Arquitectura de ordenadores. Practice 3.

Pablo Cuesta Sierra, Álvaro Zamanillo Sáez

November 17, 2021

Note: during this whole document (in the plots), when we write "Matrix size (N)", it means we are plotting (normally in the x-axis) the values of N, where $N \times N$ is the size of the used matrices.

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 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.

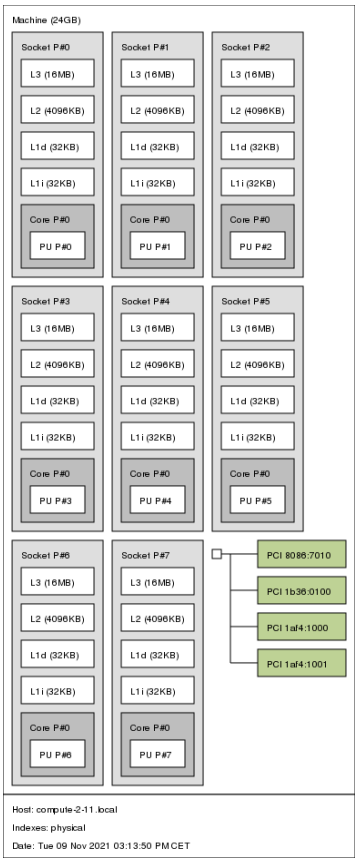

Figure 3: Topology of the used CPU.

Exercise 1

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

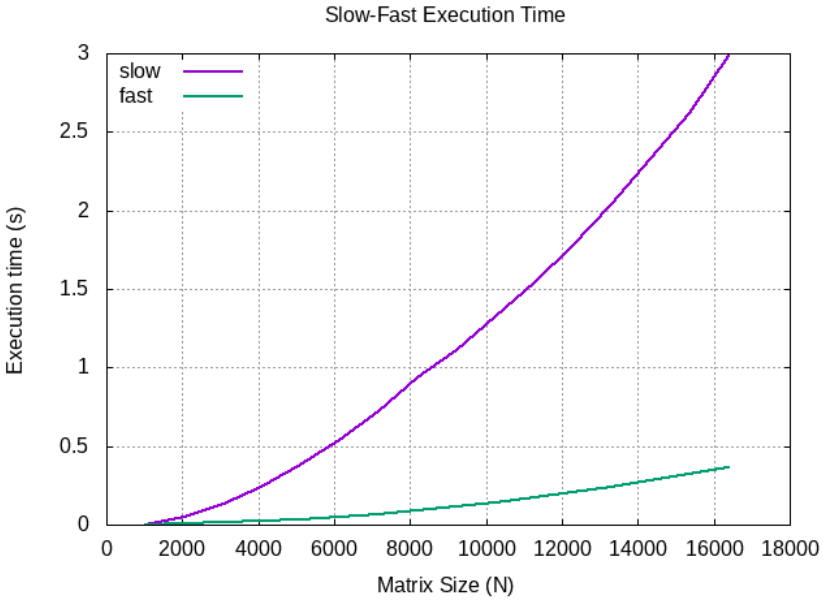

Figure 4: Slow vs Fast execution times.

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, it is 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:

