Rodrigo Juez y José Manuel Freire

ARQO - P3

November 26th 2020

EXERCISE 0

We created a script to obtain this information on both the AMD machine that we used in the cluster (we always used AMD and **not** intel) and our personal computers. (We used qsub so the information is not from the "entry" computer rather than the one we used to run the scripts).

Cluster:

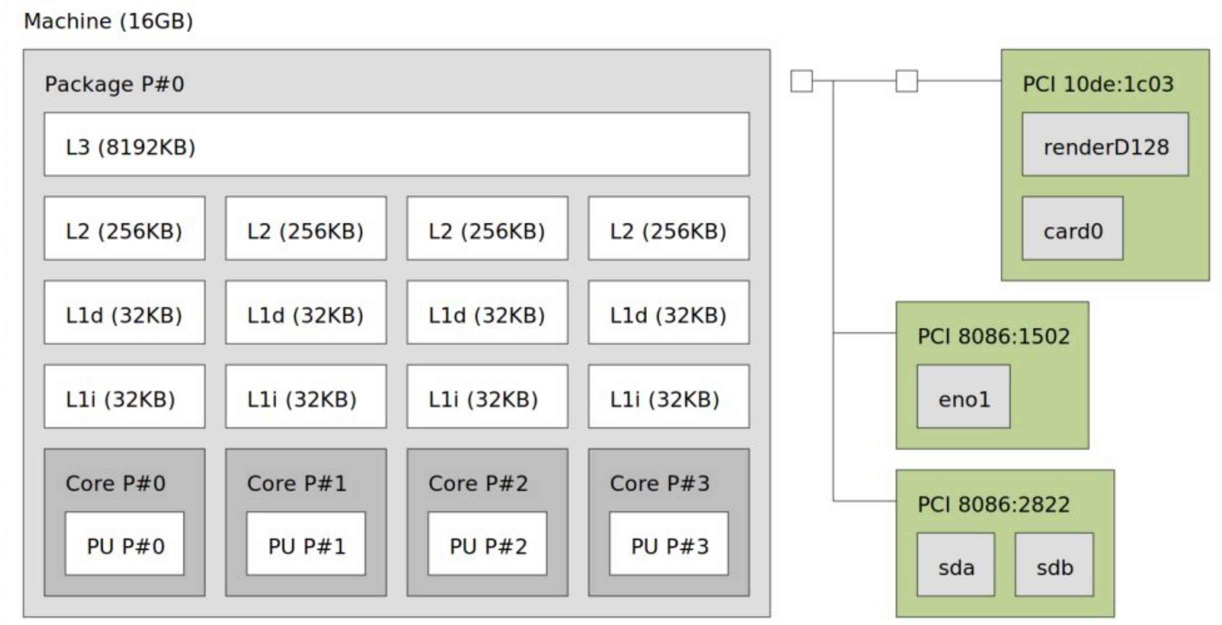
Through cat /proc/info we learnt that this computer from the cluster has 12 cores formed up from two Six-Core AMD Opteron(tm) Processor 2427.

And through getconf -a we found that the L1 cache is **not** unified and both data and instructions have 64KB bytes of capacity, 2 ways and 64 bytes line size. The level 2 cache is unified and has 16 ways, 524288 bytes of capacity and a line size of 64 bytes. The level 3 cache also is unified and has a capacity of 6MB and a 48 ways, with a word size of 64 bytes.

Personal Computer:

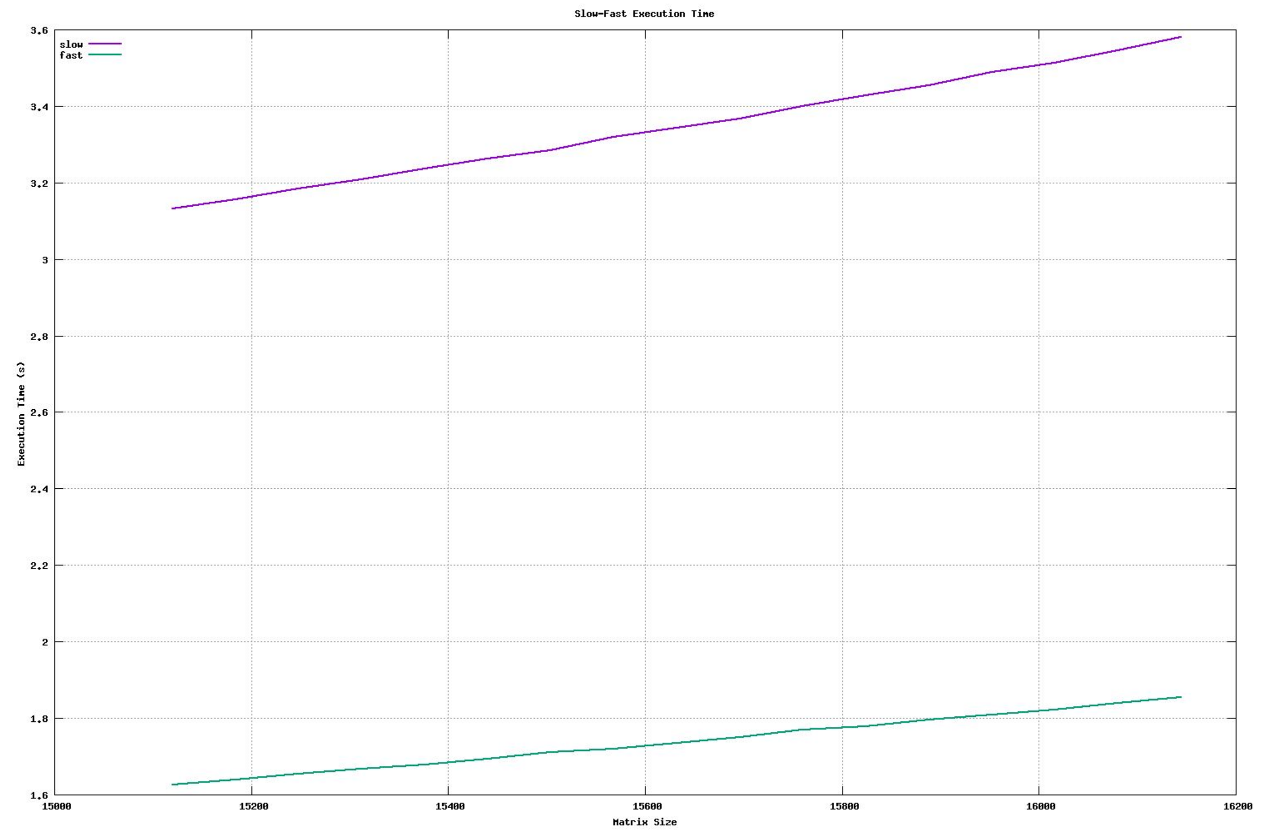
On our personal computer we learnt that we have 4 core Intel(R) Xeon(R) CPU E3-1225 V2 @ 3.20GHz

