PRACTICE 3

EXERCISE 0: SYSTEM CACHE.

When running “sudo cat /proc/cpuinfo” we get information about the CPU. In my case, it is an Intel i7-8700K which runs at 3.7GHz with 12 MB of cache and consists of 6 physical cores but can run 12 sub processes concurrently, so the output of the command shows as if there were 12 cores. The result is something similar to this for each of the “12 cores” (only 6 real physical):

```
processor       : 11
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
stepping        : 10
microcode       : 0x96
cpu MHz         : 1327.228
cache size      : 12288 KB
physical id     : 0
siblings        : 12
core id         : 5
cpu cores       : 6
apicid          : 11
initial apicid  : 11
fpu             : yes
fpu_exception   : yes
cpuid level     : 22
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pg
e mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts r
ep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pn
i pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16
 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer
aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault invpc
id_single pti tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adju
st bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx rdseed adx smap clflu
shopt intel_pt xsaveopt xsavec xgetbv1 xsaves ibpb ibrs stibp dtherm
ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp
bugs            : cpu_meltdown spectre_v1 spectre_v2
bogomips        : 7392.00
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

The command “sudo dmidecode” shows a huge amount of information about the computer’s hardware but when run as “sudo dmidecode --type cache” we obtain the result shown in the next image from where we can deduce several things:

```
pedro@pedro-MS-7B58:~$ sudo dmidecode --type cache
# dmidecode 3.1
Getting SMBIOS data from sysfs.
SMBIOS 2.8 present.

Handle 0x003F, DMI type 7, 27 bytes
Cache Information
        Socket Designation: L1 Cache
        Configuration: Enabled, Not Socketed, Level 1
        Operational Mode: Write Back
        Location: Internal
        Installed Size: 0 kB
        Maximum Size: 0 kB
        Supported SRAM Types:
                Synchronous
        Installed SRAM Type: Synchronous
        Speed: Unknown
        Error Correction Type: Parity
        System Type: Unified
        Associativity: 8-way Set-associative

Handle 0x0040, DMI type 7, 27 bytes
Cache Information
        Socket Designation: L2 Cache
        Configuration: Enabled, Not Socketed, Level 2
        Operational Mode: Write Back
        Location: Internal
        Installed Size: 0 kB
        Maximum Size: 0 kB
        Supported SRAM Types:
                Synchronous
        Installed SRAM Type: Synchronous
        Speed: Unknown
        Error Correction Type: Single-bit ECC
        System Type: Unified
        Associativity: 4-way Set-associative

Handle 0x0041, DMI type 7, 27 bytes
Cache Information
        Socket Designation: L3 Cache
        Configuration: Enabled, Not Socketed, Level 3
        Operational Mode: Write Back
        Location: Internal
        Installed Size: 0 kB
        Maximum Size: 0 kB
        Supported SRAM Types:
                Synchronous
        Installed SRAM Type: Synchronous
        Speed: Unknown
        Error Correction Type: Multi-bit ECC
        System Type: Unified
        Associativity: 16-way Set-associative
```

- We have 3 levels of cache.
- All of them use Write Back.
- The size is unknown for this program.
- L1 is an 8-way set associative.
- L2 is a 4-way set associative.
- L3 is an 16-way set associative.

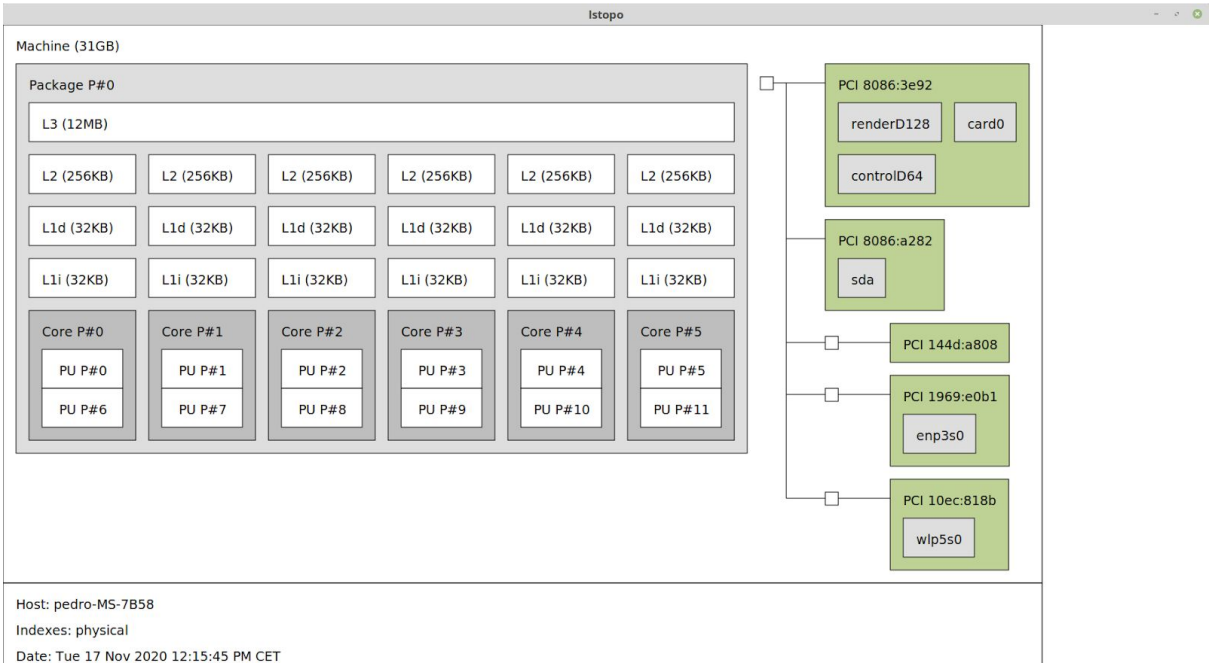
The command “sudo getconf -a | grep -i cache” gives us more information about each cache’s size, from where we can find out (apart from what we already know):

- There are 2 level one caches, one for instructions and another for data.
- Level 2 and 3 are unified (only one cache for data and instructions).
- Both L1 caches are 32KB.
- L2 cache is 256 KB.
- L3 cache is 12 MB.
- There is no L4 cache.

```
pedro@pedro-MS-7B58:~$ sudo 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                  12582912
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC                 0
LEVEL4_CACHE_LINESIZE              0
```

Finally, when we run “sudo lstopo”, we get a picture representing our processor’s physical architecture from where we can deduce that:

- We have 6 physical cores, each of them can emulate 2 cores as shown as PU P#i, Pu P·6+i, i = 0,1,2,3,4,5.
- Each of them has a level 1 cache for instructions and another L1 cache for data of 32 KB each, and a level 2 cache of 256KB.
- The level 3 cache is common for all the cores, it is shared and it is 12 MB.