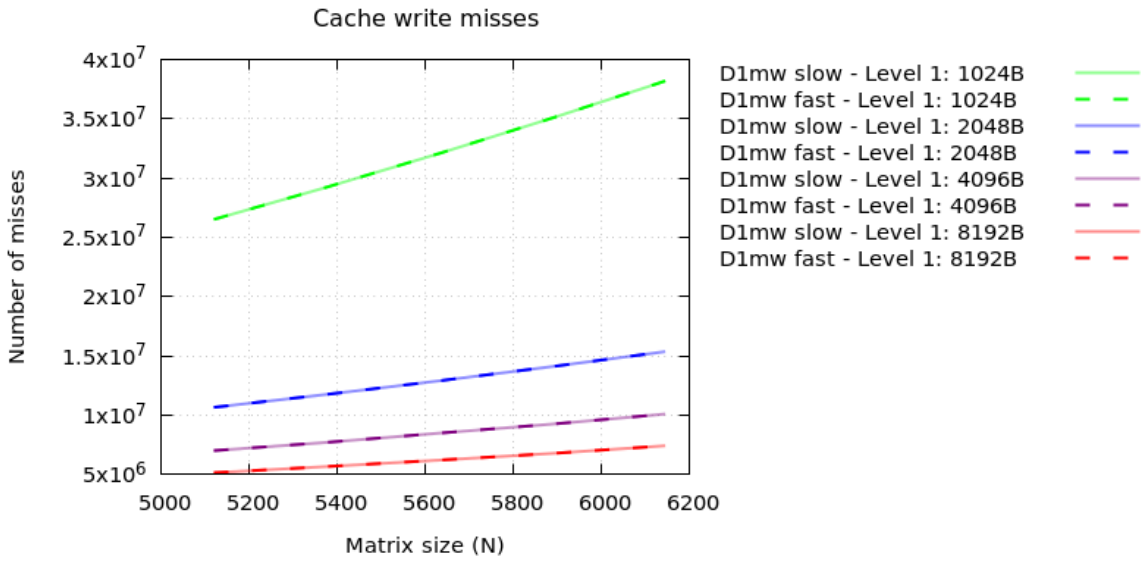

Figure 5: Cache write misses.

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, using the function ,generateMatrix(int size).

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

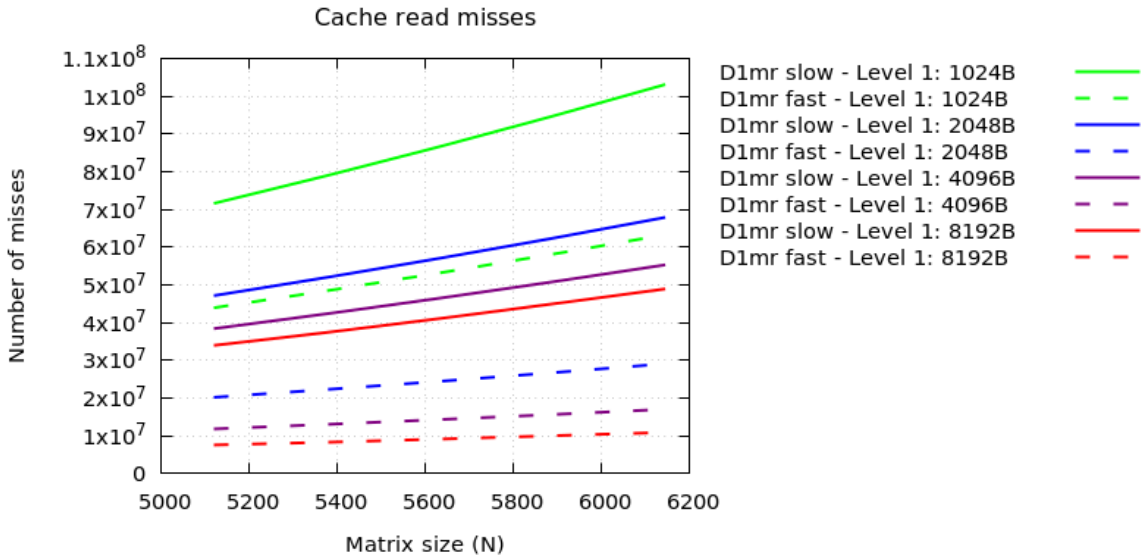

Figure 6: Cache read misses.

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 the matrix by columns, while the fast version accesses it 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.