Using getconf -a and lstopo again we found that the L1 cache is **not** unified and both data and instructions cache have the same characteristics: 32KB of capacity, 16 ways and 64 bytes line size. The level 2 cache is also unified and has 8 ways, 256KB of capacity and a line size of 64 bytes. The level 3 cache again is unified, has a capacity of 8MB and a 16 ways, with a word size of 64 bytes.



EXERCISE 1

In this exercise we had to compare the execution time of two programs that added two matrixes. The difference between them was that one (fast.c) used the memory by rows, while the other (slow.c) accessed the memory jumping between the columns of the matrix, so it needed to reload the block in cache all the time.

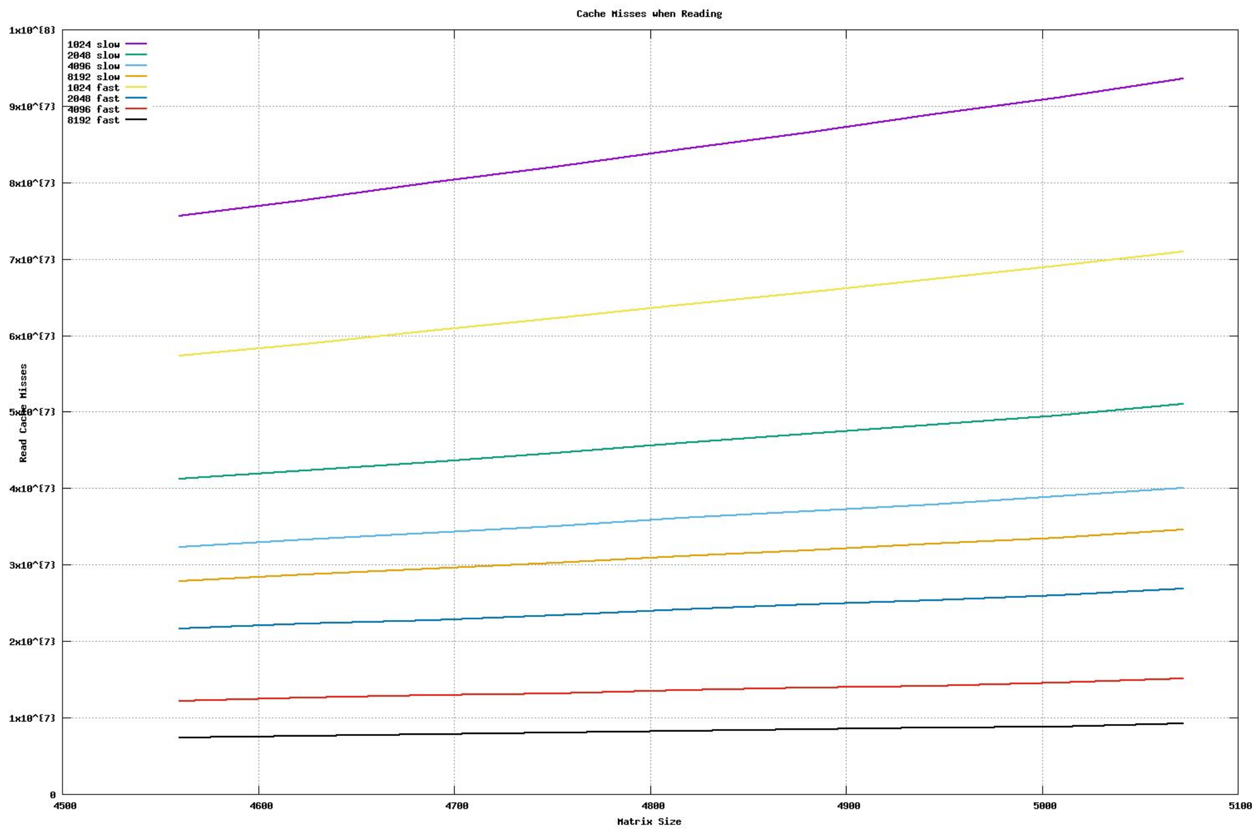


The plot represents a simulation of 10 iterations per matrix size.

