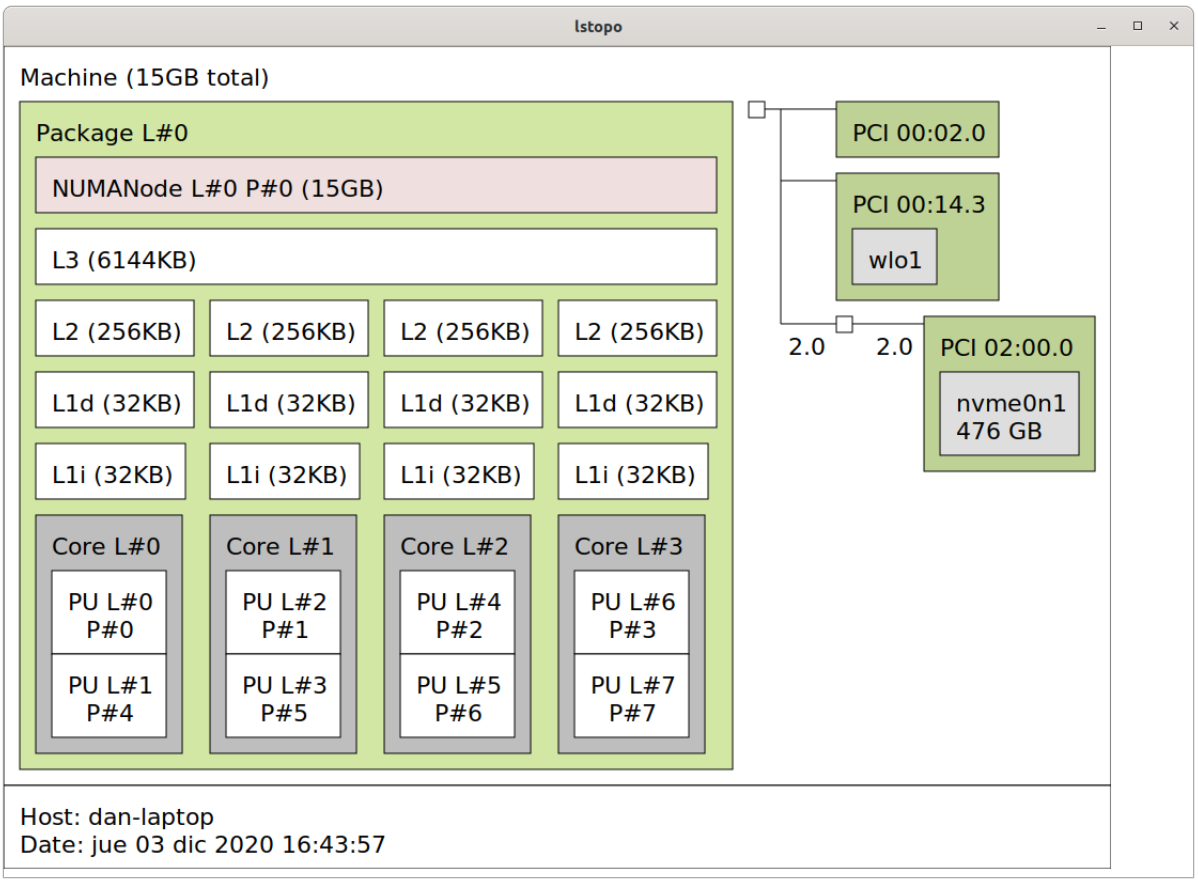

Figure 1: Output of the command *lstopo*.

Exercise 1

The reason why performance measurements need to be taken multiple times for each program and matrix size is to avoid "false" results. That is, to avoid the state of the processor or cache in the moment of execution to have an impact in the results. By repeating the process several times and calculating the mean of the results you end up filtering that "noise" and in the end get a more consistent result.

This can be clearly seen in the following figures:

The script used interleaves the execution of both programs, *slow.c* and *fast.c* in the following way:

$$slow_{N1}, fast_{N1}, slow_{N2}, fast_{N2}, ..., slow_{Nf}, fast_{Nf}$$

Then this process is repeated the number of iterations indicated.

When executing the script with only one repetition the figure obtained presents some rough peaks, specially in the execution time of the *slow* program. This means that the variation in time of execution varied irregularly. This is to be expected, as it was mentioned earlier, because the execution time depends on the state of the processor and memory when executing.

