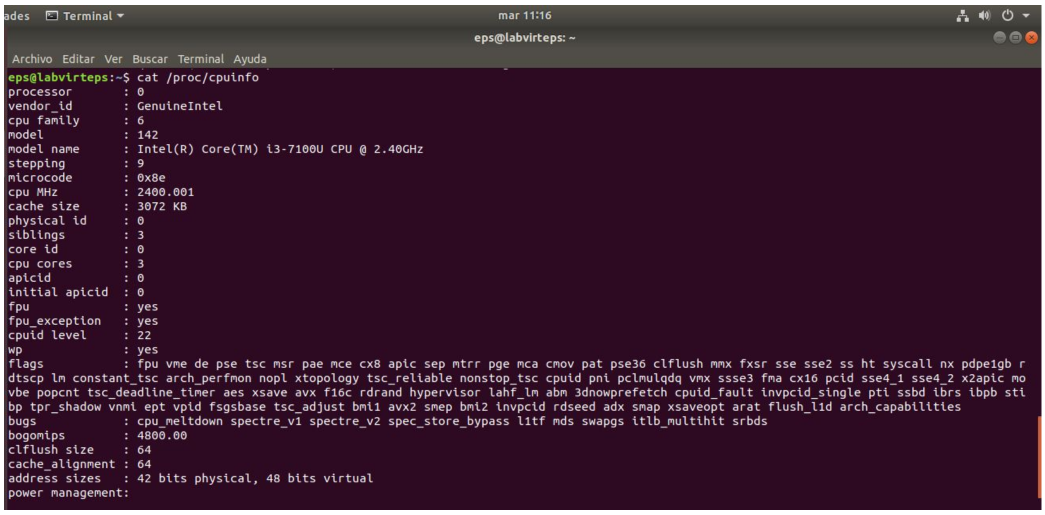
TASK 3: **CACHE AND PERFOMANCE**

Computer Architecture

S. Xiaofan Fernández & Samai García
Group 1391

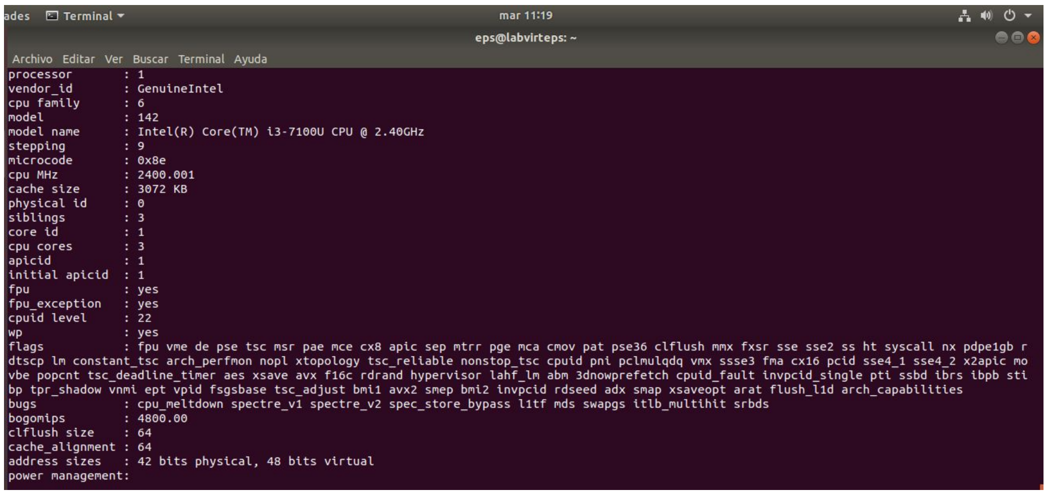
EXERCISE 0: **System Cache**

Processor 0:



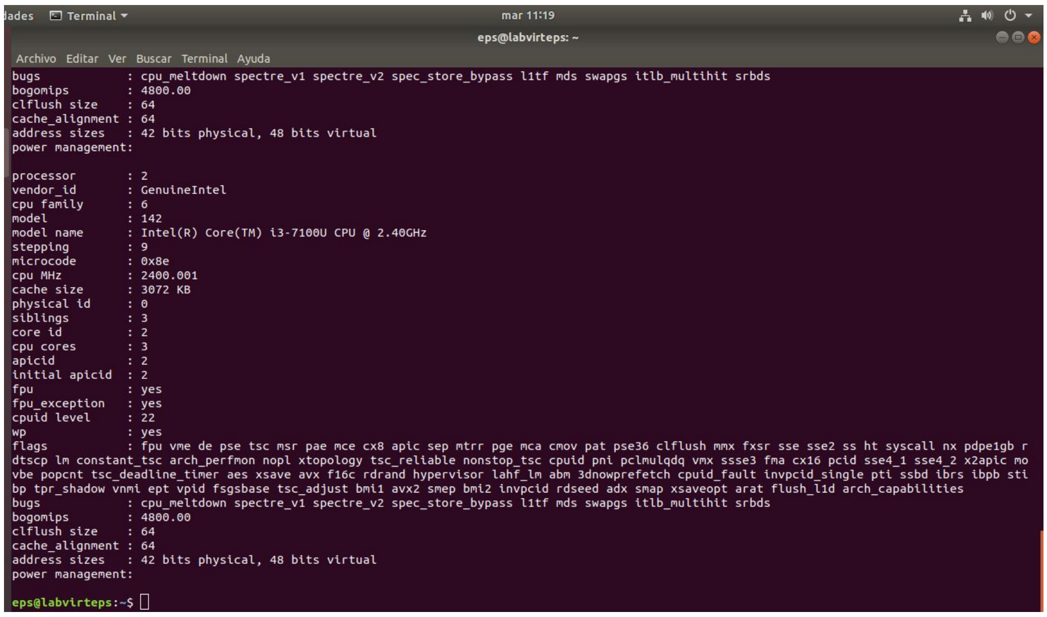
```
eps@labvirteps:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 142
model name      : Intel(R) Core(TM) i3-7100U CPU @ 2.40GHz
stepping        : 9
microcode       : 0x8e
cpu MHz         : 2400.001
cache size      : 3072 KB
physical id     : 0
siblings        : 3
core id         : 0
cpu cores       : 3
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb r
dtscp lm constant_tsc arch_perfmon nopl xtopology tsc_reliable nonstop_tsc cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic mo
vbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb sti
bp tpr_shadow vnmi ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 invpcid rdseed adx smap xsaveopt arat flush_l1d arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit srbds
bogomips        : 4800.00
clflush size    : 64
cache_alignment : 64
address sizes   : 42 bits physical, 48 bits virtual
power management:
```

Processor 1:



```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 142
model name      : Intel(R) Core(TM) i3-7100U CPU @ 2.40GHz
stepping        : 9
microcode       : 0x8e
cpu MHz         : 2400.001
cache size      : 3072 KB
physical id     : 0
siblings        : 3
core id         : 1
cpu cores       : 3
apicid          : 1
initial apicid  : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb r
dtscp lm constant_tsc arch_perfmon nopl xtopology tsc_reliable nonstop_tsc cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic mo
vbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb sti
bp tpr_shadow vnmi ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 invpcid rdseed adx smap xsaveopt arat flush_l1d arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit srbds
bogomips        : 4800.00
clflush size    : 64
cache_alignment : 64
address sizes   : 42 bits physical, 48 bits virtual
power management:
```

