Arquitectura de ordenadores. Practice 3.

Pablo Cuesta Sierra, Álvaro Zamanillo Sáez

November 16, 2021

Exercise 0

We have been submitting our tasks to the *MV* queue for all exercises of the practice because it returned the best (smoothest) results.

After running the commands shown in the statement, we got these results.

As can be seen in figure 1, the first level is the only one where there are two different caches: one for data and another for instructions. Both are of the same kind: size of 32768 B (32 KB), with blocks of 64 B and 8 ways. In the second level, there is a unique cache which is larger (2097152 B = 2 MB) but has the same degree of associativity. For the third level, the size increases (16777216 B = 16 MB) as well as the associativity (16 ways) whereas the block size remains the same.

```
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  2097152
LEVEL2_CACHE_ASSOC                 8
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  16777216
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Figure 1: Output of `getconf -a | grep -i cache`

Here is the info of the each of the processors (this is processor 0, but they are all the same):

```
processor : 0
vendor_id : GenuineIntel
cpu family  : 6
model   : 26
model name  : Intel Core i7 9xx (Nehalem Class Core i7)
stepping  : 3
cpu MHz   : 2399.996
cache size  : 16384 KB
fpu   : yes
fpu_exception : yes
cpuid level : 11
wp    : yes
flags   : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat pse36 clflush mmx
    fxsr sse sse2 syscall nx lm constant_tsc rep_good xtopology unfair_spinlock pni ssse3
    cx16 sse4_1 sse4_2 x2apic popcnt hypervisor lahf_lm
bogomips  : 4799.99
clflush size  : 64
cache_alignment : 64
address sizes : 46 bits physical, 48 bits virtual
power management:
```

Figure 2: First lines of `/proc/cpuinfo`

The following diagram shows the archiquetecture of the different caches in each of the 8 cores.

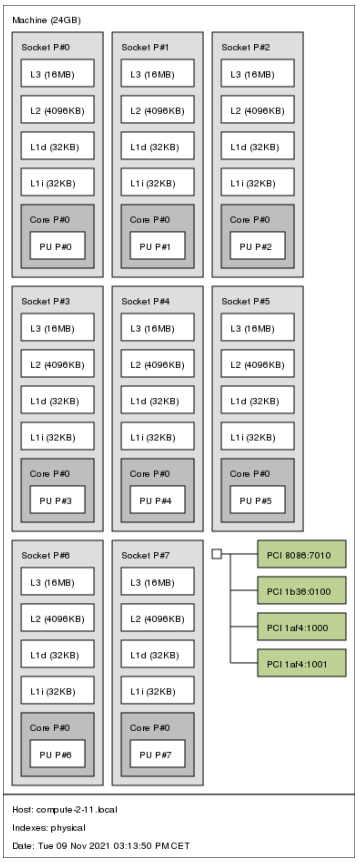