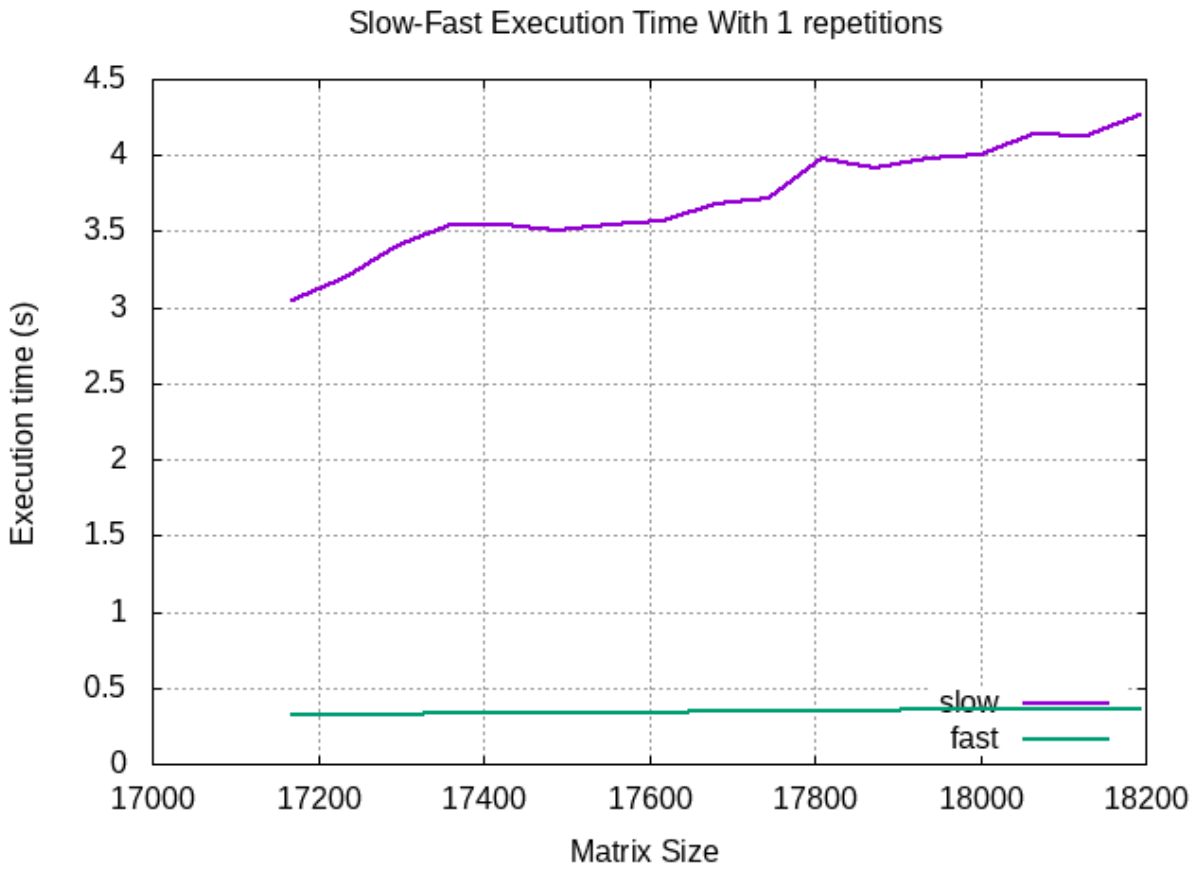

Figure 2: Slow-Fast with 1 repetition.

In order to get a more consistent result we incremented the number of repetitions until the graph smoothed out. This was achieved at 8 repetitions, as it is showed in **Figure 3**.

With this, arguably, more consistent result it is now safe to confirm that the difference in execution time between both programs is smaller with smaller matrix sizes.

The explanation for this is straight forward. The matrices are created as a single array of length $n \times n$ and then each row is "assigned" as the direction in memory of the $i \times n$-th element. Elements of the same row are stored consecutive in memory. Accessing then the positions of the matrix by column instead of row completely ignores the Principle of Locality and is the reason why the *slow* program has greater execution times than the *fast* program. The reason for a smaller difference in time at slower sizes is the same.