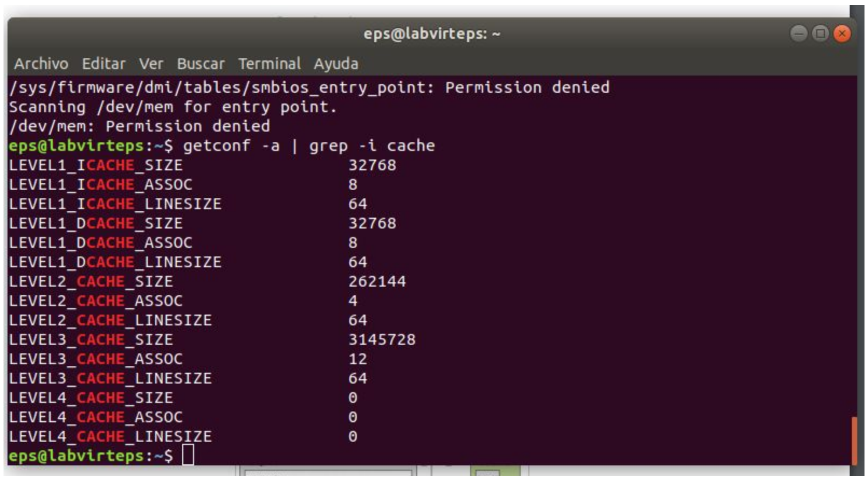
Processor 2:



```
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit srbds
bogomips        : 4800.00
clflush size    : 64
cache_alignment : 64
address sizes   : 42 bits physical, 48 bits virtual
power management:

processor       : 2
vendor_id       : GenuineIntel
cpu family      : 6
model           : 142
model name      : Intel(R) Core(TM) i3-7100U CPU @ 2.40GHz
stepping        : 9
microcode       : 0x8e
cpu MHz         : 2400.001
cache size      : 3072 KB
physical id     : 0
siblings        : 3
core id         : 2
cpu cores       : 3
apicid          : 2
initial apicid  : 2
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb r
dtscp lm constant_tsc arch_perfmon nopl xtopology tsc_reliable nonstop_tsc cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic mo
vbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb sti
bp tpr_shadow vnmi ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 invpcid rdseed adx smap xsaveopt arat flush_l1d arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit srbds
bogomips        : 4800.00
clflush size    : 64
cache_alignment : 64
address sizes   : 42 bits physical, 48 bits virtual
power management:

eps@labvirteps:~$ 
```

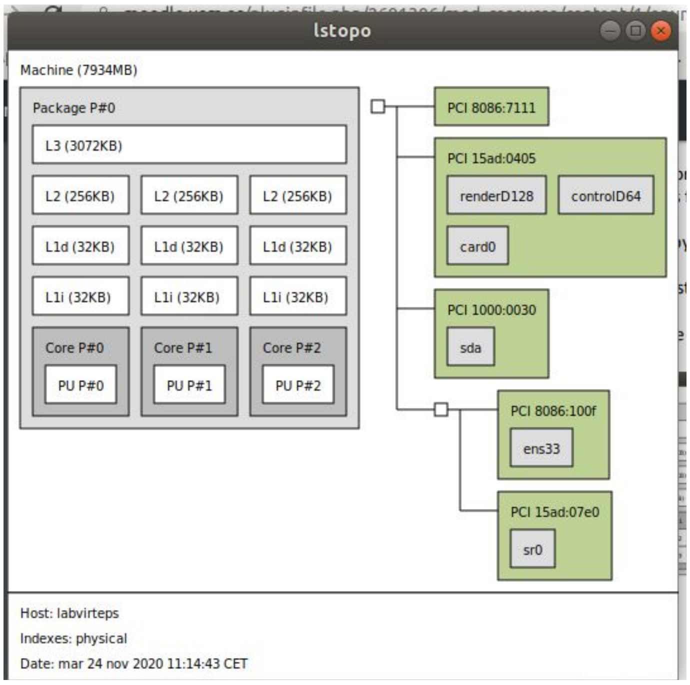


```
/sys/firmware/dmi/tables/smbios_entry_point: Permission denied
Scanning /dev/mem for entry point.
/dev/mem: Permission denied
eps@labvirteps:~$ getconf -a | grep -i cache
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC                8
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  3145728
LEVEL3_CACHE_ASSOC                 12
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
eps@labvirteps:~$ 
```

As we can see, there are four different levels for storing the data in cache memory, although the last one is out of use in our computer. The data stores in one level or other depending on the frequency of use.

Level 1 is the only one divided in two sublevels, one for data and other for instructions. This is the level with the best performance, as its content is the most frequent. For both cases, the size is 32768 Bytes, associative with 8 lines, with line size 64 bits.

Level 2 and 3, are not subdivided. They stored data less used, and as higher is the level, less faster is. We can observe that the line size remains 64 bits in the three levels. But the associativity changes, being 4 for level 2, and 12 for level 3. We observe also a great increment in cache size, being 262144 for level 2 and 3145728 for level 3.