Figure 3: Topology of the used CPU.

Exercise 1

The following image shows a graph with the execution time of both programs for different sizes of the matrix.

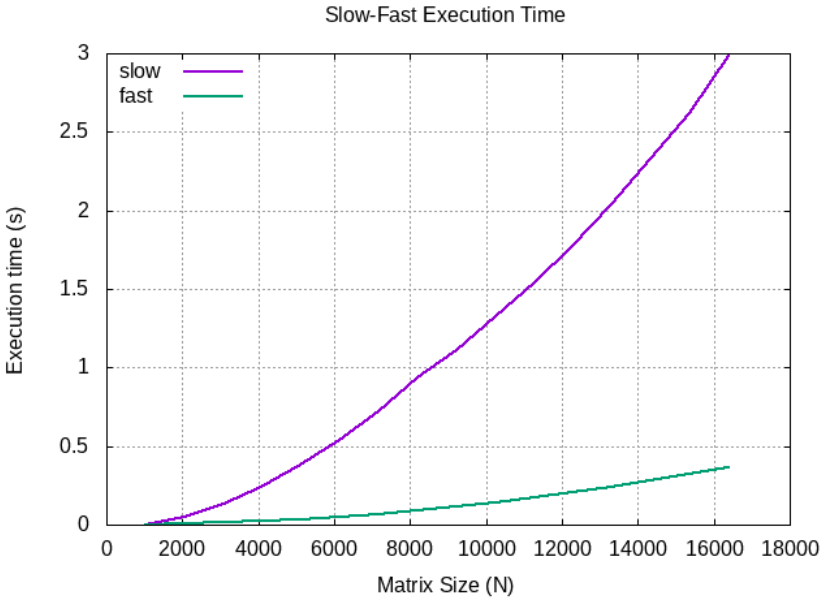

Figure 4: Slow vs Fast execution times.

As pointed by the graph, the execution time for small values of N is similar for both programs (most of the matrix can be stored completely in the cache). However, as N grows, the execution time for the slow program shows a significant cuadratic behaviour while the fast version barely grows.

This is due to the fact that the matrix is stored by rows in main memory, and therefore stored in the cache by rows too. The slow version access the matrix by columns which leads to a higher number of misses and consequently, a higer number of accesses to main memory (hundreds of times slower).

For obtaining the results, it is necessary to run the simulation several times in order to calculate a mean and have more than one result, so that the programs do not present abnormal outliers because of other factors different from what we are measuring. Furthermore, the chosen execution pattern alternates between the slow and fast version to prevent the programs from reusing the cache from the previous execution (with the same size).

Exercise 2

First, we will analyze the write misses:

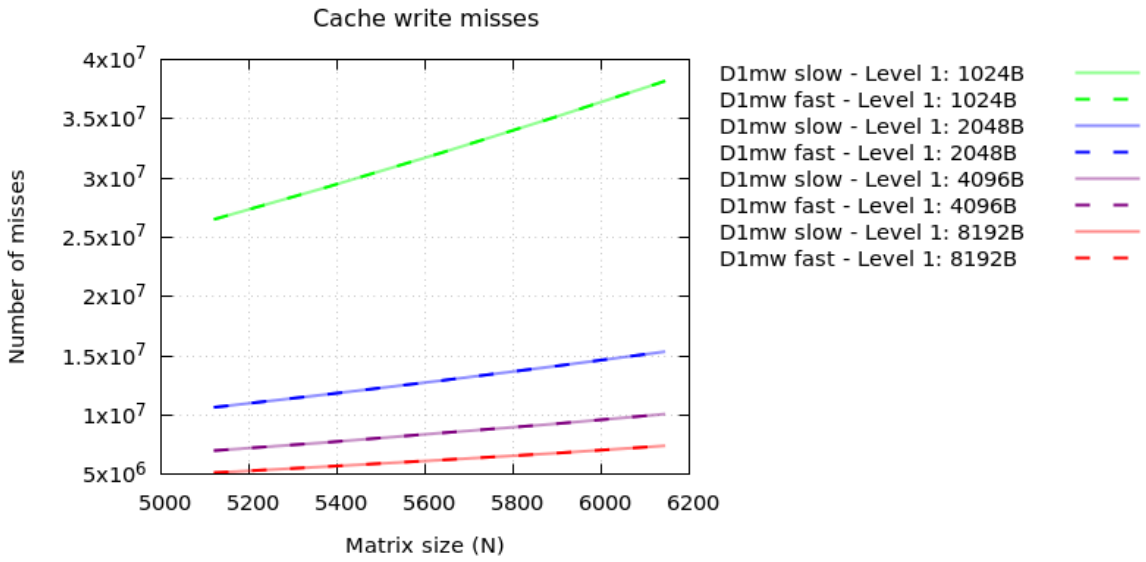

Figure 5: Cache write misses.

It may seem that only four lines have been plotted, however this is not the case. (We have plotted the slow graphs with a bit of transparency so that the fast graphs can also be seen). The slow and fast version have (almost) the same number of write misses (for the same execution values of N and size of the cache). The reason behind this is that both programs fill (write) the initial matrix in the same way, by rows (`generateMatrix(int size)`).

On the other hand, the read misses show different results:

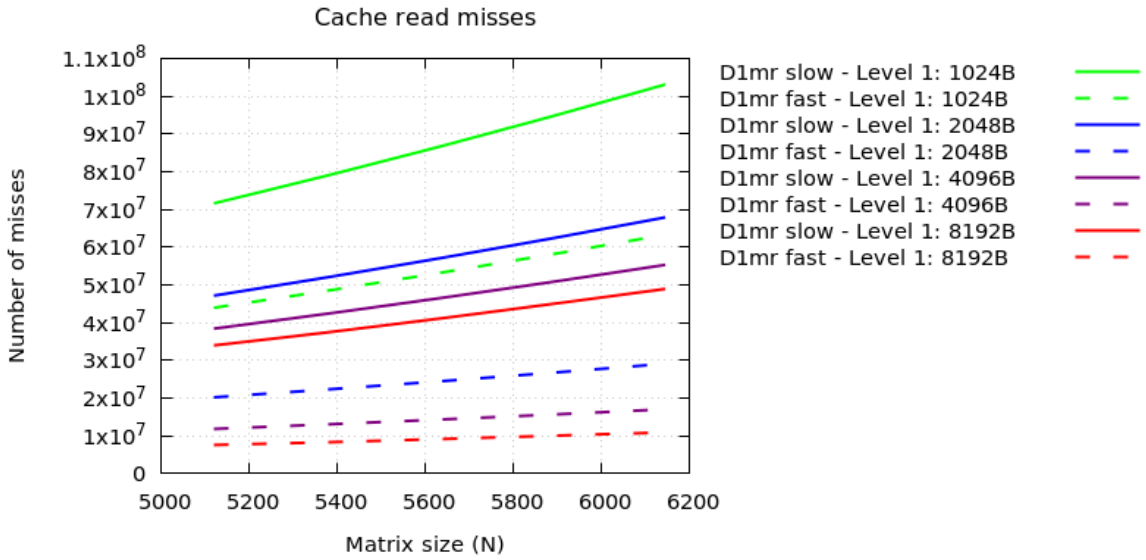

Figure 6: Cache read misses.