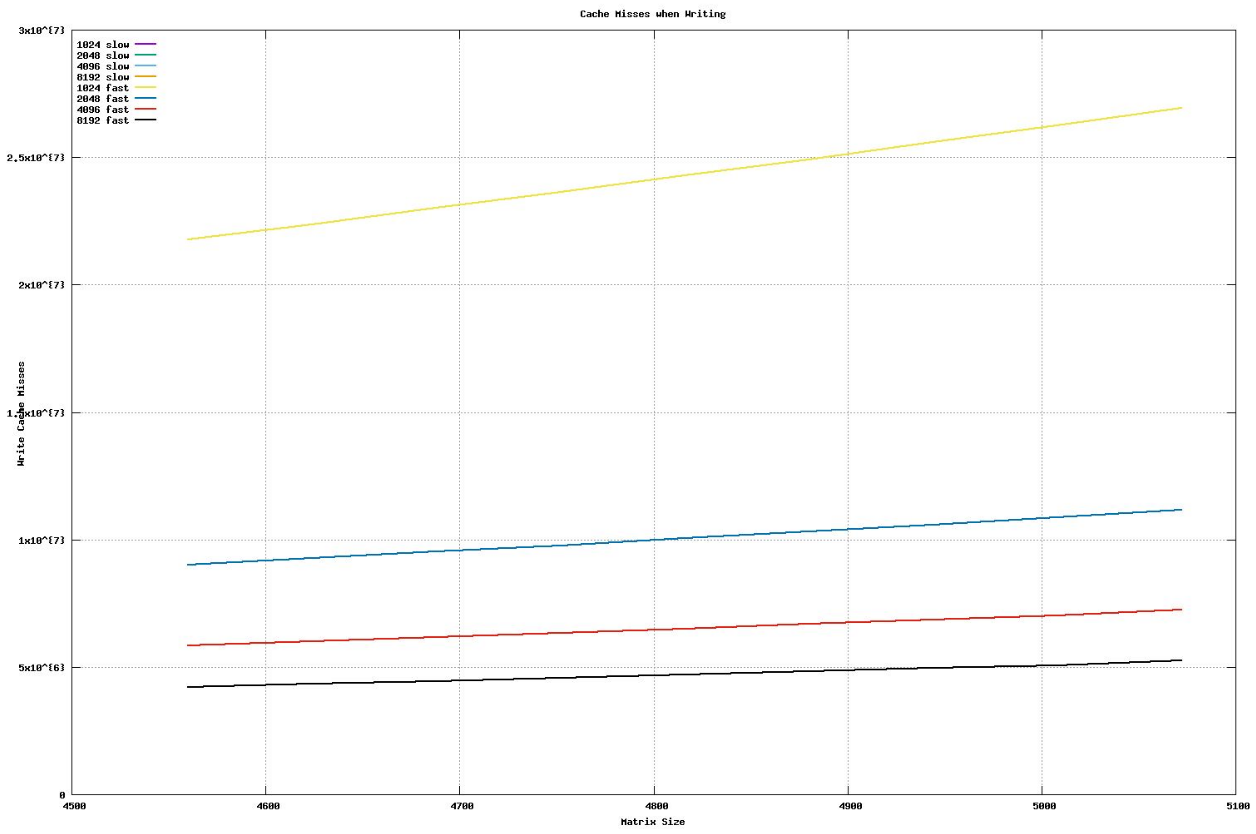
It can clearly be seen the gap widens the bigger the matrix becomes. In the fast execution the matrices are accessed by row, and the slow gets them by column, this causes that the greater N is, the less rows of the matrix fit in the cache. The slow only uses a small portion of the block before replacing it, while the fast uses the whole block before replacing it.

EXERCISE 2

In this exercise we were asked to get the cache misses of the same operation (matrix addition) depending on the size of the matrix for 4 different cache sizes (1024, 2048, 4096 and 8192).



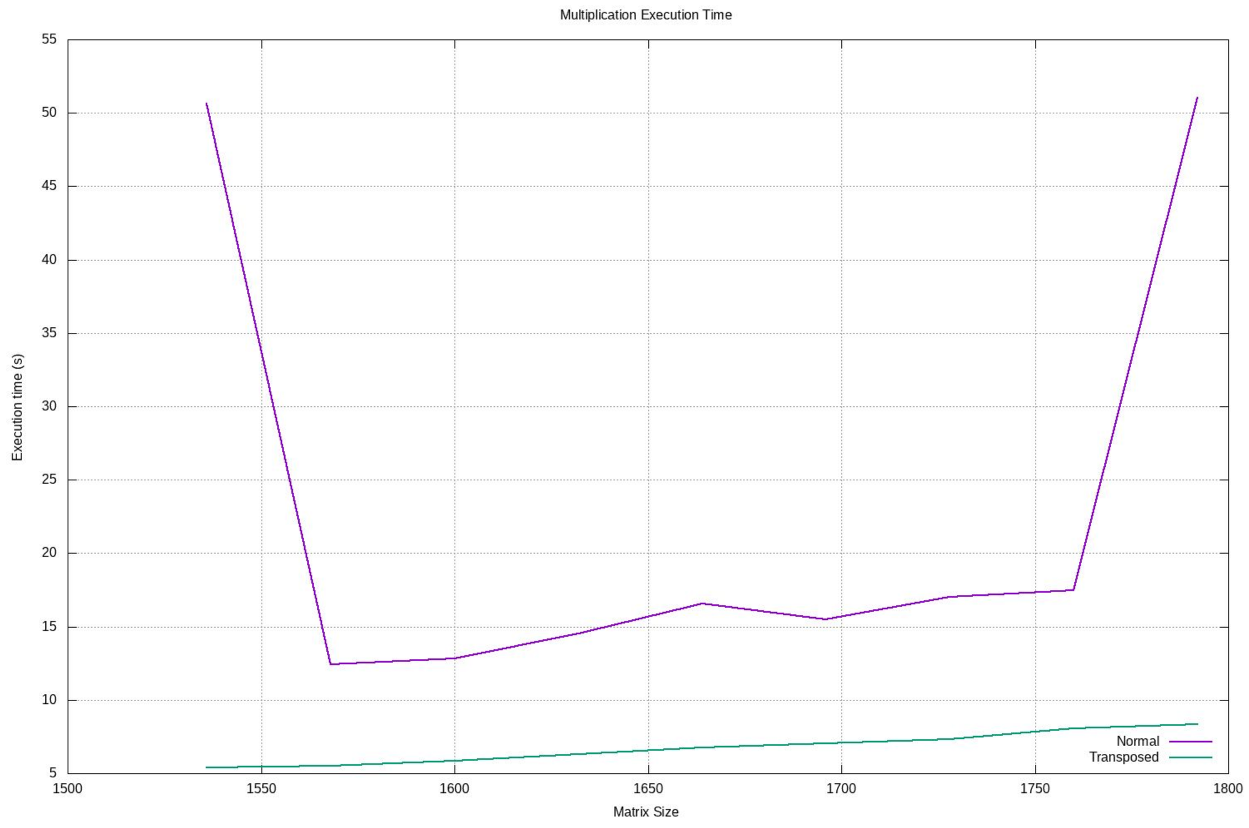
In this plot we can see that the difference between the cache sizes affects severely the performance, overall in the case of the slow.c program. In the case of slow.c, it will have to load more rows and the smaller the cache is, the more it will have to load, but it will always use only one value of the row. Meanwhile, fast.c will use the whole row that is loaded in the cache. This is why slow.c will be much more affected by cache size.