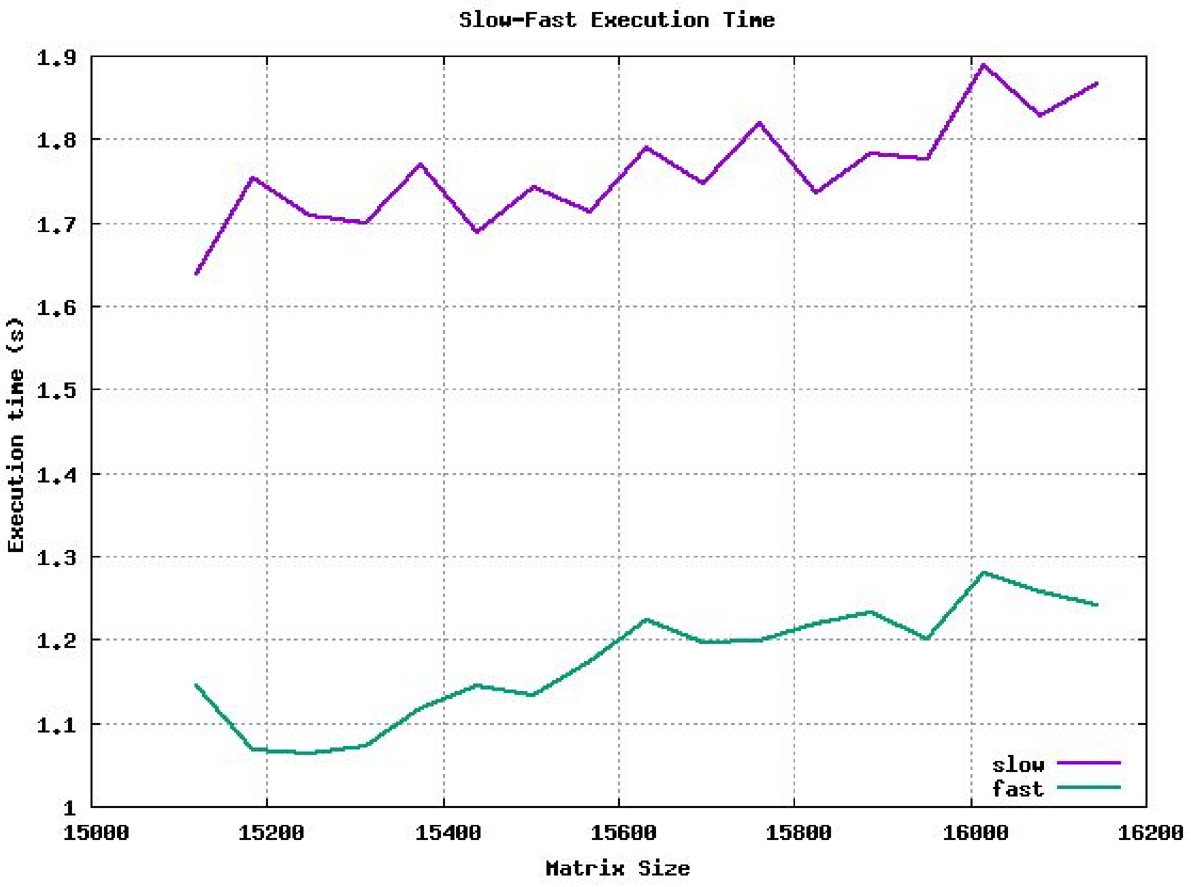


Host: labvirteps
Indexes: physical
Date: mar 24 nov 2020 11:14:43 CET

EXERCISE 1: **Cache and Performance**

In this exercise, it is necessary to execute measurements multiple times for each program and matrix size because if we execute it just once, the result would not be very representative, because the speed with which the computer executes the script depends on its state in that particular moment and on the processes it has to execute at the same time. So to get a more exact measure is better to execute it multiple times and then make the mean of the results. In our case we made 10 repetitions.