This time, we get 8 different lines, as one could expected beforehand. The dotted lines show the fast

version, which necessarily must have less read misses than the fast one (when compared with the same cache size). Again, this is due to the fact that the matrix is stored by rows (one row beside the next, sequentially), and the slow version accesses by columns, while the fast version accesses by rows.

The effect of the cache size can also be observed as the fast version for the smallest cache produces more misses than the slow version of the two largest caches. In general, it is to be expected that smaller sized caches would provoke a higher number of misses for the same program; which is what we see in figure 6.

Exercise 3

The image below (figure 7) shows the execution time for the two different multiplication methods. The transposed multiplication is significantly faster as it accesses matrixes by rows when multiplying. On the other hand, in the regular multiplication, the second operand matrix is accessed by columns which produces a high rate of read misses and as a result a higher execution time.

Commenting on the strange peaks that appear in figure 7, we do not know the real reason why this happens. These peaks are neither because of some mistake on calculating values, nor inconsistencies in execution that could be solved repeating the tests. The values of over 10 seconds when executing for size $N = 1408$ are pretty consistant when executing in queue *mv.q* of the cluster (the same goes for the rest of the matrix sizes); meaning that the time in each iteration does not vary more than $0.1s$ for each N. If we execute the same script in another queue (*amd.q*, for instance), peaks still appear, but not necessarily in the same sizes (N). We think this strange and counterintuitive beahviour might be caused by inefficient use of caches, and that certain sizes of matrixes are managed more efficiently by the cache (because its size is a multiple of the block size, for example) than others; even if this other ones are smaller.