To illustrate this, let's propose a simple example. Imagine a matrix implemented as a single array of length $N = n \times n$. For the sake of simplicity let's also imagine that the cache only holds one block of data at a time and that the matrix can be stored in exactly 2 blocks of data. When accessing the matrix by rows, you would only have to load into the cache two blocks of data in all the execution. The first one at the start and the second one when reaching row number $\frac{n}{2}$. The majority of the times (every-time except 2) the iteration of the loop wouldn't give a cache miss. If instead of accessing the matrix by row we access it by column the problem is that for every iteration of the loop the number of misses would be two. This is because for the first half of the rows we would want to read block 1 and for the second half

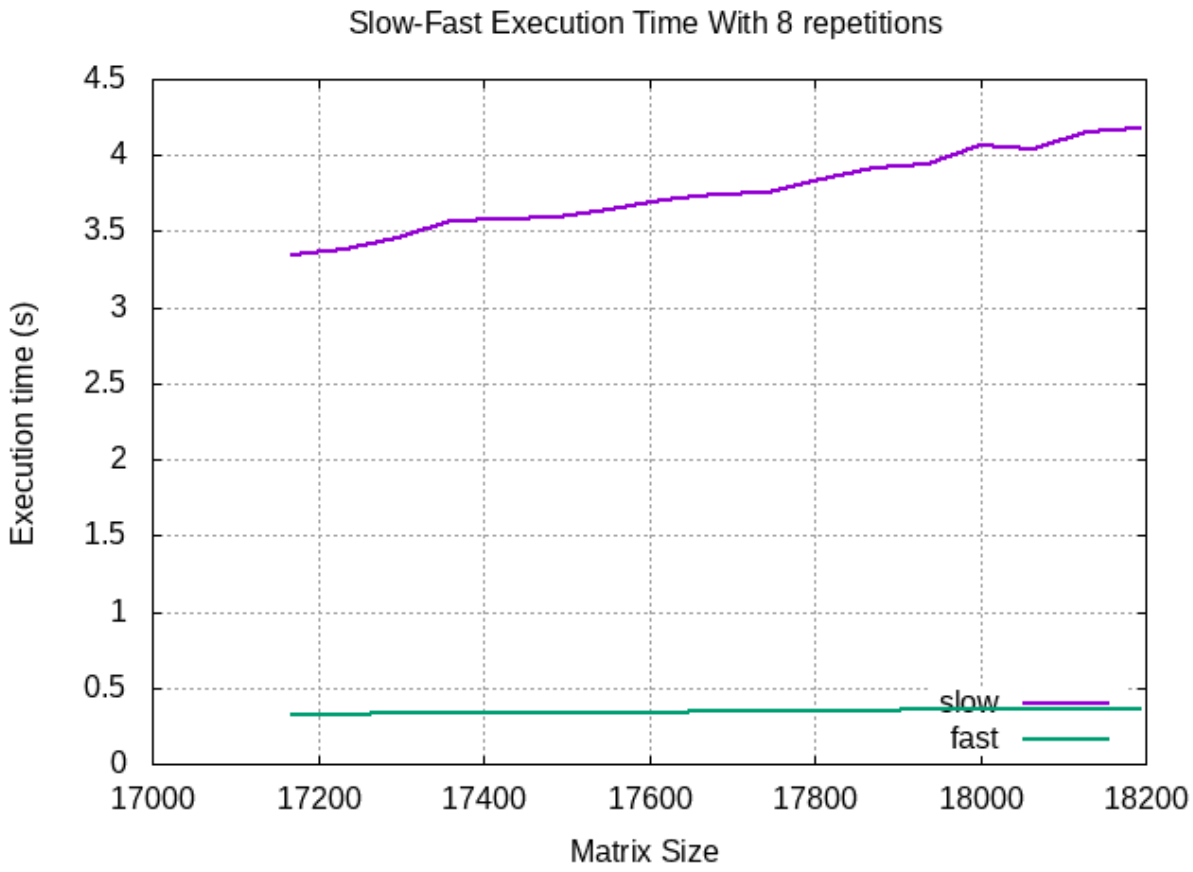

Figure 3: Slow-Fast with 8 repetitions.

of the rows we would want to read block 2. As this happens in every iteration of the loop, the program would obviously be much slower. If we now make the matrix twice as big (it can be stored in 4 blocks) the number of misses when accessing by row would be 4 but the number of misses when accessing by column would be $4 \times n$, as it is 4 for every iteration of the loop. This is exactly why with a bigger matrix size the difference in execution times also becomes bigger.