This is the result of our execution:



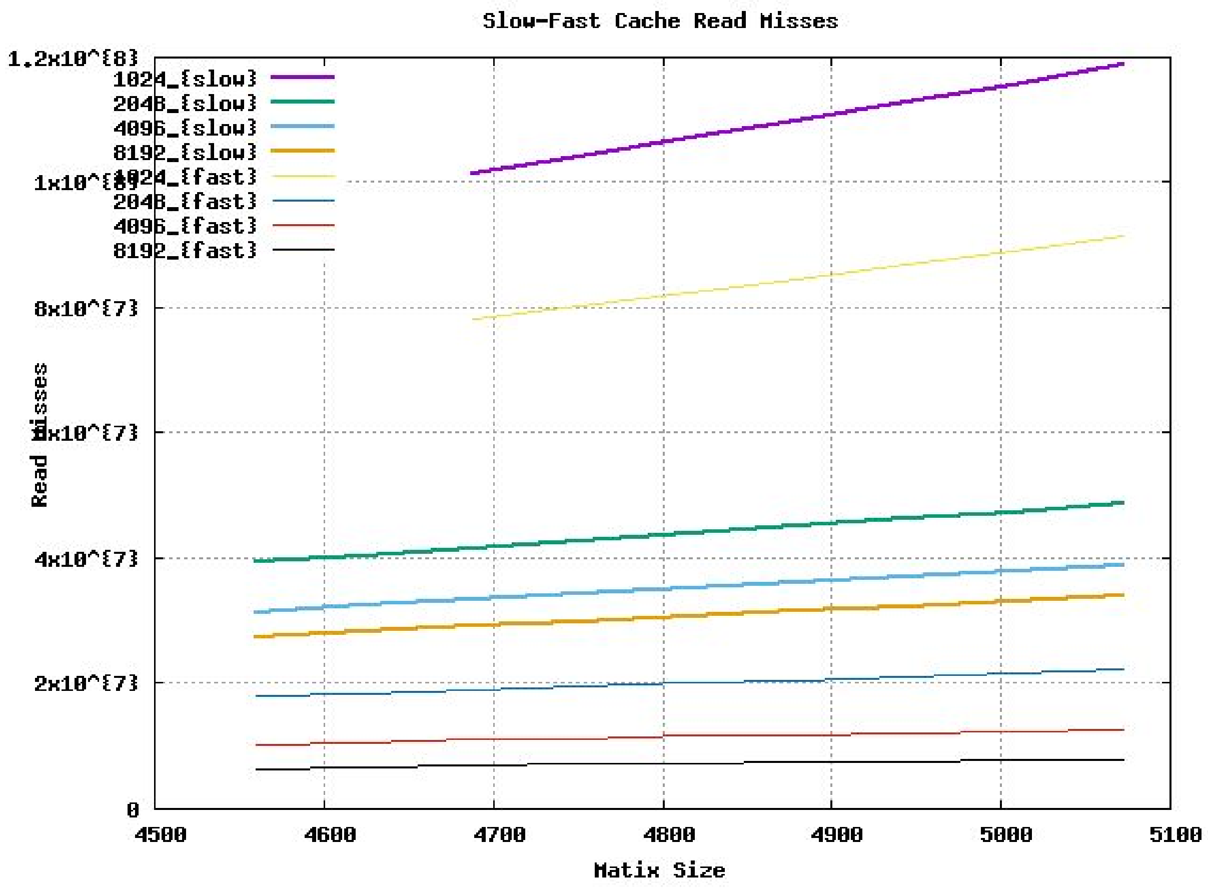
We can observe that the fast program is much faster than the slow one. While fast varies only 0.8 seconds for smaller matrices and 0.9 for larger ones, the slow program takes 1.6 and 1.9 seconds doing the same.
To get these results, we have intercalated the different matrix sizes, to avoid the cache memory to store the result of the matrix in execution, and obtaining a very small time, very far from the real one.

In the plot, we can see that the execution time of both programs differ since the beginning. That is because the matrix size starts in 15.000, a big size where you can already appreciate the difference between both programs.

The matrix is stored in memory per row, elements of the same row are stored consecutive in memory. In the case when the matrix is small, a single block can allocate all elements, so all elements of the matrix can be accessed immediately when the block is in level 1. This gives us a small execution time for both programs.
However, when the matrix size is larger, we need more blocks to store the elements. In this case, the slow program that makes the sum by columns needs to update the cache constantly with the block where the data is at that moment. Also the data are more separated in the cache, so more time is needed. But the fast program, as it sums data by rows, finds the data in the same block, it does not need to update cache as much as in slow programs, so in consequence it is faster.