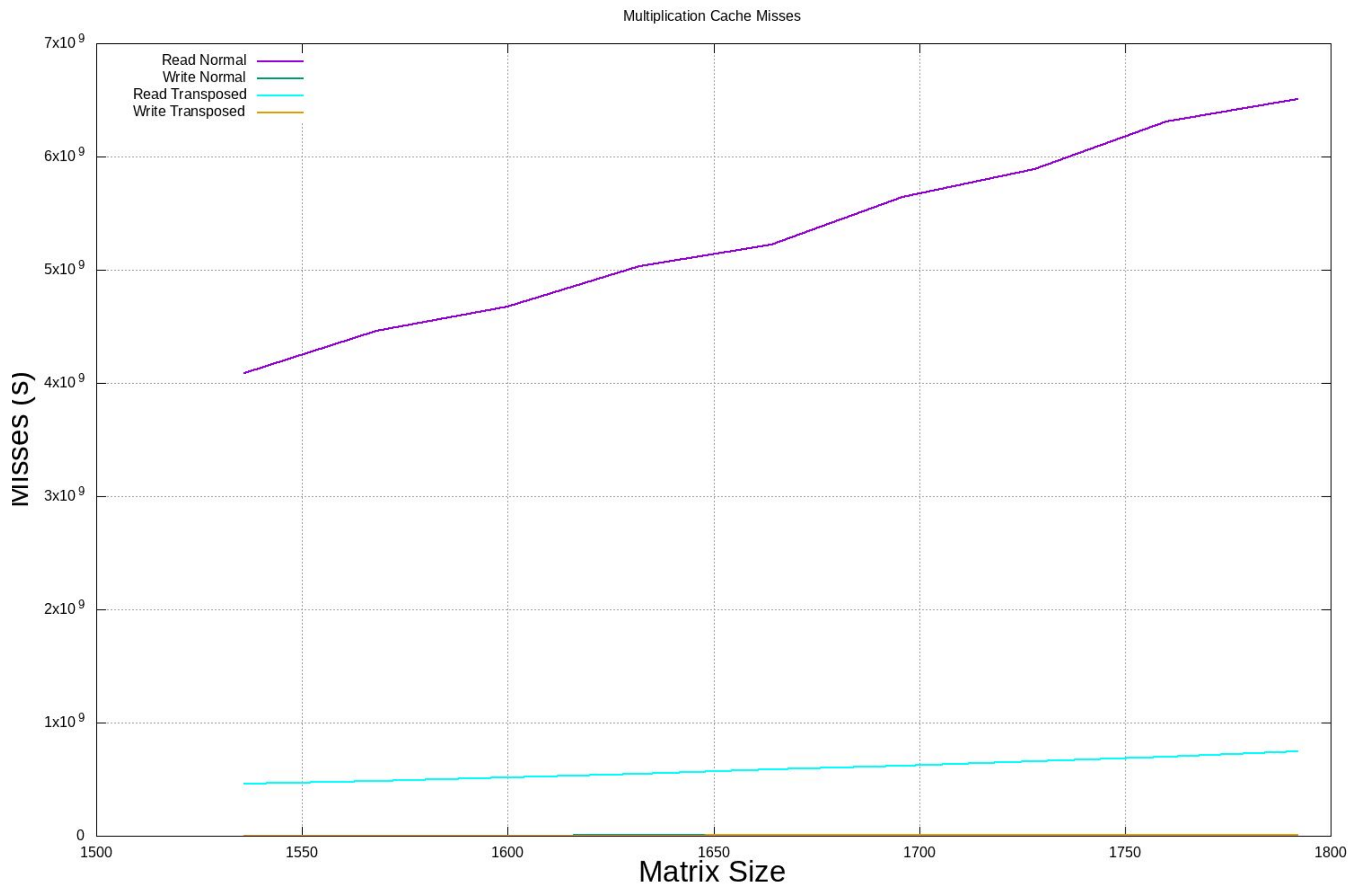
In this case the situation is more or less the same. In this plot it looks like there are only four lines, that is because the read and write are the same for slow.c and fast.c, since they write in the memory the same way so they have the same quantity of misses.

EXERCISE 3

In the third exercise we had to create a program that multiplies two matrices in two different ways: the first way multiplies as we usually did in math, adding the result of the multiplications of the rows of the first one and the columns of the second one; the second method was using the transposed matrix, so now to multiplicate we just needed to read thee matrix by rows, as the first one.



In this plot we can see a staggering result. In the normal matrix the first datapoint is the smallest size, which takes around 50 seconds, then we can see a steep decline which could be caused by the row size not fitting the block right, the problem is that when we simulate with valgrind we can't see this behaviour. We suspect that this is due to variations that we can't control, although it was simulated in the cluster. The transposed multiplication is much more stable, as expected and climbs much slower, the higher the size is the more it distances itself from the normal multiplication as the cost for transposing isn't nearly as high as multiplication.