- The system has 31GB of main memory or RAM.



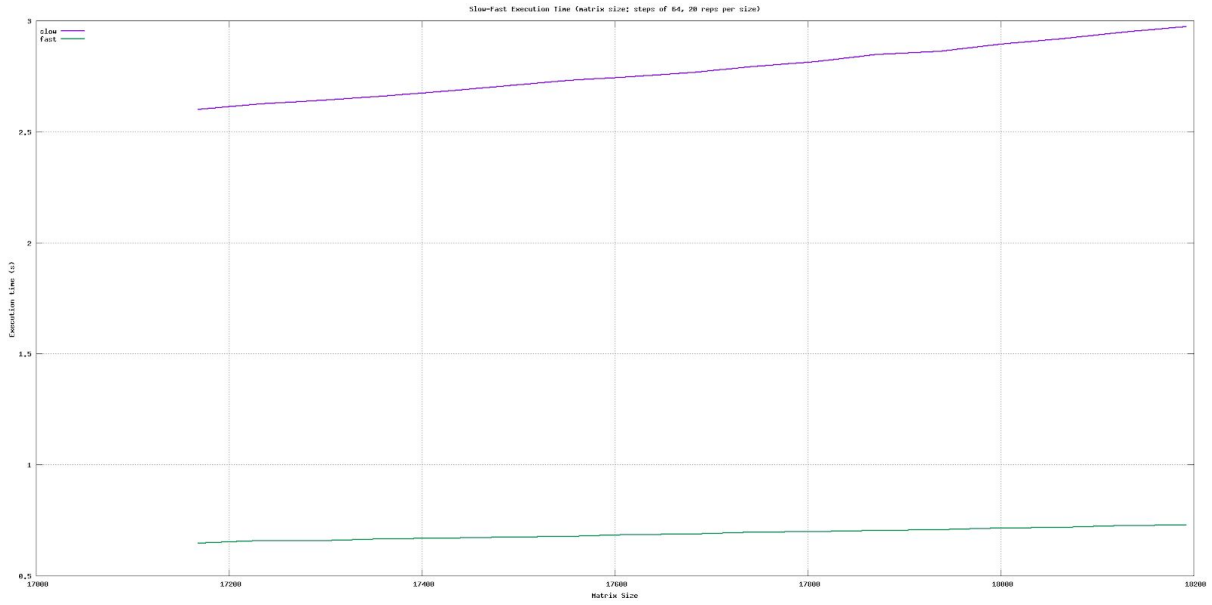
EXERCISE 1: CACHE AND PERFORMANCE.

1. Execution results were obtained in the cluster, with the execution of the script slow_fast_time_cluster.sh which sends as a job to the cluster the real script slow_fast_time.sh which performs the requested tasks. The results are saved in the file slow_fast_time.png (and shown below) and slow_fast_time.dat in the ex1_results directory.

2. As explained in the statement of the practice, each program could obtain different results because floating-point operations are not commutative, even though using the same seed and the same size, the result should be the same for all executions. Also, if the processor is busy at some point because other processes are being executed, the execution time of a certain size could be abnormal, so running it several times and normalising significantly reduces the chances that big peaks in the final plot happen.

3. All the execution results can be seen in the ex1_results directory.

4.



5. In order to generate the results we have modified the given script so that it executes the addition of the content of the matrixes for the desired sizes which we are asked to, also executing them in an interleaved manner so that peaks do not occur (because of data already loaded in cache). We also have to execute each program with each matrix's size several times so that dispersion in the results is reduced. The script slow_fast_time.sh is executed by the script slow_fast_time_cluster.sh which sends the job to the cluster, where the final execution and the resulting plot is obtained. The output of the program is saved in slow_fast_time.out, as we cannot see the screen of the cluster.

The script slow_fast_time.sh runs for the requested matrix sizes and for an adjustable number of repetitions for each size, the programs slow and fast following the next pattern: (slow N, slow N + paso, fast N, fast N +paso) repsPerN times, (slow N + 2*paso, slow N + 3*paso, fast N + 2*paso, fast 3*paso) repsPerN times, … . We chose this pattern in order to try to reduce big peaks and valleys from happening. After that, the results are normalized, saved in file, shown in the terminal and finally plotted.

The matrices are stored in memory by rows, as we can deduce from the line code “matrix[i] = &array[i*size];” (arqo3.c) where the array an n*n array stored in contiguous memory (if possible) cells, as it has been allocated with a malloc.

That is why when matrix size is small, the fast and slow program give similar results, as in both cases the complete array containing the matrix can be in the cache, whereas when the size is increased, if we access the elements one row each time there will be much more cache misses as the complete array (or matrix) is not in the cache.

In conclusion, the trend is clear, adding the matrix in a way that does not take advantage of the cache (therefore causing lots of cache misses) makes the program run much slower than if the access to the elements of the matrix is done in the same way the elements are stored in the cache.

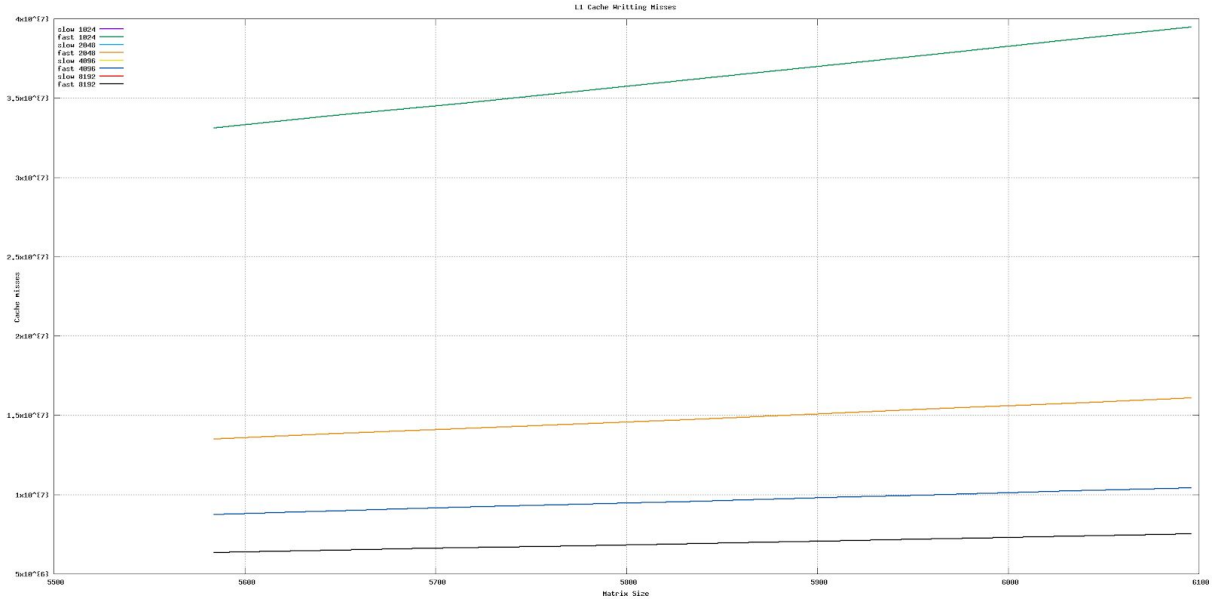
EXERCISE 2: CACHE SIZE AND PERFORMANCE.