Exercise 2

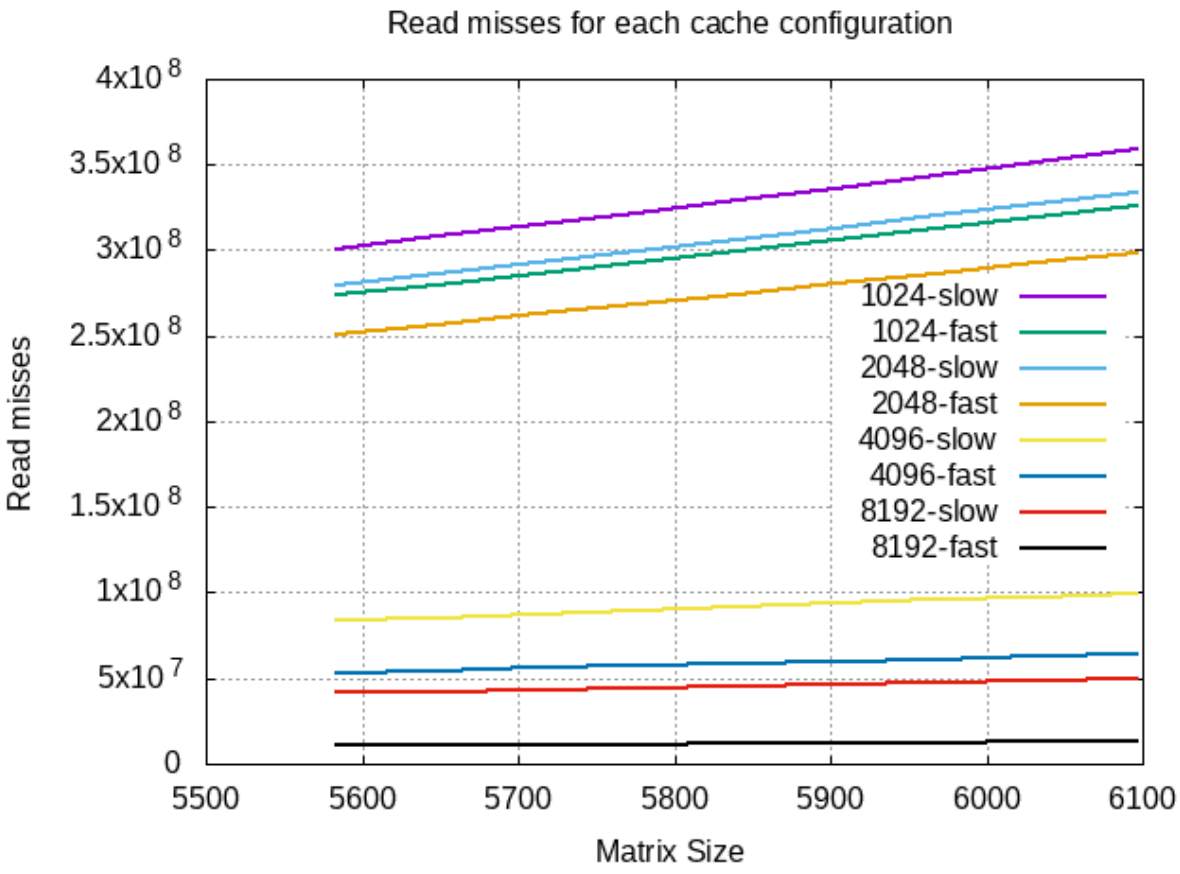

Figure 4: Read misses during execution of Slow-Fast.

The first obvious trend seen in **Figure 4** is that a bigger matrix size correlates with a larger amount of read misses. This makes complete sense.

The second observation we can make is that for the same cache configuration (same size of level 1) the execution of the *slow* program has always more read misses than the execution of the *fast* program. This was also expected and basically confirms what we explained in Exercise 1: the slower execution times are caused by accessing by columns the matrix, causing a larger amount of cache misses in the process. This observation holds for every configuration of cache tested.

The final main trend present in the figure is that a bigger cache size correlates with a smaller amount of cache misses (when comparing the same execution program). Again, this is expected as a bigger cache means that a larger amount of blocks can be stored in it and thus a smaller amount of cache misses is caused.

Apart from these main characteristics of the figure there are also a couple of small details that we have found interesting. For example, the slope of the curves seems to be steeper for the executions of programs in smaller caches. This could mean that the impact of a bigger matrix size in the amount of read misses is lessened when using a bigger cache.