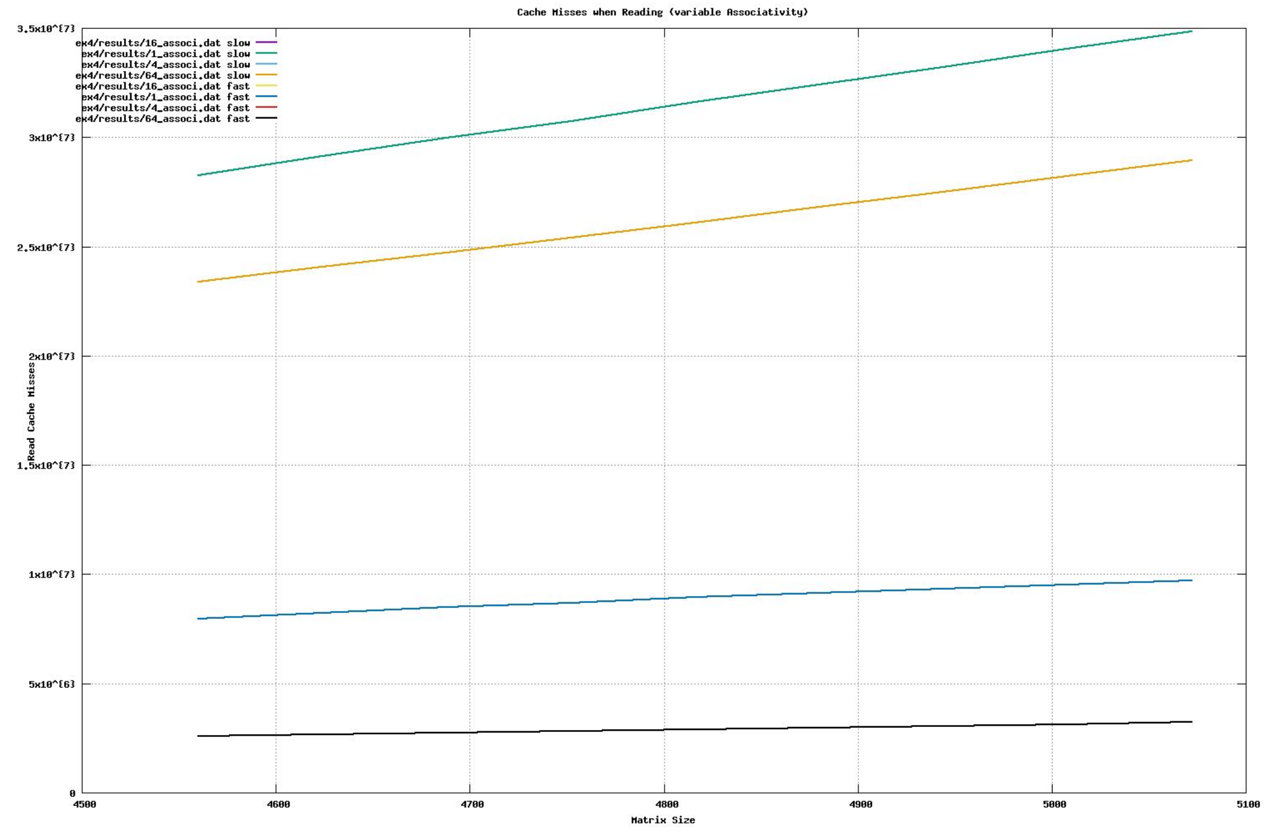


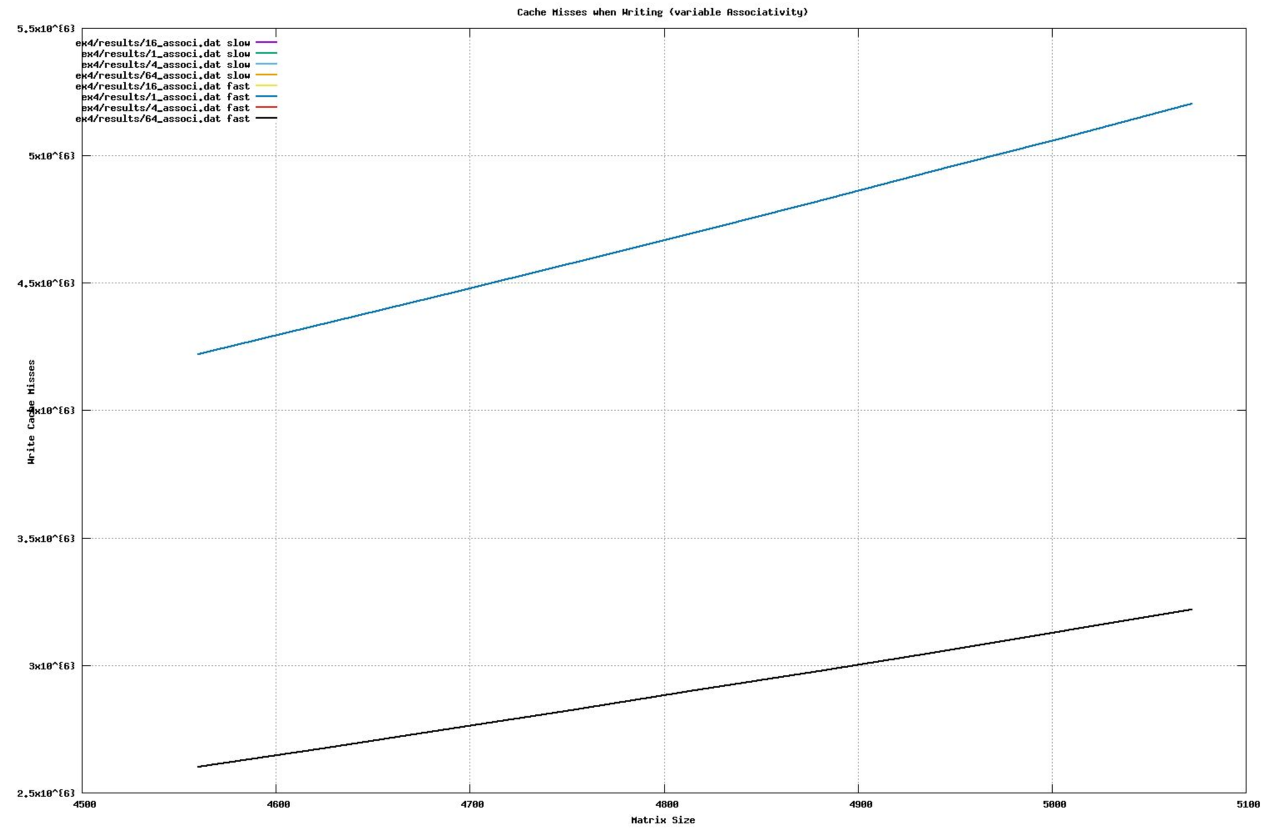
Here we can't see the write misses, they probably are very similar. What is really interesting here is how few misses the transposed has, moreover its advantage widens the bigger the matrix is, as the transposed is just a double loop, what generates the most misses is the triple loop. By multiplying line by line we are "eliminating" one loop, of course its still there but its in cache so is very fast doing the third loop.