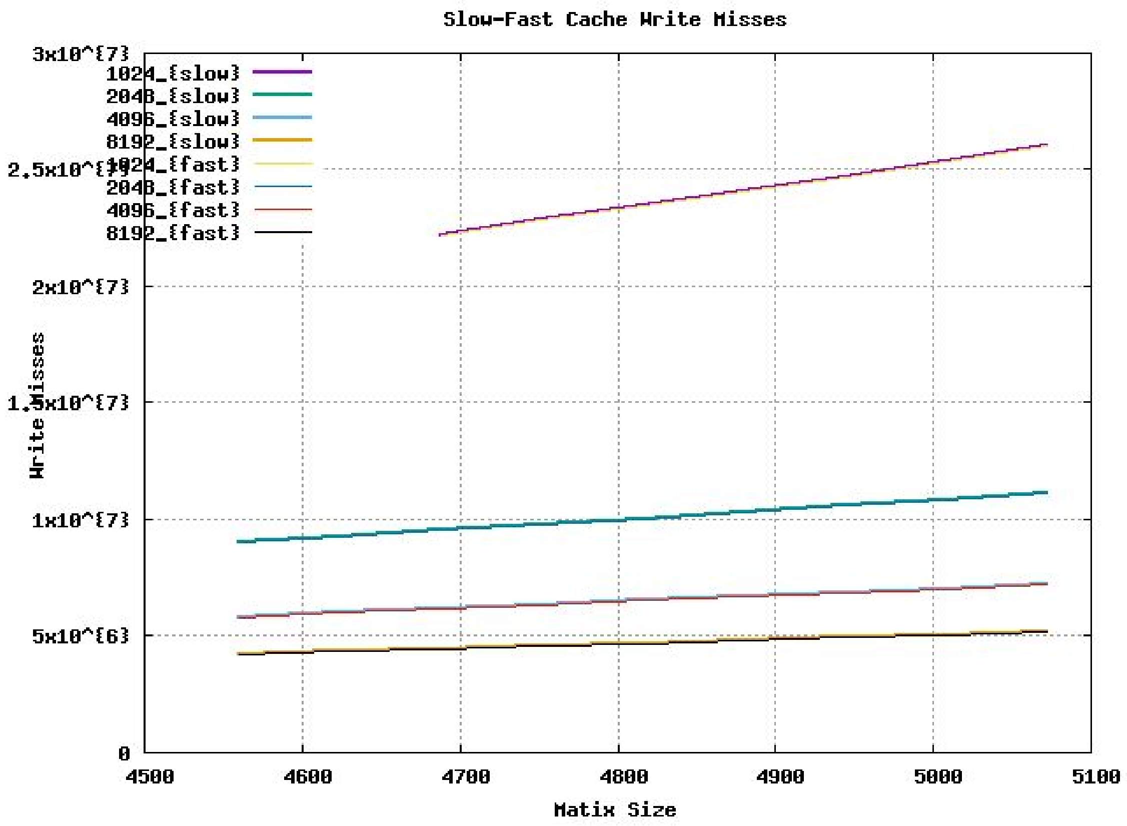
EXERCISE 2: **Cache size and Performance**

Diagram **cache_lectura**:



A cache read miss occurs when we try to read a data that is not in the cache, so that it has to update to look for it.
In this graph we can observe how the number of failures increases as the matrix size does it. This is because as bigger is the matrix, more accesses to memory are needed and so more blocks. We can also see how the fast program produces less failures than the slow program for any size of cache. This is because fast travels the matrix by rows, and the data are consecutive in the same block. But slow has to change the block every time it reads from the matrix, which requires more updates and in consequence more failures.
We also observed that when we increase the cache size, the read misses decrease. That is because if the cache is bigger, also the blocks are bigger, which increases the possibility that the data that we need is in the first level.