1. The script cachegrind.sh performs the requested tasks. As we have obtained all our results in the cluster, an auxiliary script cachegrind_cluster.sh has been created in order to submit the job of the cachegrind.sh script to the cluster. The output of the program is saved in cachegrind.out as we cannot see the cluster's screen.

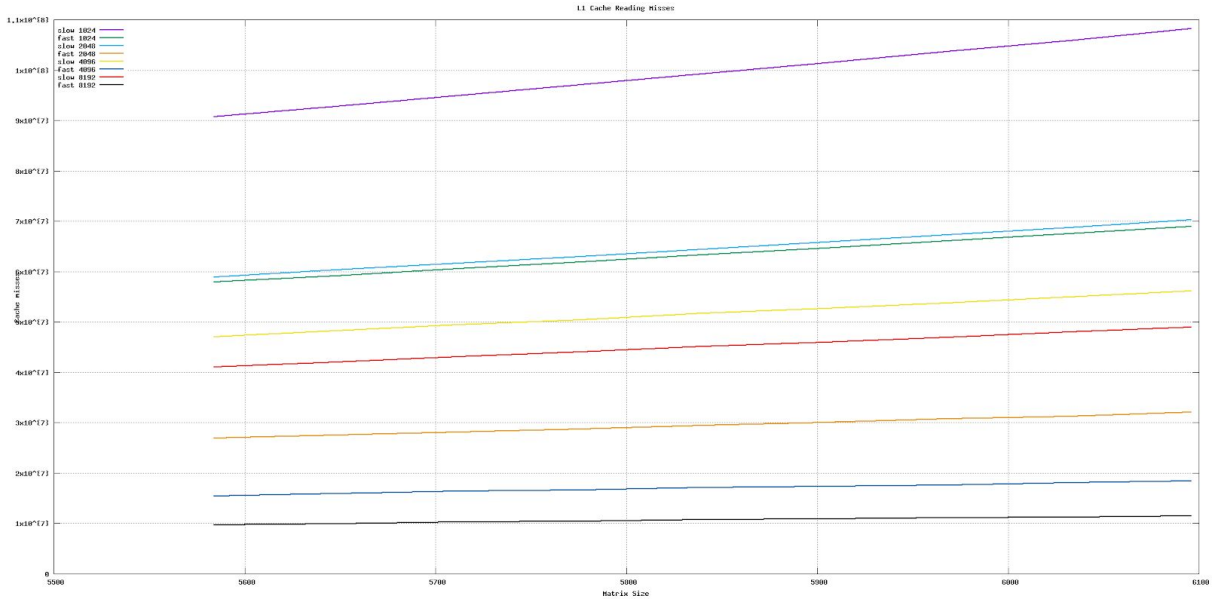
The cachegrind.sh script executes for every L1 cache size given in a list, cachegrind in both slow and fast programs for every requested size of the matrix, showing the results in the terminal (in our case in the file cachegrind.out as we run it on the cluster) and saving them in the corresponding data files with a fixed format, so that it can finally plot the results using gnuplot.

2. All results are saved in the ex2_results directory.

3. Even though we are requested to use the cache as the abscissa axis, we thought it was more illustrative to show the matrix size in the abscissa axis and draw different plots for each cache size:



Concerning the L1 Cache Writing Misses plot, we have obtained what could have hypothesized before seeing the results, that both, slow and fast programs make the same number of writing cache misses when their L1 cache is of the same size. This is logical because the only writing done by slow and fast programs is when generating the matrices that they are going to sum, and if the size is the same this would give the same writing misses for both programs. The other important observation is that as we increase the cache size, there are obviously less writing cache misses because the bigger the cache, more data fits in it, so writing misses are reduced.



Concerning the L1 Cache Reading Misses plot, we can observe the huge differences in the cache reading misses between the slow and fast program. The hypothesis form exercise 1 to explain why the fast program runs faster than the slow is now confirmed with the data from the plot: the slow program always makes much more reading misses than the fast program, more or less three time more reading cache misses, being the difference maximum for large sizes of the cache and, even still notorious, minimum for smaller cache sizes. This must be because in the smaller caches, rows do not fit entirely in the cache so the difference between the fast and slow programs are reduced (but still 2 times faster the fast program).

Obviously, as we increase the cache size, reading misses are reduced in both programs for the same reasons given above. The difference in the number of cache misses for both programs is the same as we explained in the previous exercise: the fast program accesses the elements of the matrix in a favorable way to the cache (accessing data as it is stored in the cache), whereas the slow program doesn't.

4. So, we can conclude that bigger sizes of the cache gives always better results for this type of program (as a higher percentage of the matrix elements fit in the cache), and that the way programs access the data affects reading cache misses significantly, while not modifying the number of writing misses.

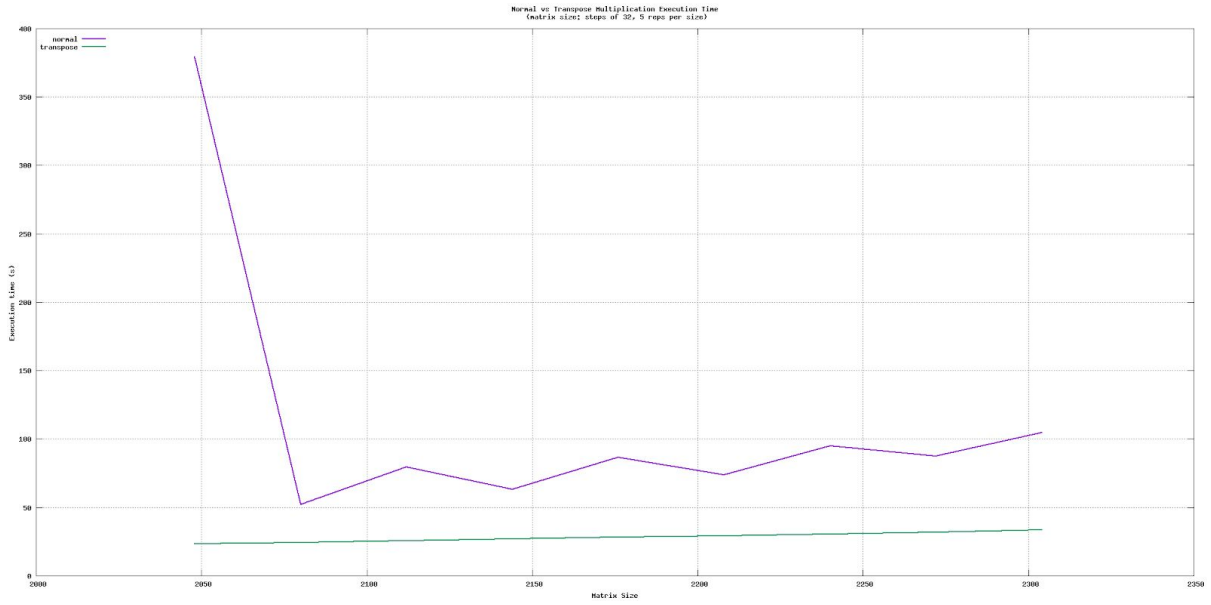
EXERCISE 3: CACHE AND MATRIX MULTIPLICATION.