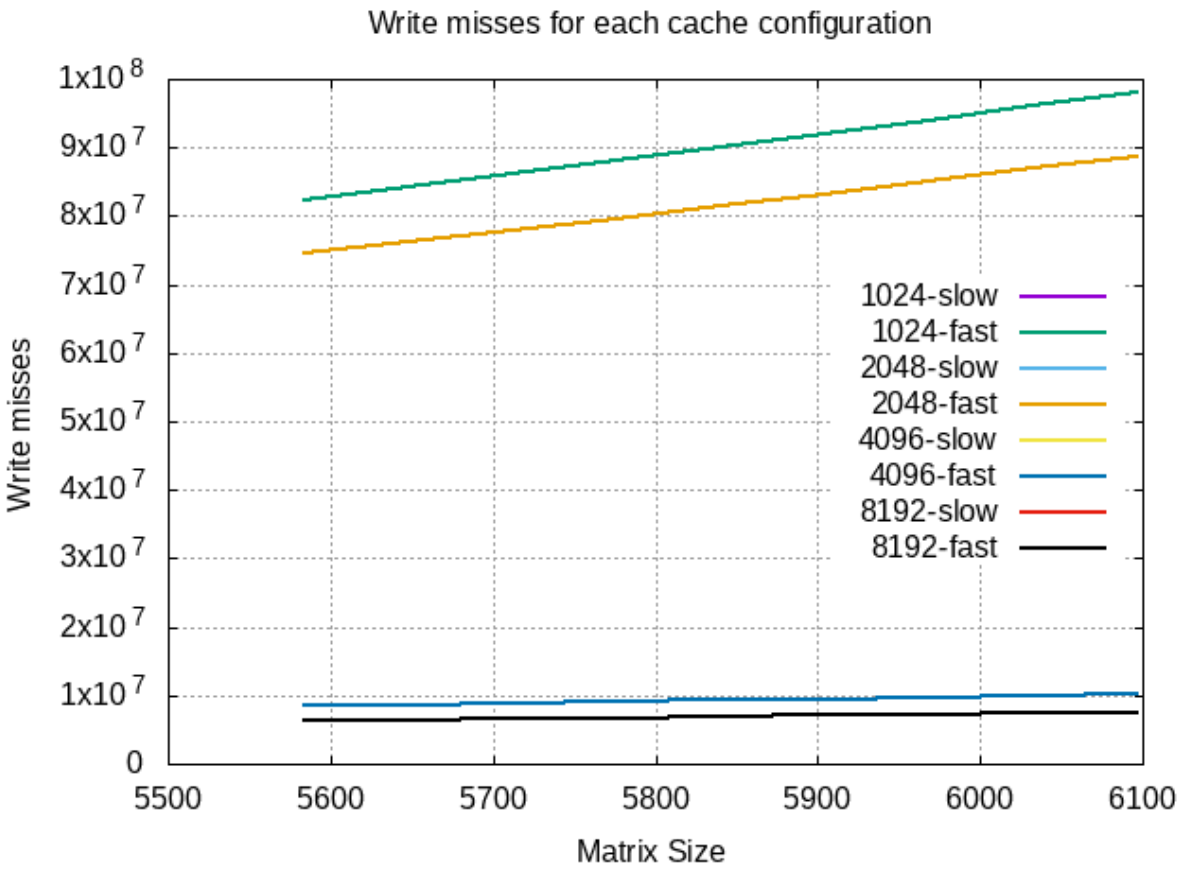

Figure 5: Write misses during execution of Slow-Fast.

In **Figure 5** we can observe that there are only 4 curves visible. This happens because of the overlap of the fast and slow curves for each cache configuration. There is no difference between the way *slow* and *fast* write in the matrix so it makes sense that the number of misses is the same in both programs. We can also see that the smaller the cache memory is, the larger the amount of misses appears to be, exactly as it was expected (and also explained through **Figure 4**).