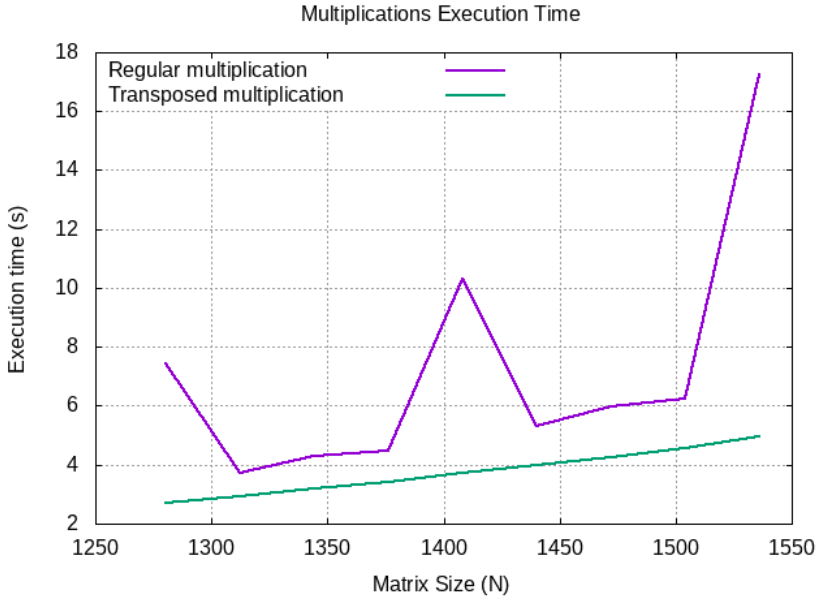

Figure 7: Multiplication execution time

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

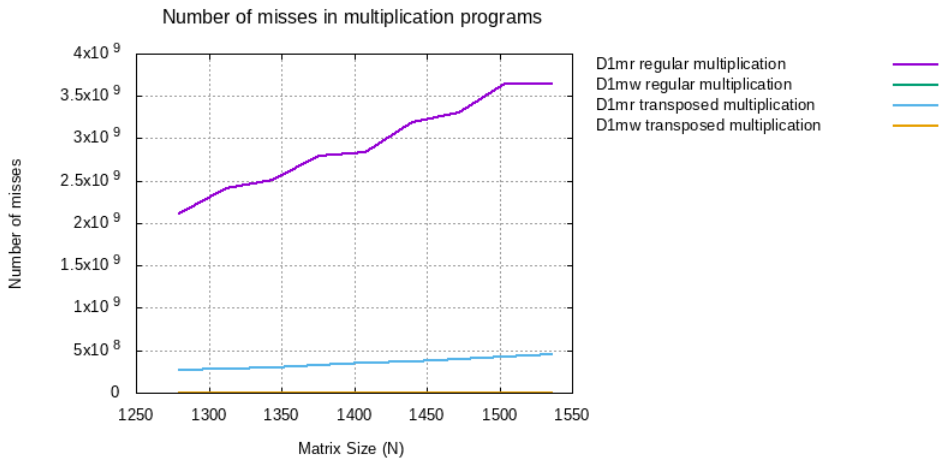

Figure 8: Cache misses of the multiplication programs

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.

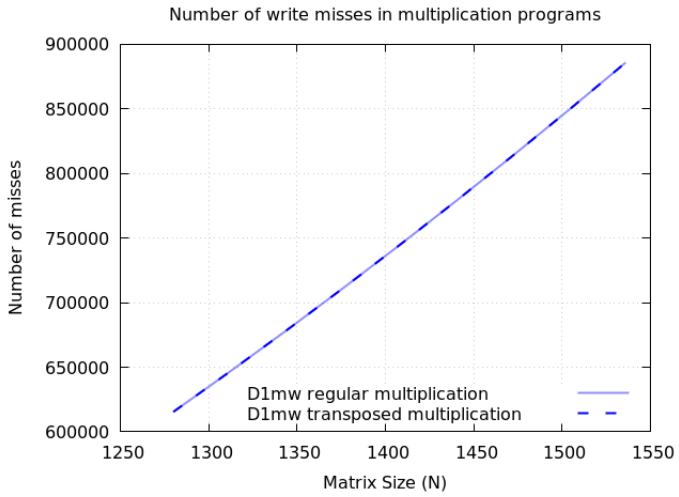

Figure 9: Cache write misses of the multiplication programs

Exercise 4

For this exercise, we have distinguished two main experiments. In each of them, we modify a different parameter of the cache, leaving the rest untouched. For both, as requested, we use the programs written for exercise 3: *multiplication.c* and *multiplication_t.c*

Exercise 4. Experiment 1: associativity

Here we simulate (using *valgrind*) the execution of both programs. We fix the line (block) size (64B), as well as the cache sizes: level 1 caches (both data and instructions) are of size 4kB, and level 2 cache is of size 8MB. For each experiment, we test with matrix sizes $(N \times N)$ for $N \in \{3072, 3328, 3584, 3840, 4096\}$. The difference between experiments is the type of set associative caches. We test for 1-way, 2-way, 4-way and 8-way set associative caches.