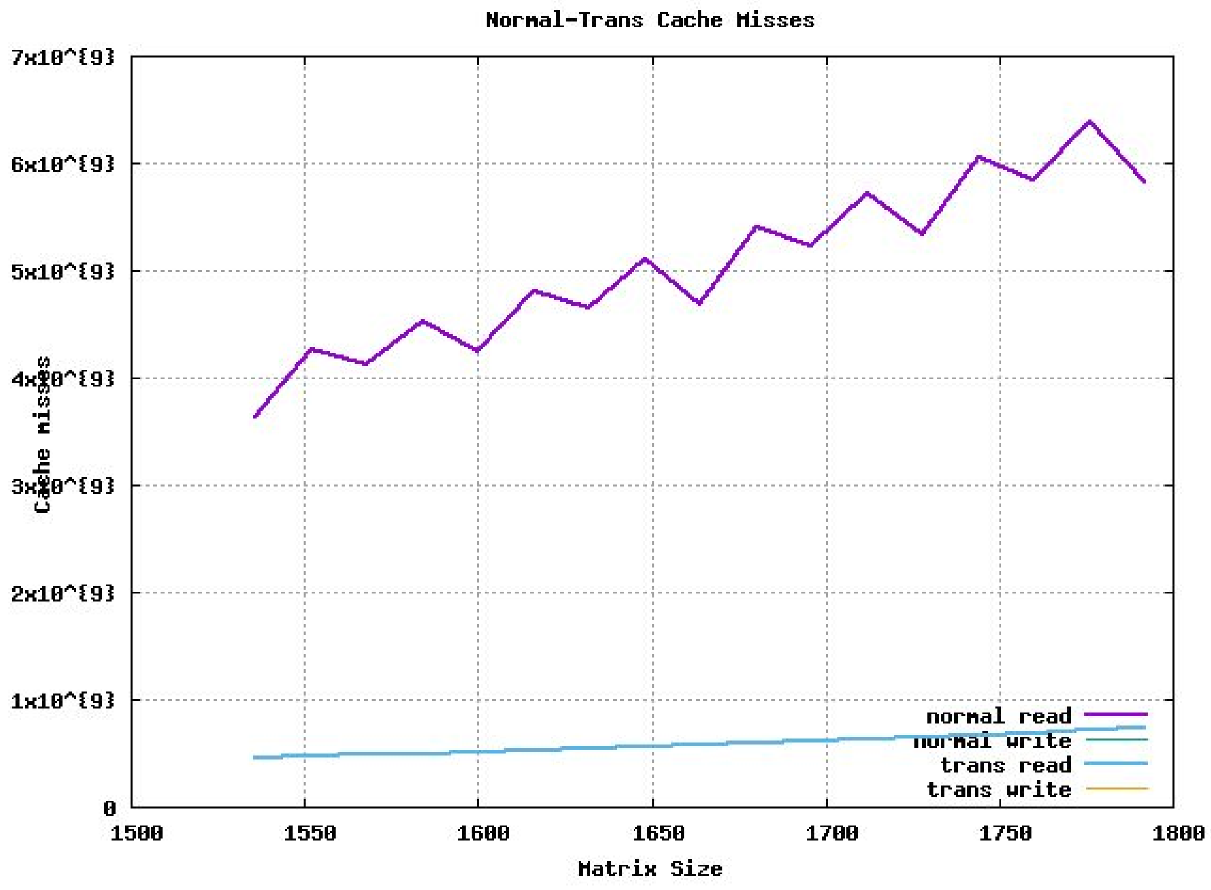
Diagram **cache_escritura**:



A write miss is produced when we try to write a data in cache with no space in memory.
What we observe in this graph is that both programs produce the same write failures.
This could be because both programs write only once for each element of the matrix, what produces the same number of failures for both programs.
Also, as in the read graph, we can see that the write failures increase as the matrix size does it, and decrease when the cache size increases.
If we compare read and write misses, we see that there are much less write failures. That is because much more readings than writings are necessary, although this also depends on the algorithm used.