1. The script mult.sh performs the requested tasks. As we have obtained all our results in the cluster, an auxiliary script mult_cluster.sh has been created in order to submit the job of the mult.sh script to the cluster. The output of the program is saved in mult.out as we cannot see the cluster's screen.

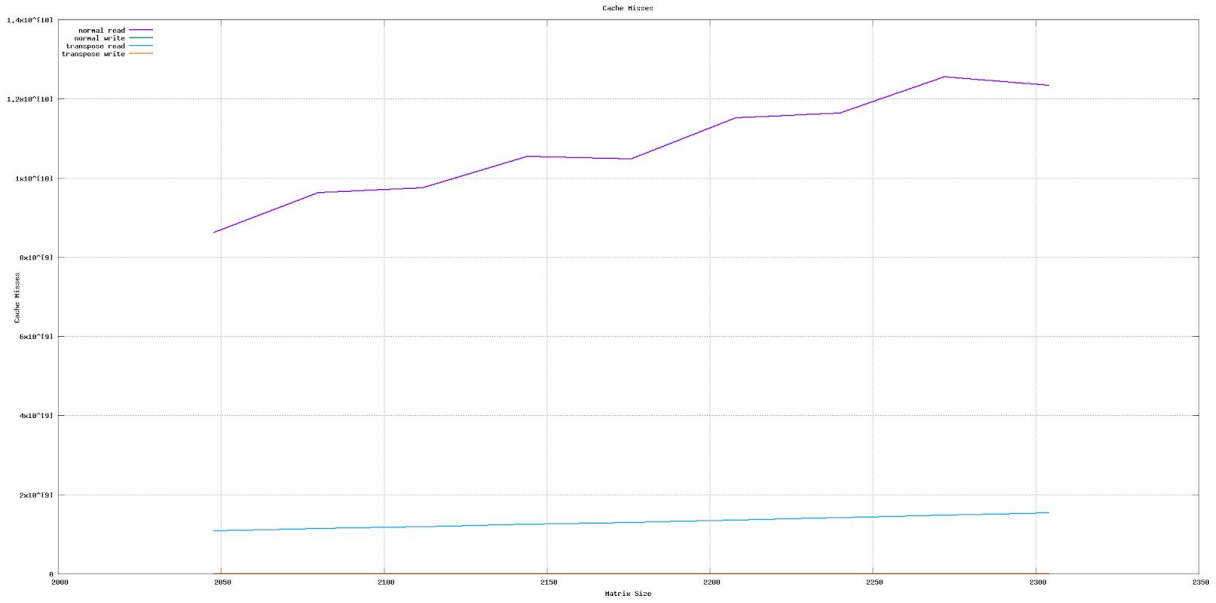
The mult.sh script performs the same work as the requested in the previous two exercises for slow and fast, but now running the programs mult (matrix multiplication) and tmult (transposed multiplication of matrices). There is a clear correspondence between the mult and slow program (as both access their needed data in a disfavorable way to the cache) and the tmult and fast program (as both access their needed data in a favorable way to the cache), easily observed in the results, and everything commented in the two previous exercises can be commented here just changing the words slow by mult and fast by tmult.

2 and 3. All results are saved in the ex3_results directory.

4.



Without taking into account the result of the first size, this plot shows the same trend as the one observed in exercise 1, being the normal multiplication program (mult) the slow program in that exercise and the transpose multiplication program (tmult) the fast program. The reason is that the tmult program (as the fast program) takes into account how to access the matrix for the multiplication, transposing the second matrix so that the data is stored in the way it is going to be accessed to do the multiplication. As the algorithm for the multiplication uses at each step one row of the first matrix (A) and one column of the second matrix (B), if we transpose B before multiplying we have to access rows of A and rows of B, which favors the work of the cache as matrices are stored row by row, therefore creating much less reading cache misses (as shown in the next plot) and therefore running much faster, more or less twice as fast.



In this plot we can observe several things. First, writing cache misses are for, both programs, nearly nothing compared to the reading cache misses. Second, the number of reading misses is much higher in the case of the mult (normal multiplication program) as explained previously, because it accesses data of the second matrix as the slow program, column by column (in the opposite way matrix is stored in cache), instead of row by row access accomplished if we first transpose the matrix. Finally, the plot of the cache misses is perfectly linear in the case of the transposed matrix multiplication, whereas the normal multiplication has certain peaks and valleys. This may be caused by the divisibility of the size of the matrix by the size of the cache or a multiple of a power of 2.

5. If we had executed the program with smaller matrices we would have probably seen that the difference in the execution time and the number of cache misses would be smaller as a higher percentage of the data stored in the matrices would fit in the fixed size cache.