Exercise 3

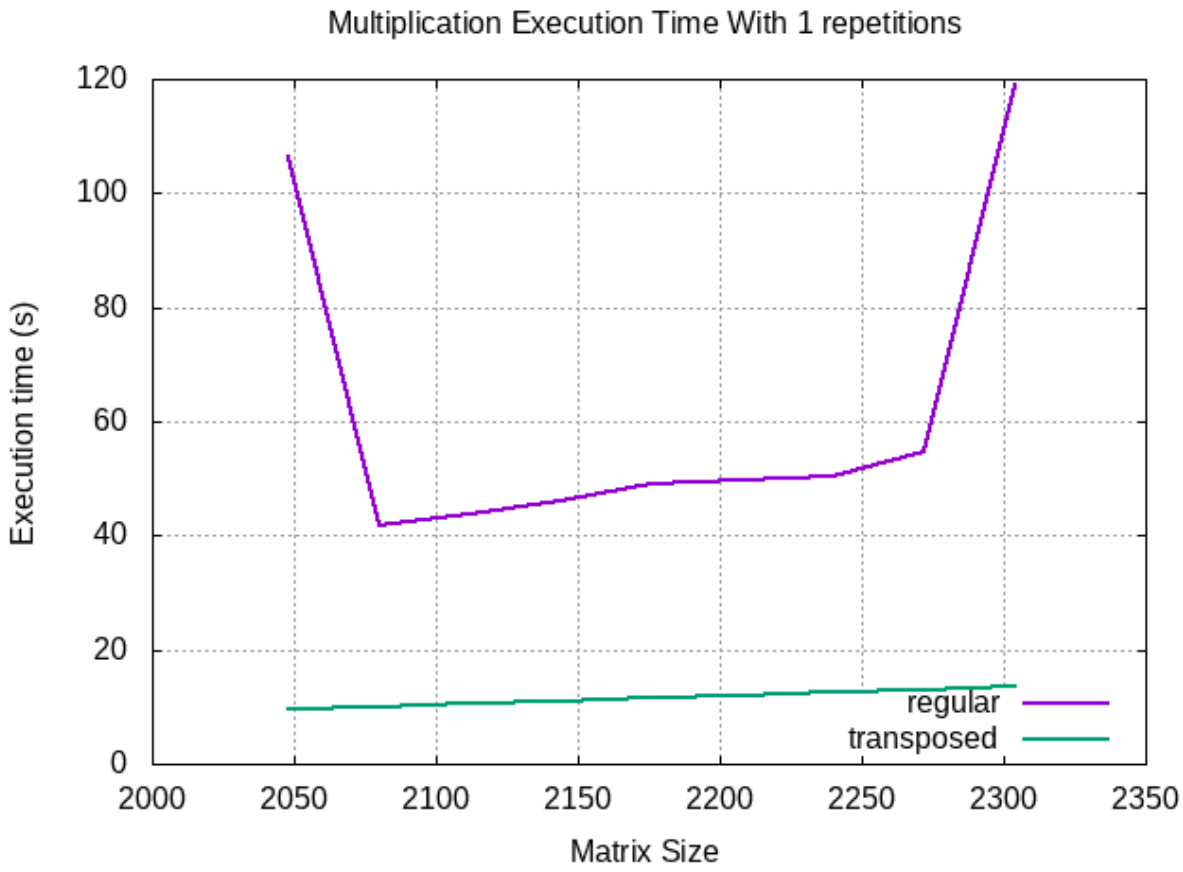

Figure 6: Multiplication with 1 repetitions.

First, we executed the script with only one iteration to check the results. We found that it had some big peaks and the curve wasn't as smooth as we imagined so we continued testing with a bigger number of repetitions. We tested up until 8 repetitions, where we saw that the roughness of the curve wasn't really changing. So, this was probably not a problem of repetitions.

The main trend in **Figure 7** is that the transposed multiplication appears to be faster than the regular multiplication. This could make sense as the access by column in the transposed multiplication is only done while calculating the transposed matrix. Because of this, the cache is "only" being used to store blocks related to matrix *B*. When doing the regular multiplication instead, the access by columns of matrix *B* is done while also accessing blocks of matrix *A* and thus having a smaller amount of cache to store data from *B* (which is the matrix that causes the most cache misses). This could mean that the regular multiplication causes a larger amount of misses and, in result, a bigger execution time. It is also important to mention that in the transposed multiplication the access by columns is done inside only two nested loops while the regular multiplication accesses by columns inside three nested loops (thus more accesses and more misses).

The other obvious thing when looking at the figure is the significant peaks that appear in the curve for the regular multiplication. As it can be seen, even with 8 repetitions of the process (around 4 hours of

Figure 7: Multiplication with 8 repetitions.

execution) these peaks still appear. We have determined that this is not an execution problem as the tests were repeated multiple times, the order of execution of the matrix sizes was also changed and the peaks still persisted. Our hypothesis for this is that when doing the regular multiplication, which involves a triple for loop while also accessing by columns inside the most inner loop, the specific matrix size has a much bigger impact in the performance compared to the addition programs or even the transposed multiplication. Our best guess is that certain matrix sizes cause more conflict misses because of the way those matrices fit in the blocks. Putting this two things together could explain the massive peaks.