EXERCISE 3: **Cache and Matrix multiplication**

Diagram **mult_cache**:

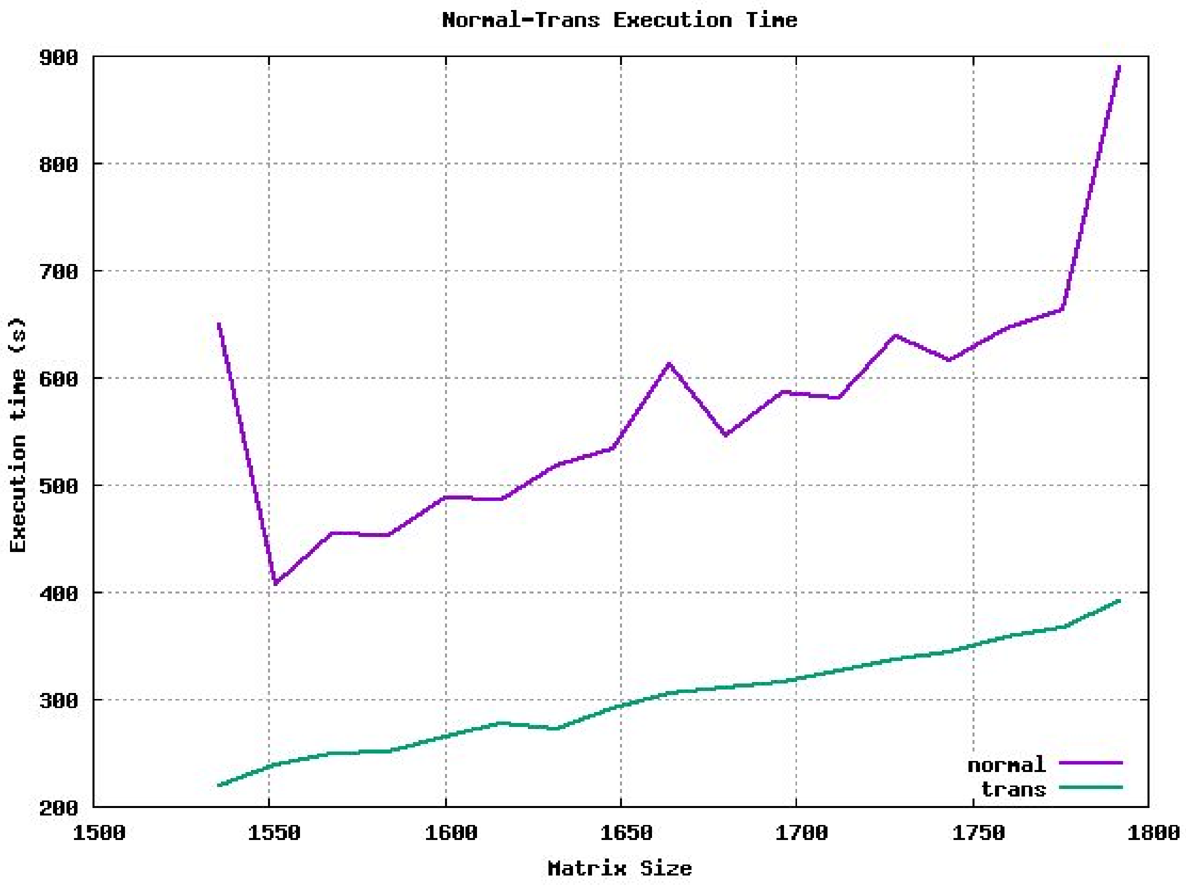


In this graph we can observe that the number of cache reading misses in the normal multiplication is bigger, as is its growth regarding the matrix size. This is because multiplication row per row is faster.

However, the cache writing misses are less. But if we observe the graph, we can see that the writing misses of the normal multiplication program is less than the transpose one. That is because in the transpose program, the matrix is stored twice, first the original one and then the transpose.

Also we admired a great difference between the reading and writing misses, because, as we will explain later, writing is done at the beginning only once when creating the matrix, but every time we use an element of the matrix, a reading is done, what is more probable to produce failures.

Diagram **mult_time**:



We can observe how the execution time increases as the matrix size does, because the number of operations is bigger.

We also see that the efficiency of the transposed matrices program is better, although it has to do the transposing of the matrix, multiplying row per row is much faster than row per column, which needs more cache updates. So transposed multiplication is faster.