These are the results:

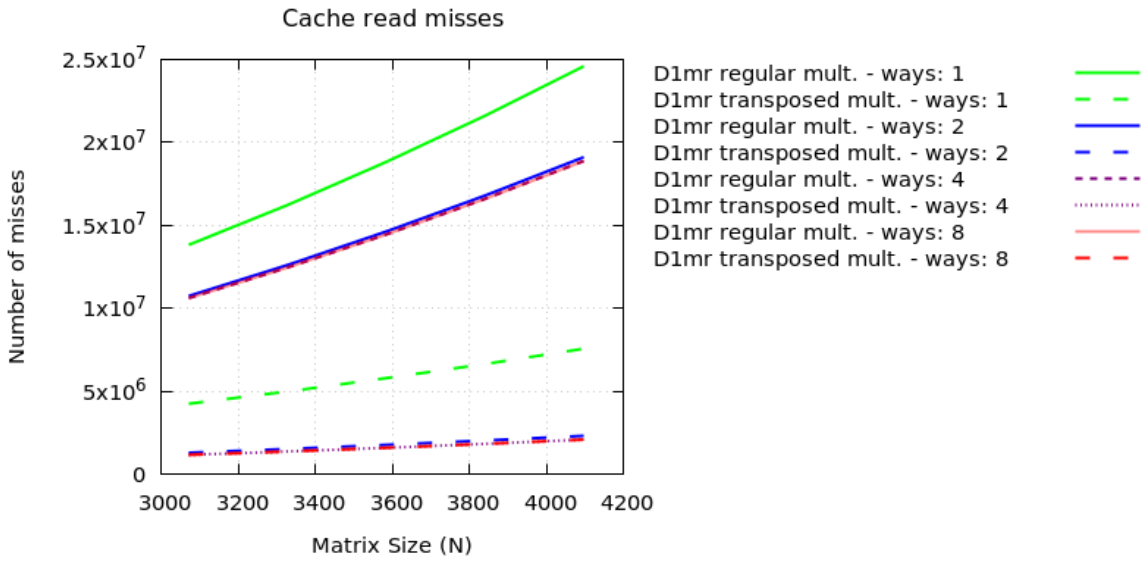

Figure 10: Cache read misses of the multiplication programs (varying associativity)

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

First of all, we will comment on the difference between both programs. Write misses (figure 11), for a fixed type of cache, are the same in both programs, as we already commented in exercise 3 (for figure

9). Read misses, on the other hand, are different. As could be expected, regular multiplication generates a much higher number of write misses than "transposed multiplication". Again, this is due to the fact that transposed multiplication accesses both matrices by rows, which is how they are stored in memory; making it so that data accessed is always (when multiplying) next to the previous data which has been recently used; whereas regular multiplication traverses matrix $\mathbf{B}$ by columns, making "jumps" in memory, and making recently loaded blocks of the cache useless.

Next, let us take a look at the differences caused by the changes in associativity. There are three distinguished sections in our plots (already explained in last paragraph): cache write misses, cache read misses of regular multiplication, and cache read misses of transposed multiplication. In all of these sections, the 1-way cache generates the highest number of misses, followed by the 2-way cache; 4-way and 8-way cache yield almost the exact same numbers, which are very close below the 2-way cache.

These results do actually make sense, and are consistant with the performance explained in the theory slides. The line size of the caches is 64B, and there are $\frac{4096}{64} = 64$ blocks in the (level 1) cache. Furthermore, each cache block contains 8 elements of a matrix (which means that, at a time, there cannot be more than 512 elements of a matrix in cache). Therefore, it is generally the case that a row occupies several blocks. This means that when $\mathbf{A}$ is being multiplied by $\mathbf{B}$, it frequently occurs (more than once per element $c_{i,j}$ of the result $\mathbf{C} = \mathbf{AB}$) that the current block being used of $\mathbf{A}$ has the same index as the one being used from $\mathbf{B}$. This implies that, if the cache is 1-way associative, the two current blocks will be replacing each other; thus the much higher number of misses for this kind of cache. However, when associativity increases, we do not longer have the problem that the two current blocks rewrite each other, so the number of misses decreases significantly.

Exercise 4. Experiment 2: block size

For this second experiment, we are going to execute the two multiplication methods with caches of different sizes (2048B - 16384B) and see how the block size affects each time to the resulting misses. (The caches used in the simulations are all 1-way associative.)