What had been previously mentioned can be seen in Figure 8. The regular multiplication method yields a larger amount of cache read misses compared to the transposed multiplication. Another important thing to mention is that we don't see the spike of cache misses we were expecting to justify the abrupt change of execution times for the regular multiplication. But, this could also have an explanation. In the figure only the misses of the Level 1 cache are shown. And, while there isn't a peak of Level 1 cache misses for the specific matrix sizes, it could very well be possible for those peaks to be caused by misses of other levels of cache. This would also explain the abrupt change of execution time as, compared to level 1 cache, the other levels are far slower (or even having to access main memory if all the levels fail).

In the case of cache write misses though, it is the other way around. As presented in Figure 9, the transposed multiplication execution causes more write misses than the regular multiplication. Again, this is expected because while the regular multiplication only writes into the result matrix, the transposed multiplication has to calculate the transpose first, effectively writing twice as much. This

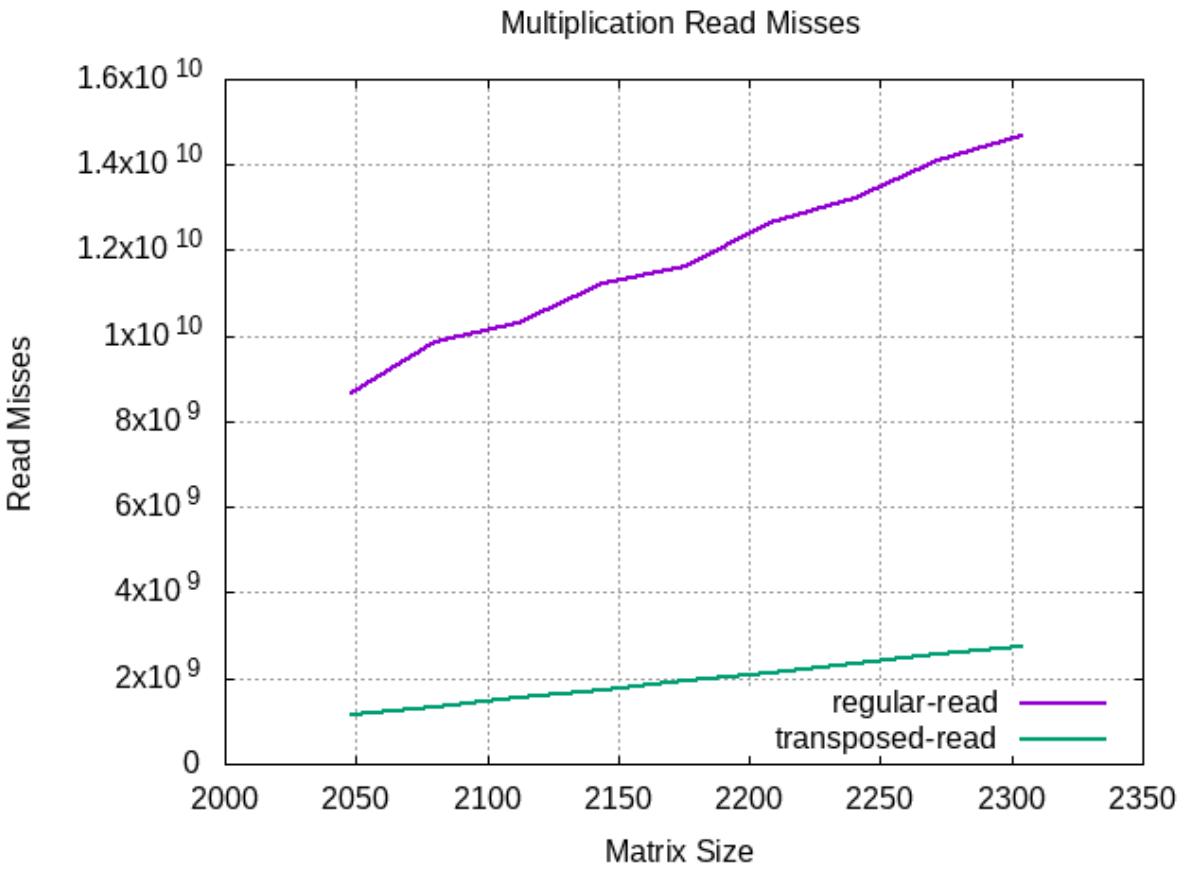

Figure 8: Cache read misses of matrix multiplication.

causes the transposed multiplication to have close to twice as much write misses compared to the regular multiplication.