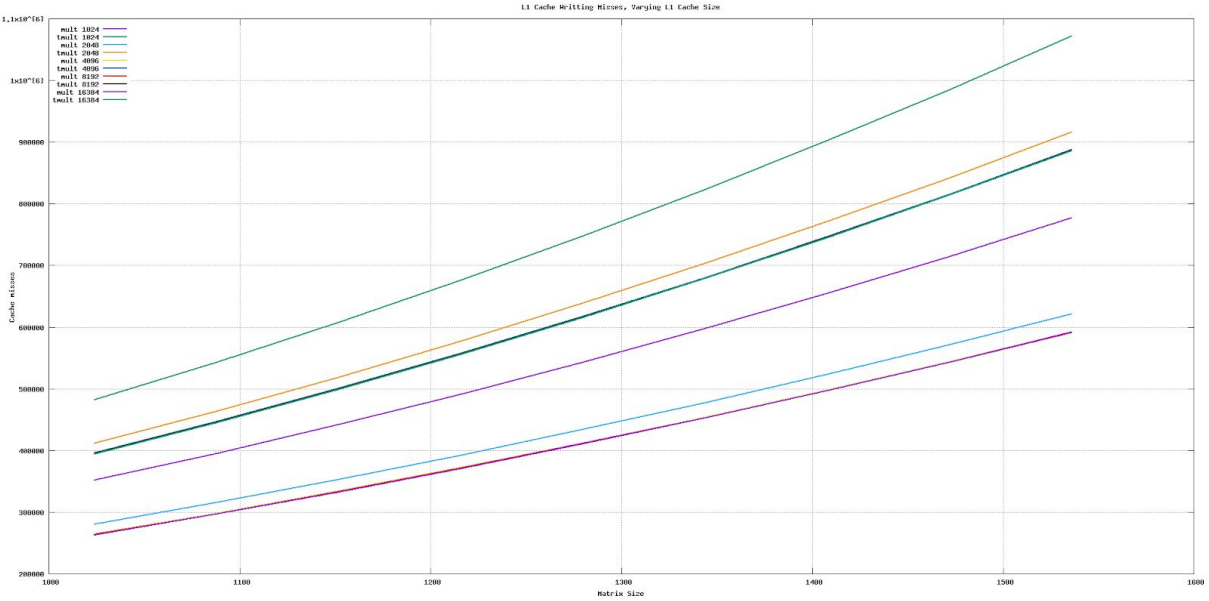
For the rest of the results everything from exercise 1 and 2 apply here as explained.

EXERCISE 4: CACHE PARAMETERS IN MATRIX MULTIPLICATION.

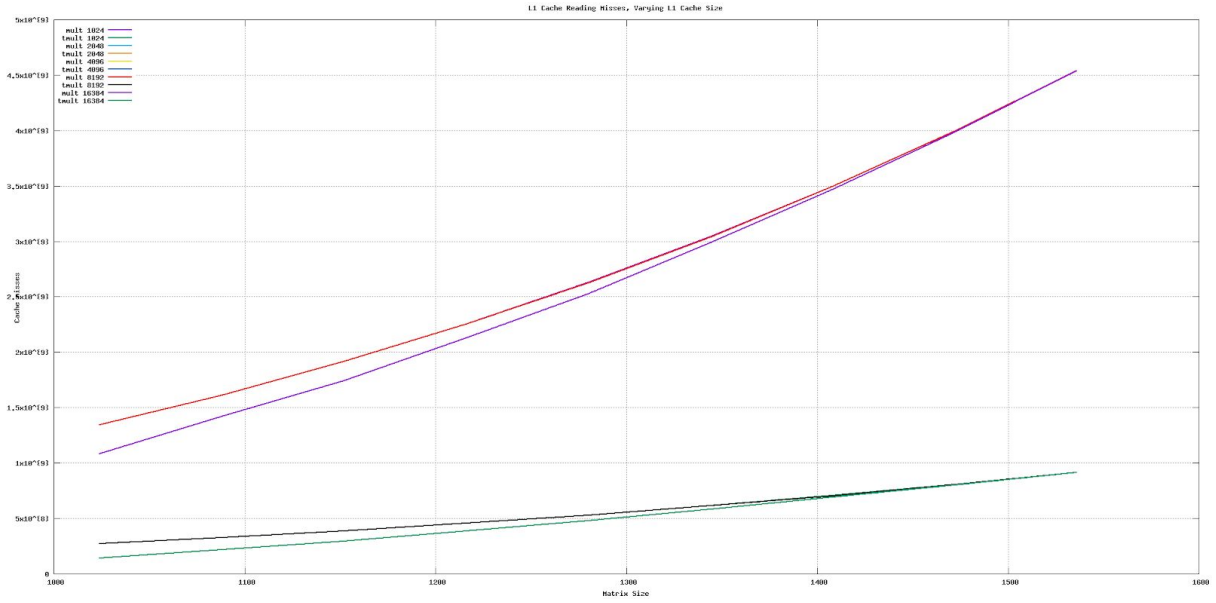
In order to see what happens when we vary the different parameters suggested in the heading of the practice, we programmed the script ex4.sh, and the corresponding ex4_cluster.sh script to submit it as a job to the cluster (where we have run all of our tests). All the resulting plots are saved in the ex4_results directory.

The script ex4.sh is very similar to the cachegrind.sh script from exercise 2, as in order to vary the caches sizes, ways and linesize. We created different lists with the different things we wanted to vary in order to study the performance when these numbers changed. We chose to vary the L1 size, LL size, L1 number of ways, LL number of ways and the linesize or blocksize (basically everything we are allowed to vary with cachegrind). For each thing we wanted to vary we created a list with the values we wanted it to vary, and proceeded exactly as in exercise 2 but running the program mult (matrix multiplication) instead of slow and tmult (transposed matrix multiplication) instead of fast program. For every parameter we varied we created a plot showing the number of cache writing misses for each dimension of the matrix and another plot showing the number of cache reading misses:

When varying the L1 cache size we tried the following values: 1KB, 2KB, 4KB, 8KB and 16KB. The results were:

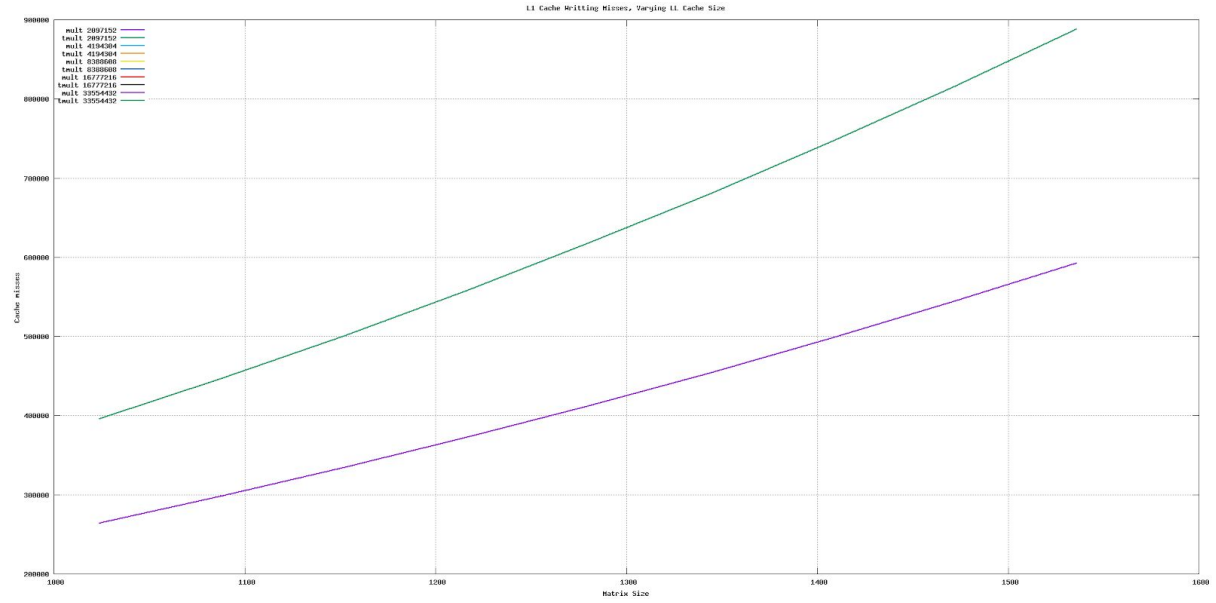


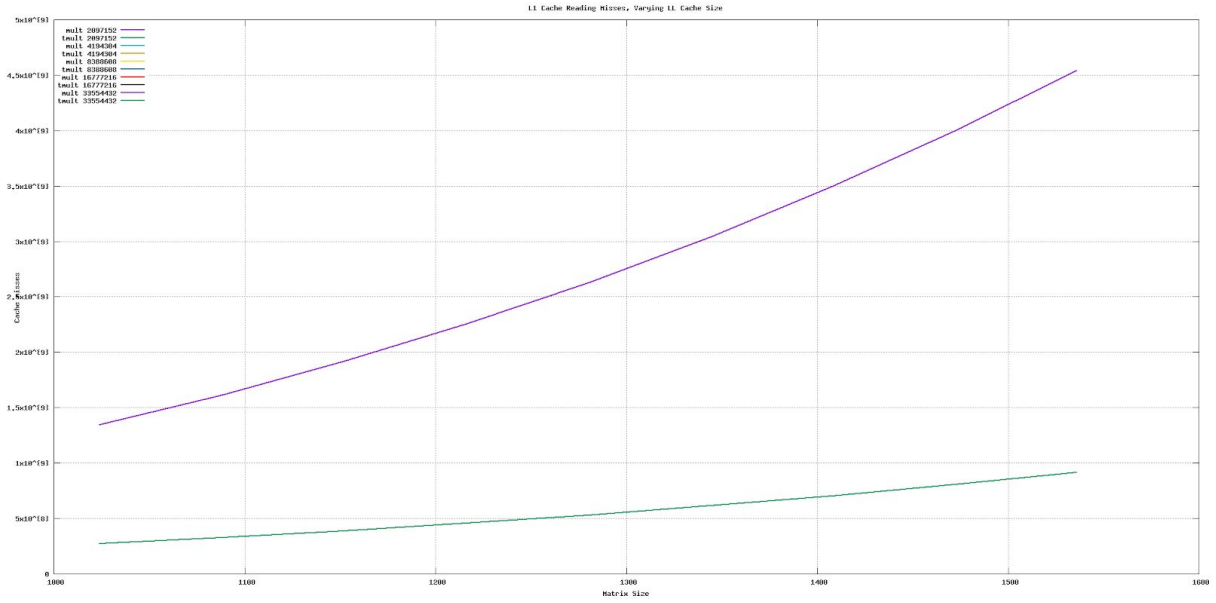
Here we can see that the tmult program produces about 1.5 more writing misses than mult program, for every matrix size and cache size. This is logical as the tmult program creates a new matrix in order to store the transposed matrix, and as the the normal mult program allocates 2 matrices (A and B) of size N and the tmult 3 (A, B and B transposed) the it is normal to obtain the 3/2 = 1.5 ratio. For each program we can see that obviously small cache sizes produce more cache misses but from 4KB on the results are more or less the same for each cache size.



When analyzing the reading misses we can see that the mult program makes much more cache reading misses than the tmult program, and not only that, but we can see that for bigger matrices these differences will be much bigger, finally diverging. It seems that for small matrices the L1 cache size matters, giving the 16 KB cache better results in both programs, but will soon converge with the rest of L1 cache sizes as