First of all, we are going to consider only read misses:

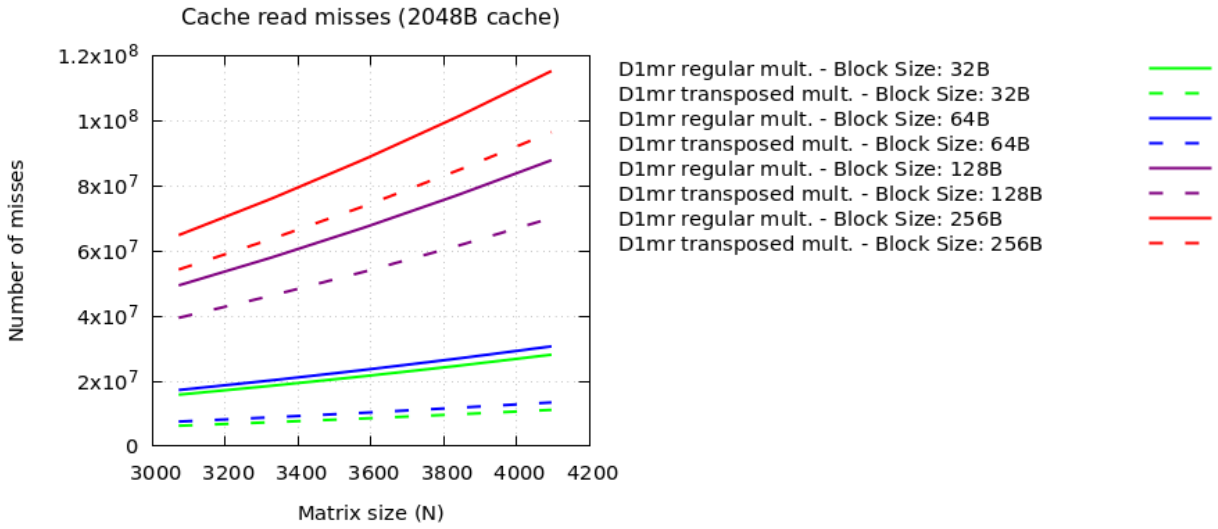

Figure 12: Cache read misses with a cache of 2048B

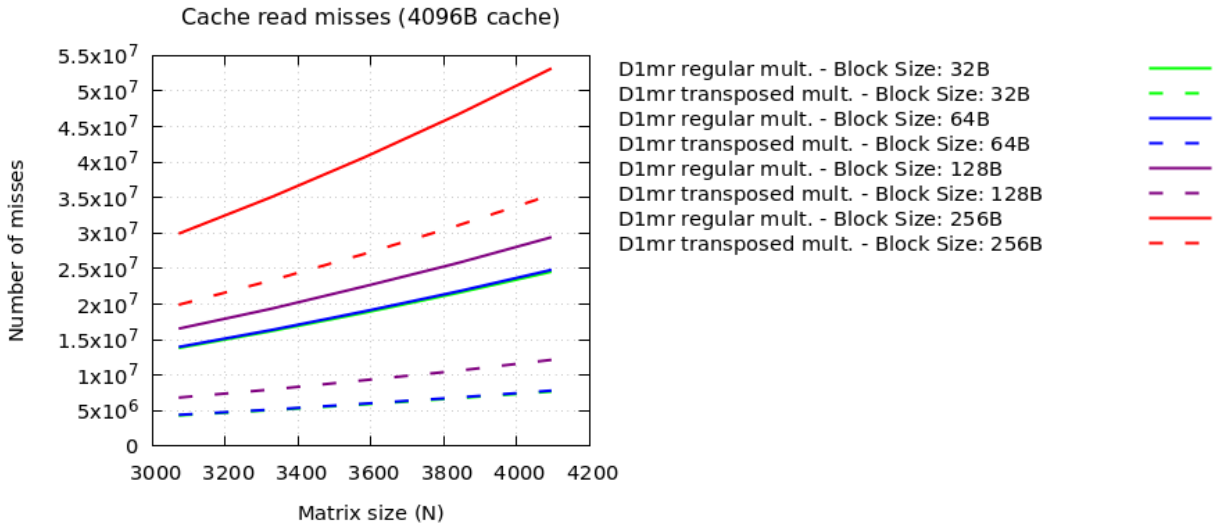

Figure 13: Cache read misses with a cache of 4096B

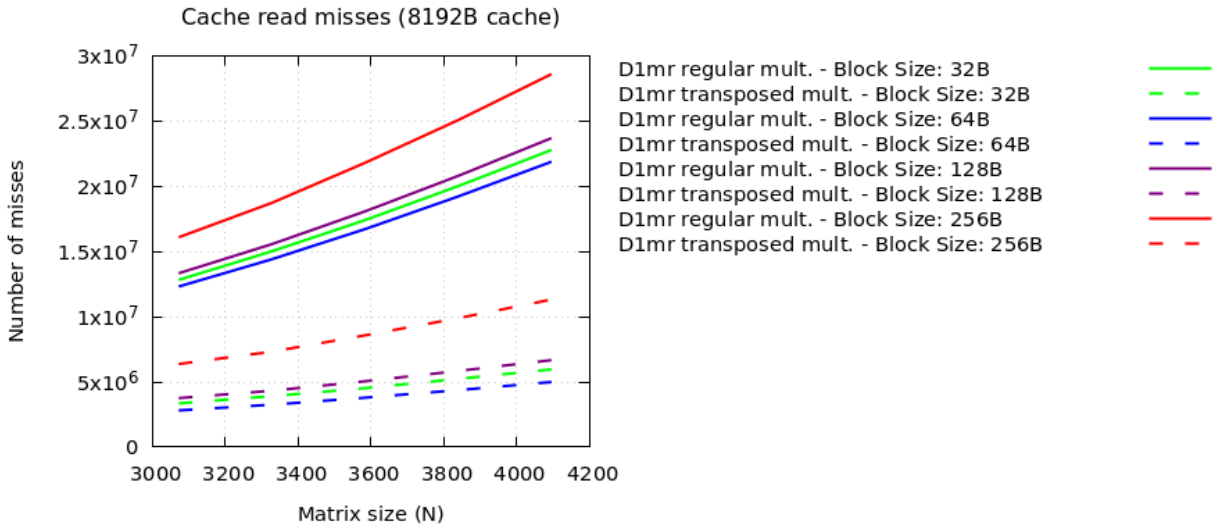

Figure 14: Cache read misses with a cache of 8192B

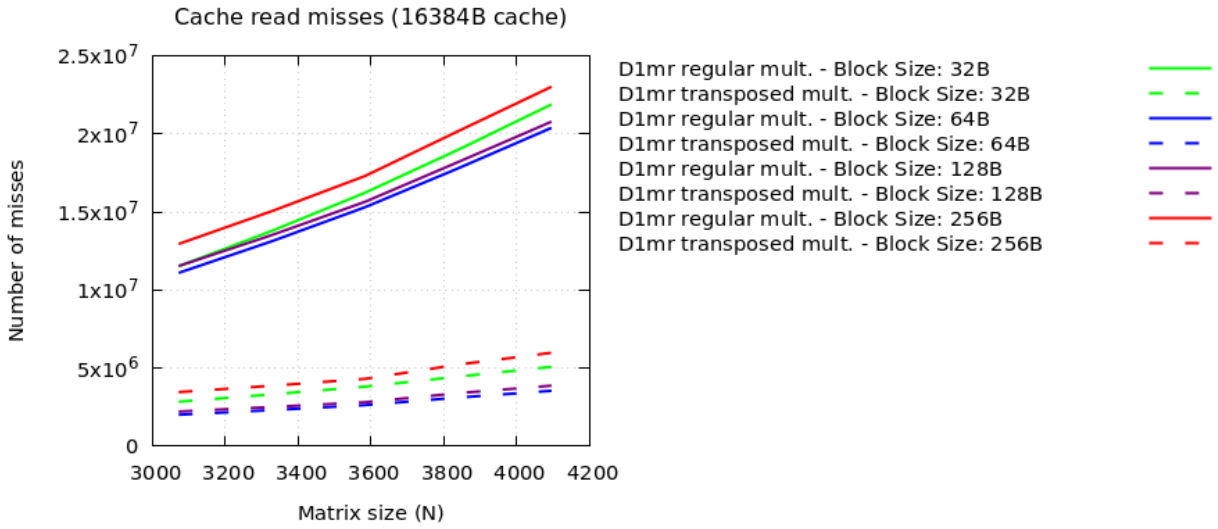

Figure 15: Cache read misses with a cache of 16384B

The first thing we notice is that as we increase the size of the cache, the lines for each block size configuration get closer (take for instace the case of a 16384B cache – figure 15). Therefore, it makes sense to think that the impact of the block size will be greater when the size of the elements of the matrix is of the order of the cache and for small caches.