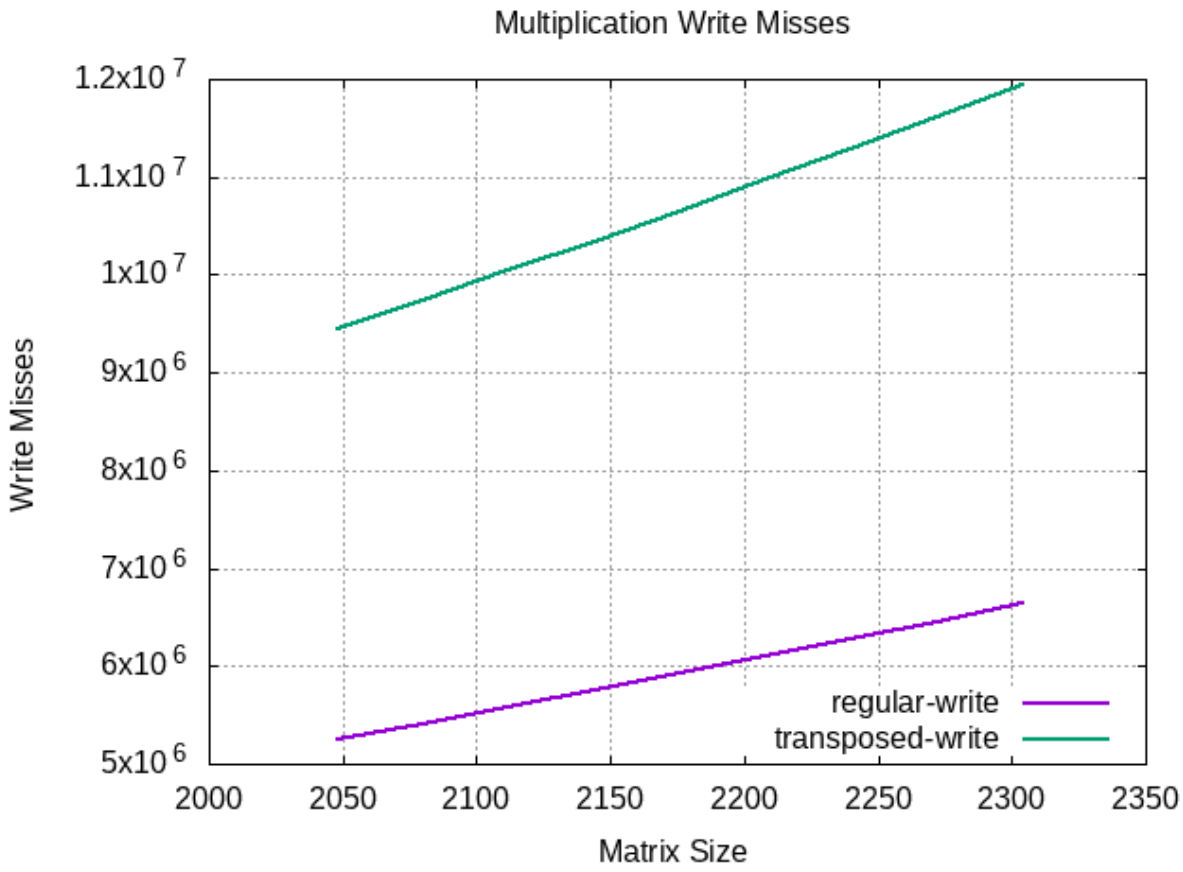

Figure 9: Cache write misses of matrix multiplication.

Organization of delivery

All the code provided to us (the *slow.c* and *fast.c* programs) is in the directory **code/**.

Then, for each exercise (excluding exercise 0), we created a directory to store the bash script used to execute and plot the tests and two sub-directories: **data/** (where the data files are stored) and **plot/** (where the figures are stored).

The file names for the data and figures aren't exactly as the ones mentioned in the practice guide in order to avoid confusion (we added to the file names the amount of repetitions needed to obtain the data). In the case of exercise 3 we decided to split the data in two files (execution times and valgrind) in order to keep the script more organized and to make it possible to execute different parts of the script individually.

The bash scripts have been organized around some flags that determine what parts are executed (to avoid having to execute the whole script when just wanting to plot for example). These flags can be changed manually in the scripts in order to decide which parts are executed.

The data used to plot the figures has also been included in the delivery (just in case).