we increase the matrix size. This suggests us that the sum of all matrix element sizes is much bigger than the actual cache sizes so we should probably try bigger L1 sizes to see if this is true.

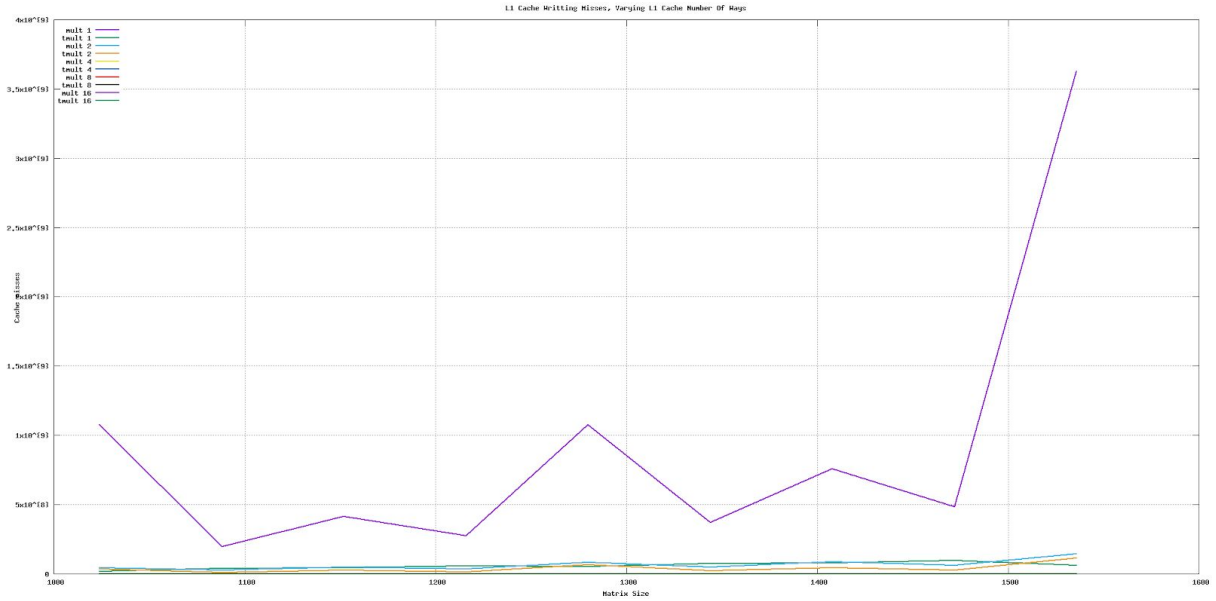
When varying the LL cache size we tried the following values: 2MB, 4MB, 8MB, 16MB and 32MB. The results were:



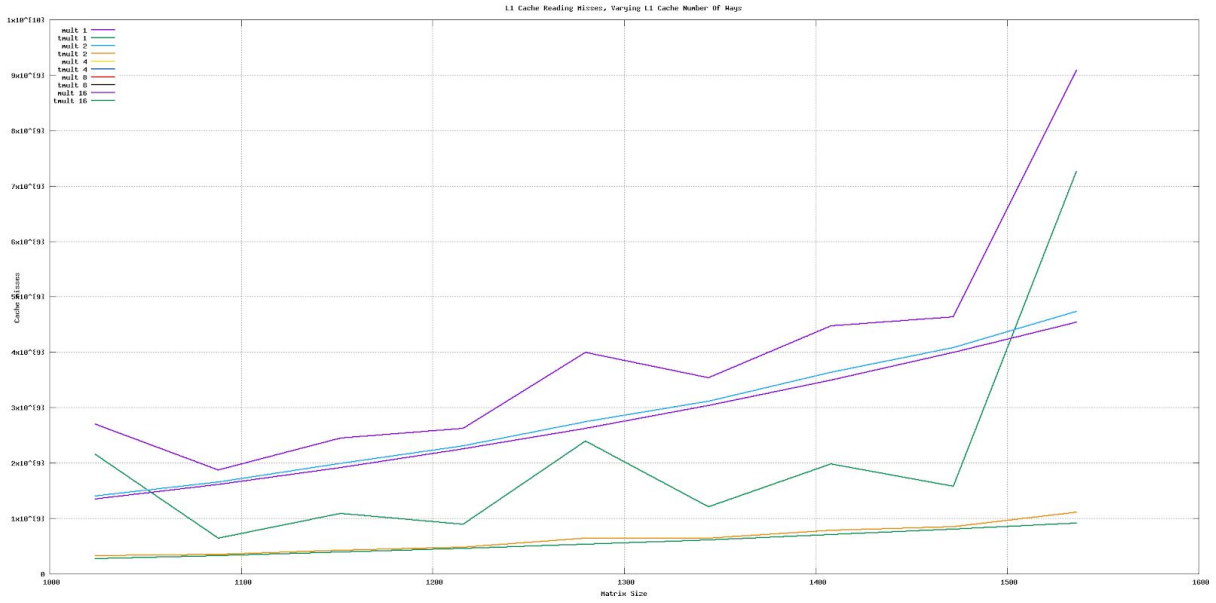


We can observe that the size of the large cache does not affect in any case the number of cache reading or writing misses, the only thing that affects this number is the program executed. As in the case of the L1 cache varying the size, the tmult program generates less cache misses because of the way in which elements are accessed (taking into account matrices are stored by rows and not columns) and there are approximately 1.5 more cache writing misses for the tmult program for the same reason as explained before (allocating 3 matrices instead of 2).

When varying the L1 cache ways we tried the following values: 1, 2, 4, 8, 16 ways. The results were:

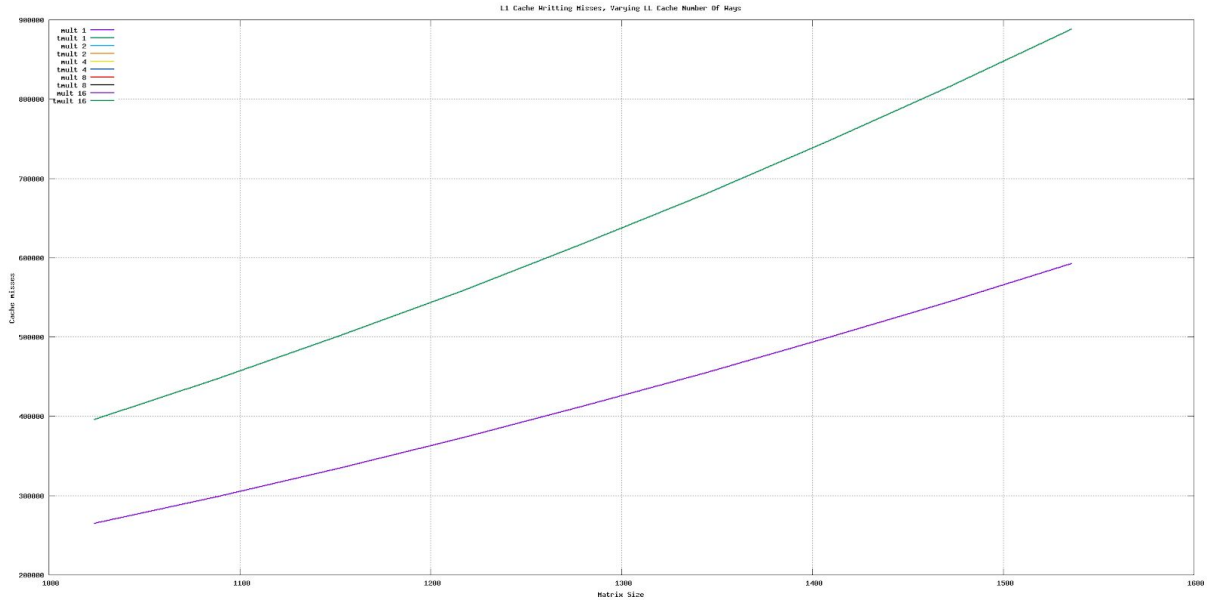


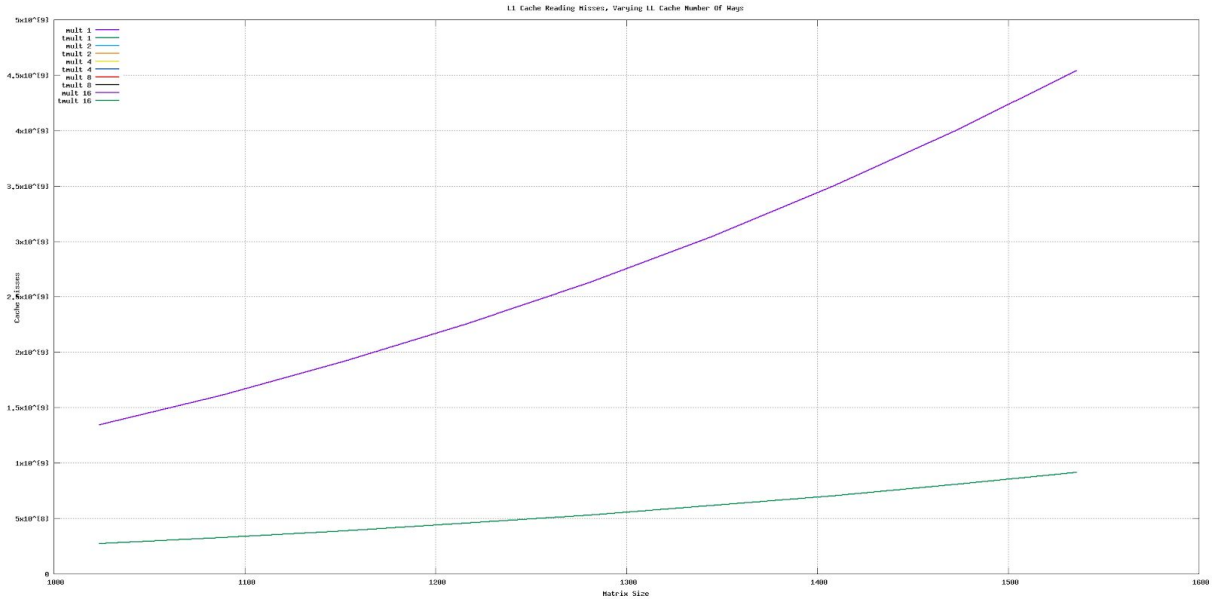
Here there obviously happened some problems when executing so results are not clear. But taking a close look we can deduce that more ways imply less writing cache misses.