EXERCISE 4

Here we played with different settings, specially line size, and observed the results.

Cache misses when number of ways changes in first level cache (Script4):

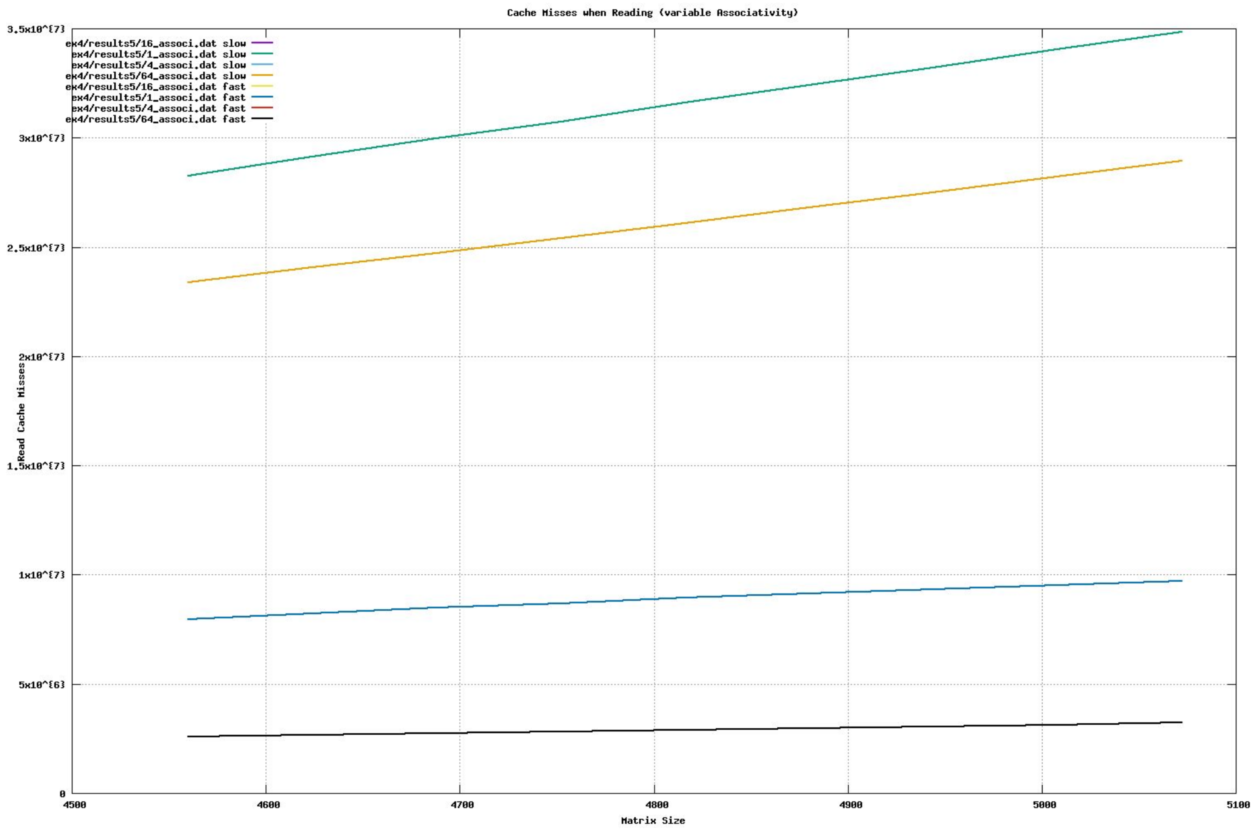

Cache Misses when Reading (variable Associativity)
ex4/results/16_associ.dat slow
ex4/results/1_associ.dat slow
ex4/results/4_associ.dat slow
ex4/results/64_associ.dat slow
ex4/results/16_associ.dat fast
ex4/results/1_associ.dat fast
ex4/results/4_associ.dat fast
ex4/results/64_associ.dat fast
3.5x10^{7}
3x10^{7}
2.5x10^{7}
2x10^{7}
1.5x10^{7}
1x10^{7}
5x10^{6}
0
Read Cache Misses
4500
4600
4700
4800
4900
5000
5100
Matrix Size