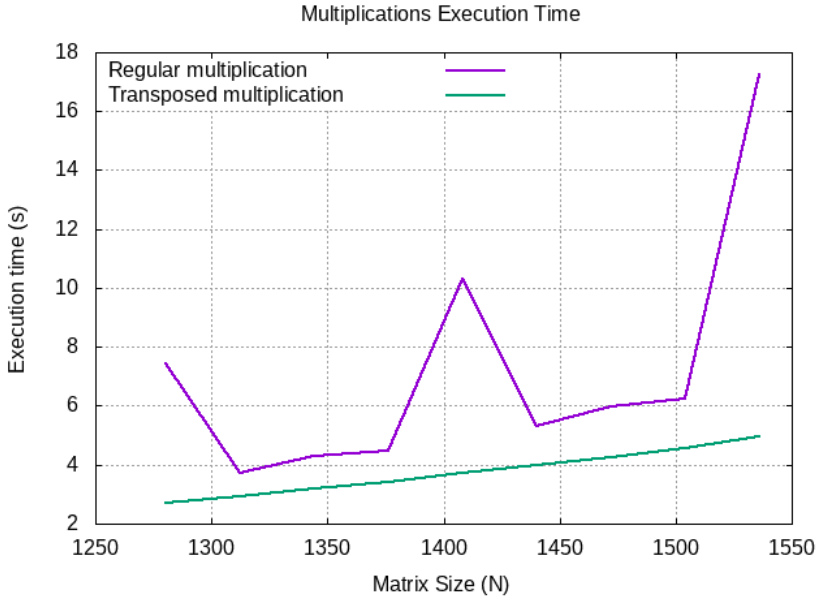

Figure 7: Multiplication execution time

When simulating with *valgrind* (figure 8), we get very similar results to what we saw in exercise 2 (figure 6). The less optimized program (regular multiplication), which reads matrix $\mathbf{B}$ by columns (instead of by rows), produces a higher number of read misses than the "transposed multiplication", which accesses matrix $\mathbf{B}^T$ by rows.

The regular matrix multiplication shows a step pattern every two times we increase the value of N. This may be due to alignment of the data as we are increasing the matrix by 32 and the block size of the cache is 64B.

The write misses cannot be appreciated in figure 8, as they are much lower in value than the read misses, so we have plotted them in a separate graph (figure 9). Like in exercise 2, both programs write in the

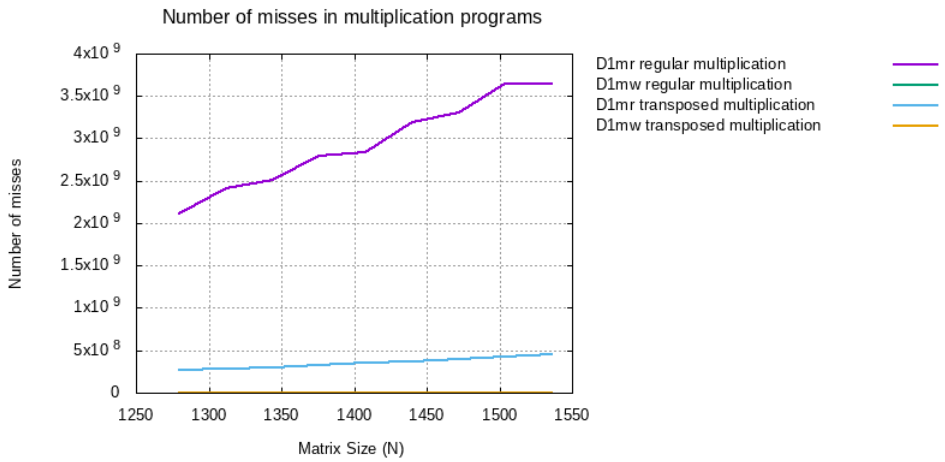

Figure 8: Cache misses of the multiplication programs

same manner: when generating matrixes **A** and **B**, and when writing the result in matrix $\mathbf{C} = \mathbf{AB}$. So, as expected, the write misses are very similar. If we look closely at the exact numbers, we can see that the number of write misses is a bit higher (by less than 10 units) in the transposed version (for all sizes of the matrixes). This is completely normal, bacause the transposed multiplication has to (re)write most of the values of matrix **B** when transposing it before multiplying; this is what causes the (insignificant) extra write misses in this program, which do not appear in the other one.

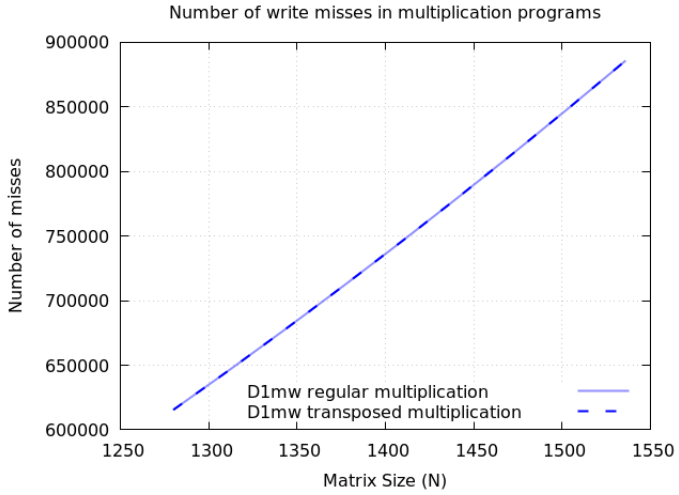

Figure 9: Cache write misses of the multiplication programs

Exercise 4

Figure 10: Cache read misses of the multiplication programs (varying associativity)

Figure 11: Cache write misses of the multiplication programs (varying associativity)