In this case the results were strange for some reason and we could not fix it so results do not let us conclude anything.

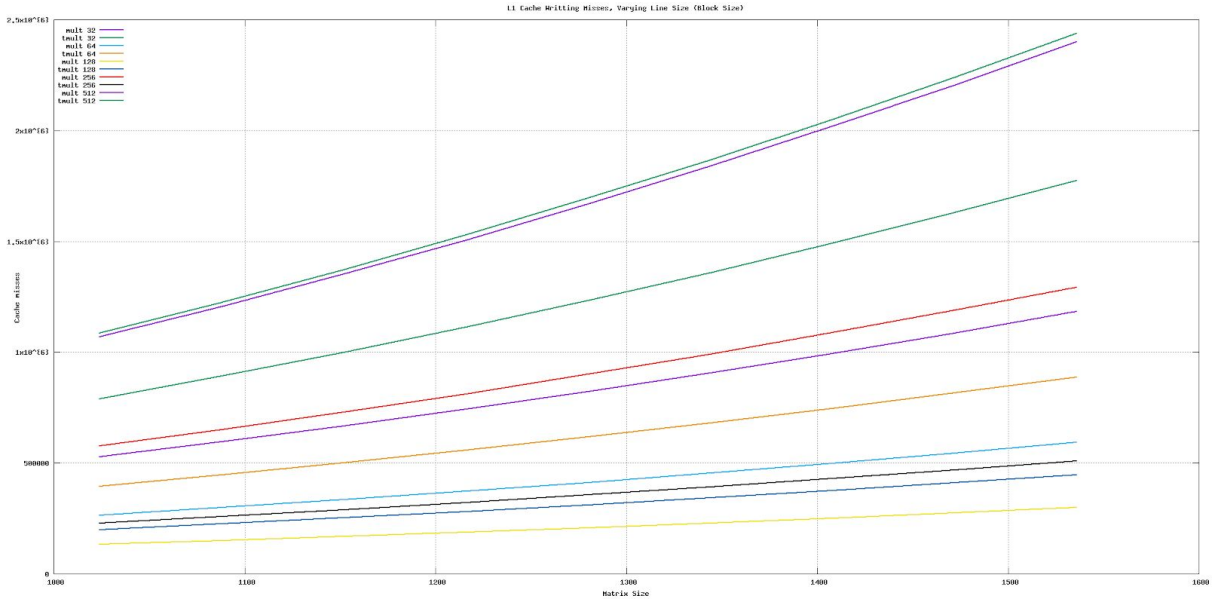
When varying the LL cache ways we tried the following values: 1, 2, 4, 8, 16 ways. The results were:



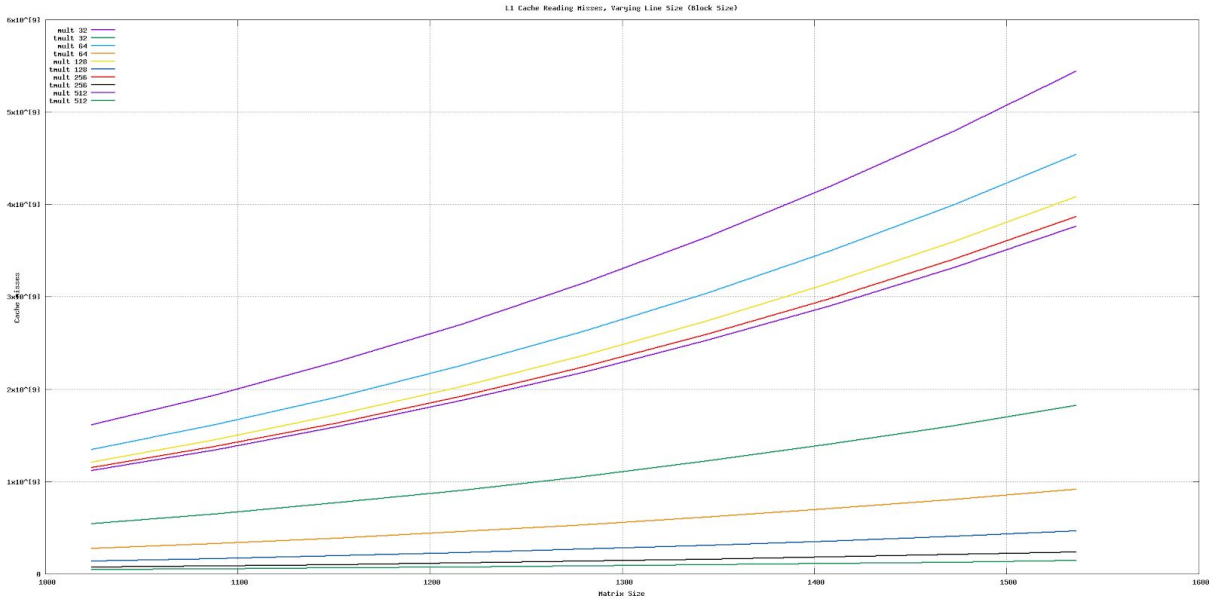


We can observe that the number of ways of the large cache does not affect in any case the number of cache reading or writing misses, the only thing that affects this number is the program. As in the case of the LL cache varying the size, the tmult program generates less cache misses because of the way in which elements are accessed (taking into account matrices are stored by rows and not columns) and there are approximately 1.5 more cache writing misses for the tmult program for the same reason as explained before (allocating 3 matrices instead of 2).

When varying the linesize we tried the following values: 32, 64, 128, 256 and 512. The results were:



The results we obtained are interesting. When linesize is 128 the number of writing misses seems to be optimal for both programs mult and tmult and the worst is line size 512 for both. Then the rest of the values are in between. We can see, as always, that the tmult program always causes more writing cache misses than the mult.



For the reading cache misses the tendency is clear: the mult program always generates much more reading cache misses and the bigger the linesize is the less number of cache reading misses there are, but this last difference is smaller and smaller as we increase the linesize, being nearly the same for linesize 256 and 512.

There is an obvious tradeoff between the number of cache reading misses and the number of cache writing misses, but the optimal value for the linesize after this study is 128 as it is optimal for the writing misses and the reading misses for this size is similar to the optimal one, the biggest possible.