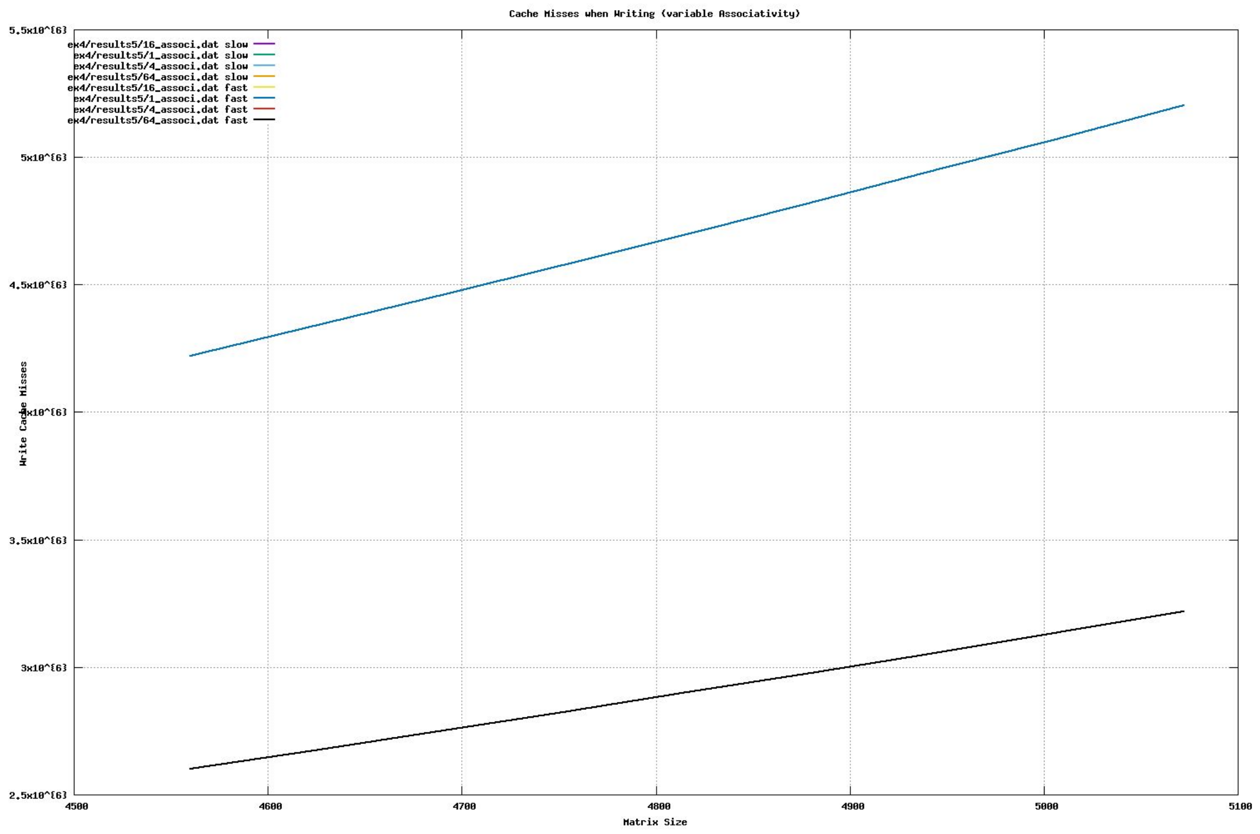

Cache Misses when Writing (variable Associativity)
ex4/results/16_associ.dat slow
ex4/results/1_associ.dat slow
ex4/results/4_associ.dat slow
ex4/results/64_associ.dat slow
ex4/results/16_associ.dat fast
ex4/results/1_associ.dat fast
ex4/results/4_associ.dat fast
ex4/results/64_associ.dat fast
5.5x10^{6}
5x10^{6}
4.5x10^{6}
4x10^{6}
3.5x10^{6}
3x10^{6}
2.5x10^{6}
Write Cache Misses
4500
4600
4700
4800
4900
5000
5100
Matrix Size

As expected the slow.c has more read misses than the fast.c, and the same misses in write, the different lines come from the direct-mapping cache which has significantly more misses for reading and writing:

The direct-map cache is the one with the most misses, and all the others are stacked in the bottom line (in read and write). This may be because the indexes in the cache address are always in a range and if we have less ways we will have longer indexes and we will have more unused blocks. Here we think that the unused blocks start between the 1 and 4 ways.
And as in exercise 2, the write misses between slow and fast are the same, that's why the second plot only has two lines.

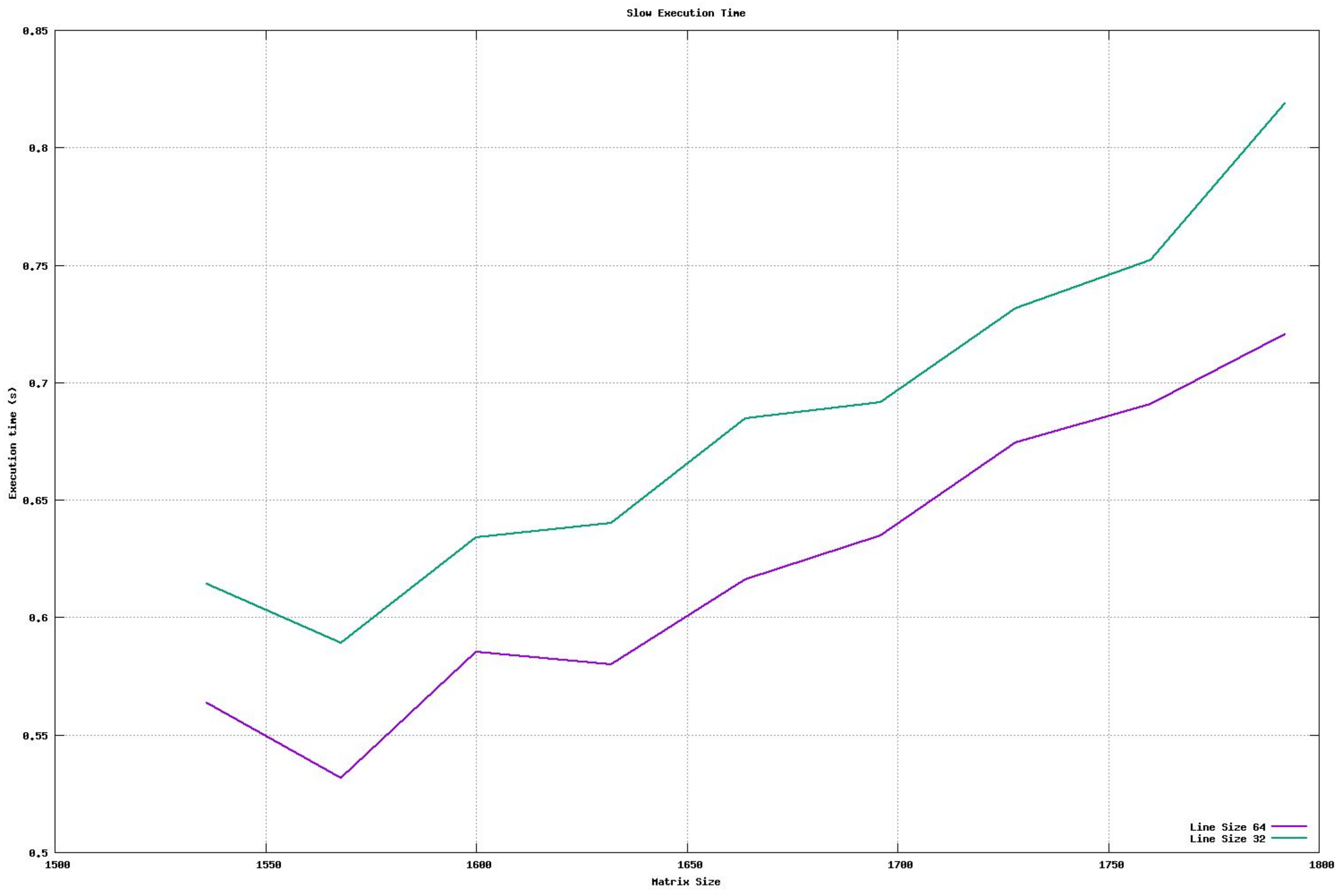
Varying number of ways in both first level and last level (Script5):