The key aspect is going to be the number of lines of the cache for each configuration. In order to determine this number, we have to do the following calculation:

$$\text{Number of lines} = \frac{\text{Cache Size}}{\text{Block Size}}$$

Let's focus now in the case of a 2048B cache. If we subsitute the different values of the block size, we get these numbers of lines:

When the block is 32 bytes long, we can store 4 elements of the matrix in each block (the type used in the matrix has a size of 8 bytes). When obtaining the element $c_{1,1}$ we have to fetch the first part of row 1

Block Size	32	64	128	256
Number of lines	64	32	16	8

Table 1: Relation between block size and number of lines for a 1-way 2048B cache.

of matrix **A** and the first part of rows 1-4 of matrix **B** (we have to fecth rows 1-4 because we have only the first 4 elements of row 1 of matrix **A**). As we have 64 rows, we have no need to replace. However, let see what happens when the block is 256 bytes, we have only 6 lines in the cache. Again for calculating element $c_{1,1}$ we first fecth the first part of row 1 of matrix A; this time this first part of the row contains 32 elements. Then we start fetching the beginning of the rows 1-32 of matrix B. As we only have 8 lines, we will have to replace blocks.

So the problem in using big block sizes is that we limit the number of rows in the cache. Furthermore, it makes no sense having large blocks when doing the regular multiplication as we only need the first elements of each block. In other words, we are fetching huge blocks just to use a few bytes of them.

We could do the same analysis for the case of a 16384B cache. This time the number of lines when the block is 256B is 64, which is bigger than 32 (the number of elements that can be stored in a block). Thus, we are not getting the problem discussed above and the result does not differ much to the ones obtained with smaller values of the block size.

Now, we will look at wrie misses: Once again, we see that the number of write misses is the same for the regular and transpose multiplication as expected. The effect of the block size is similar to the one described for the read misses. The main idea is that when the cache is small and we use a large block, we reduce significantly the number of lines. Therefore, when multiplying both matrix and writing the result, there are going to be many replaces between the blocks used for calculation and the block used for writing the result. Just to ilustrate this conflict, let's recall the case of a 2048B cache with 256B blocks. In this situation, the cache has 8 lines whose content alternates between part of the rows of the operands (which produces read misses) and part of the row of the result matrix (write misses). The scarcity of lines is what provokes a high rate of misses. Nonetheless, as we increase the size of the cache, the number of lines is "big" enough even if we use a block size of 256B (figure 18).