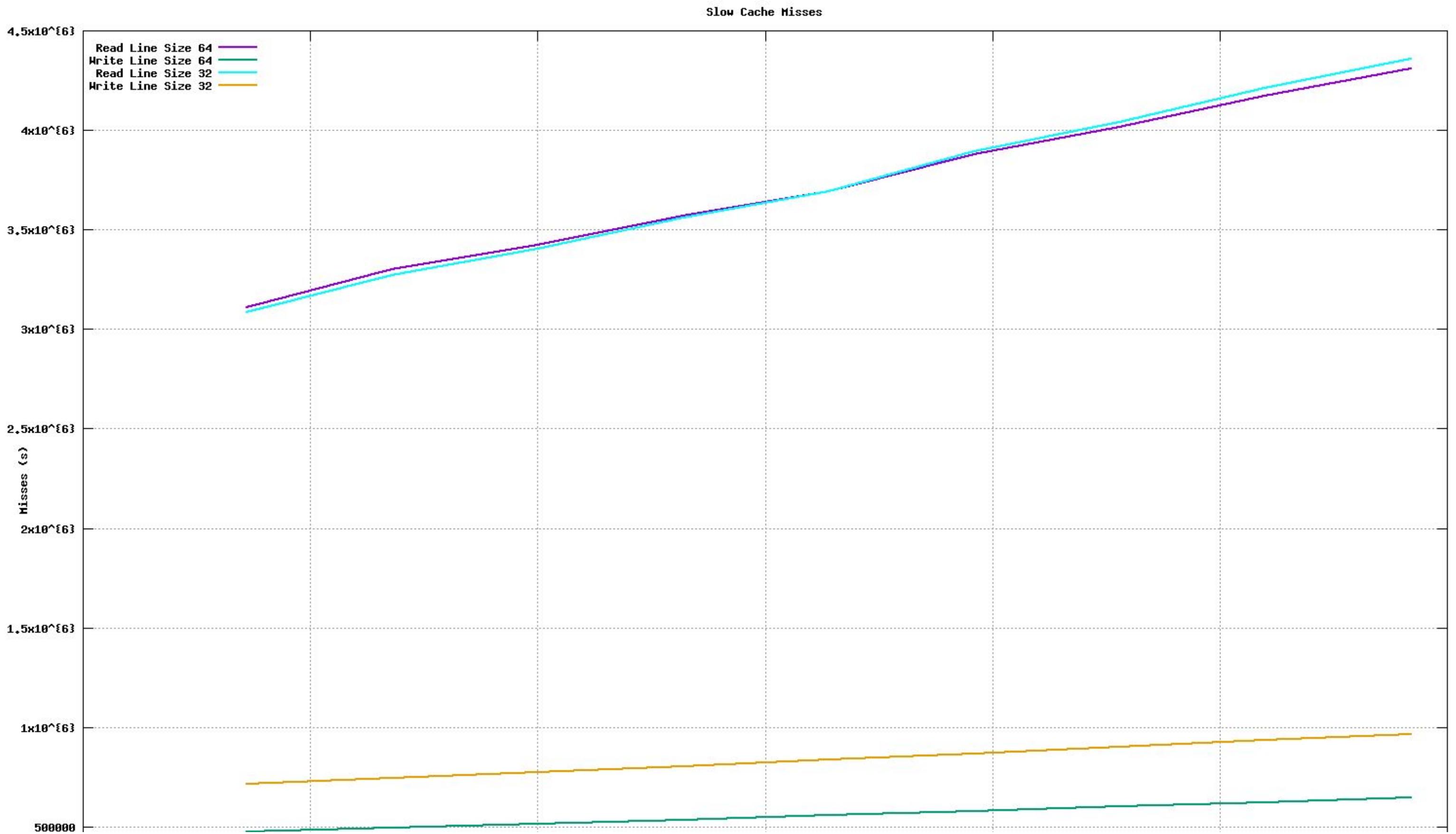




In this second test we did for the fourth exercise, we tried to also vary the second level cache, making it have the same size as the first level instruction and data caches. As we can see in the plots here, it did not affect the results too much. However, we thought it was an interesting inclusion, so we left it

Comparing Line sizes in slow.c (Script 6):


Slow Execution Time
Execution time (s)
Matrix Size
Line Size 64
Line Size 32
0.85
0.8
0.75
0.7
0.65
0.6
0.55
0.5
1500
1550
1600
1650
1700
1750
1800


Slow Cache Misses
Misses (s)
Read Line Size 64
Write Line Size 64
Read Line Size 32
Write Line Size 32
4.5x10^{6}
4x10^{6}
3.5x10^{6}
3x10^{6}
2.5x10^{6}
2x10^{6}
1.5x10^{6}
1x10^{6}
500000

sizes, but the bigger the matrix is the better the 64bytes block is doing. Also in the write misses there is a clear difference between the two line sizes, being 64 bytes better. One thing we thought was causing this difference is that maybe it has something to do with the number of operations it has to perform, given that having lines with twice the length will reduce the number of updates needed if the data is consecutive. Maybe when reading it applies a different policy than when it writes and that’s why when reading it doesn’t vary a lot and when writing it does.