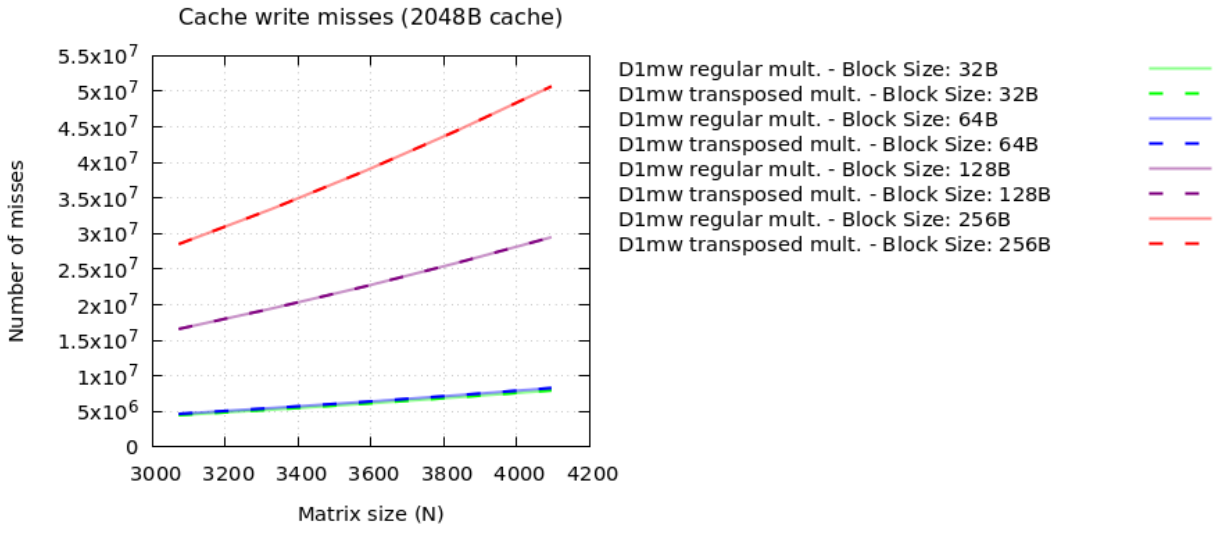

Figure 16: Cache write misses with a cache of 2048B

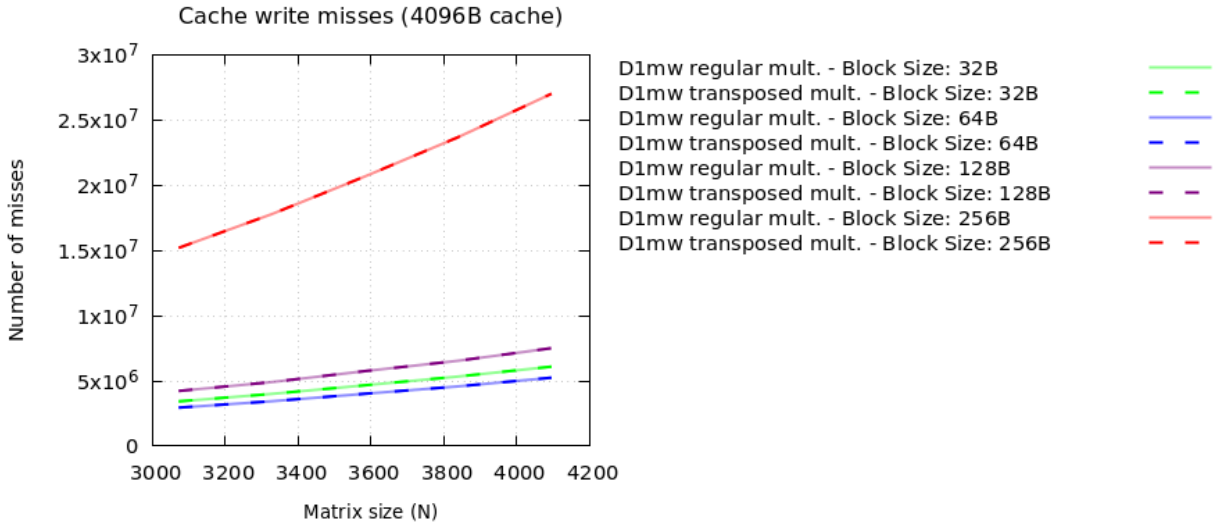

Figure 17: Cache write misses with a cache of 4096

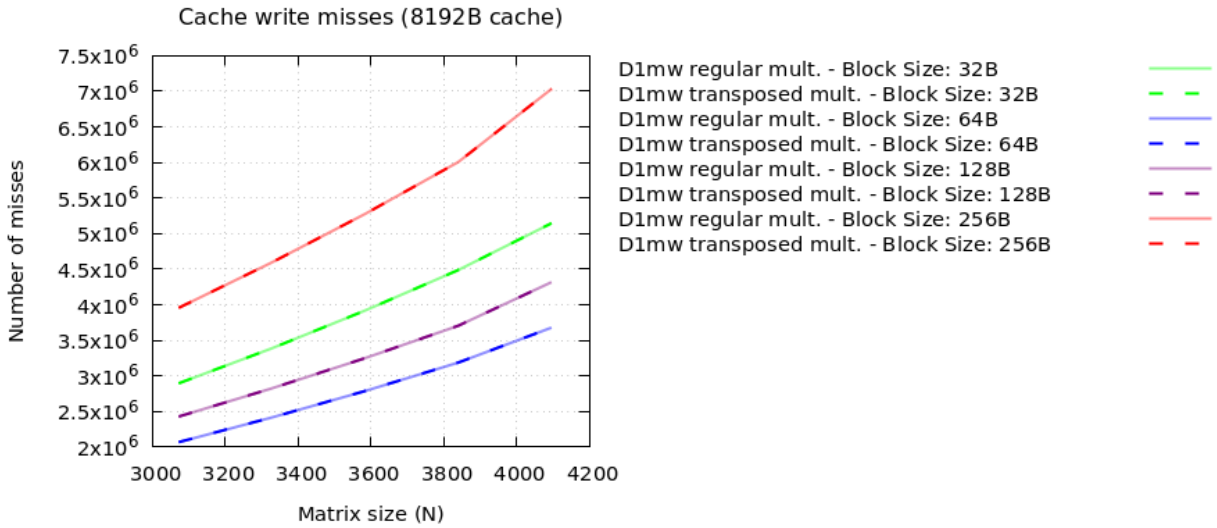

Figure 18: Cache write misses with a cache of 8192B