CS304 - Assignment 3 Report

mkem114 - 6273632

October 20, 2017

1 Cache measurement

1.1 Processor name

`Intel Core i7 7500U`

1.2 Cache sizes

There is no L4 cache in this processor, L1 is split by data and instruction. L3 is the only shared cache, there is two of the L2 and both L1s for each physical core. For sizes of each cache refer to Figure 1

```
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_DCACHE_SIZE                 32768
LEVEL2_CACHE_SIZE                  262144
LEVEL3_CACHE_SIZE                  4194304
```

Figure 1: Extract from “getconf -a | grep CACHE_SIZE”

1.3 Cache measurement table

N	size of a / bytes	time per iteration / ns	time per iteration / ns
		linear access	random access
2048	8192	2.4	2.38
4096	16,384	2.38	2.51
8192	32,768	2.32	2.59
16,384	65,536	2.33	2.96
32,768	131,072	2.33	3.94
65,536	262,144	2.33	4.52
131,072	524,288	2.57	6.19
262,144	1,048,576	2.76	16.61
524,288	2,097,152	2.87	33.65
1,048,576	4,194,304	2.95	37.70
2,097,152	8,388,608	2.98	41.83
4,194,304	16,777,216	3.00	43.49
8,388,608	33,554,432	3.01	45.04
16,777,216	67,108,864	3.01	45.28

Figure 2: Results of running cachetest1 and cachetest2

1.4 Cache measurement chart

See Figure 3 and Figure 4

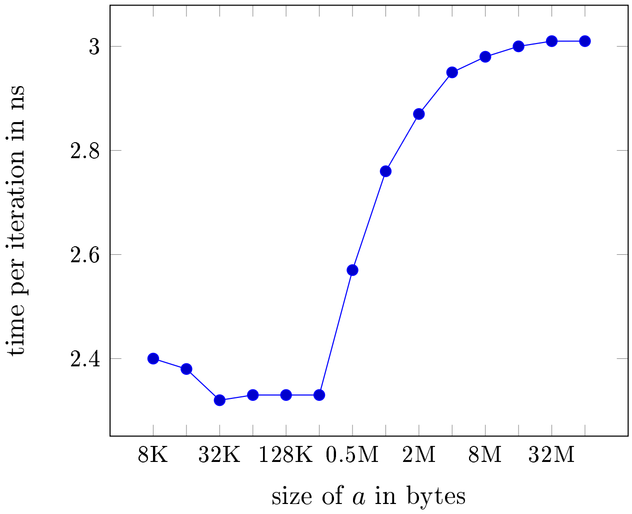

Figure 3: Time per iteration in nanoseconds over size of a in bytes for cachetest1

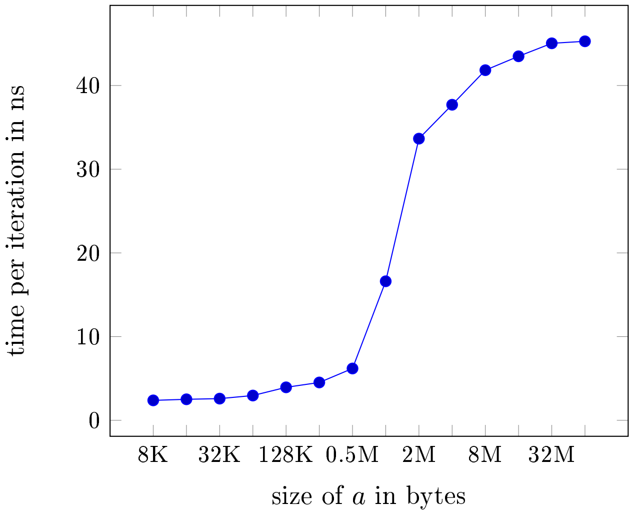

Figure 4: Time per iteration in nanoseconds over size of a in bytes for cachetest2

1.5 Differences and similarities

- From 8K till 256K the curves in Figure 3 and Figure 4 is almost the same except for the little start which will be because random access will lead to better loading on to L2 than linear. Normally everything is attempted to fit only L1 which is what happens with sequential. After the little dip there is very little change or rise in the curve because it starts just at a full L1 and goes till L2 approaches filling.

- The random access and sequential have the same trend curve shape. 512K till 8M because it is to do with the constant cache sizes of the procesor. At 512K (the first stop after exhausting L2 cache) the L2 cache is always full which is why it starts to take more time quite rapidly. The curve significantly slows from 2M till 8M as that is when the L3 cache starts to fill up and run out of space. The time penalty starts to stabilise because the L3 demand starts to stabilise at it approaches 100% use. The curve is so step because of the performance difference between L2 and L3.

- Both random access and sequential have the same trend curve shape. 16M onwards is four times the L3 cache, the time per iteration in ns won't go up because now almost everything will be memory only. The time penalty for accessing memory is constant per iteration as the percentage of cache misses stabilises so does the average time per iteration..

- Overall except for when both are run on a $8K$ array, cachetest2 is much slower because the a matrix has to be indexed randomly. Loading a in to the cache doesn't make much sense because it's too big for L3, it has to be read from memory and hence the relative time penalty factor.

2 Matrix product

As per the assignment specification the matrices were all of size 1000 × 1000 and to get some more accuracy there was 5 repetitions run. Times given have already been divided by 5.

2.1 Straight forward implementation

Took $3.26s$.
This implementation requires indexing one of the two matrices to read by rows because the whole column is required to calculate every element of the result matrix. This is slowed down because the speed advantages of cache aren't fully utilised (sequential elements aren't loaded into cache lines) due changing the row index N more times than the column index when 2D matrices in C are indexed by row then by column.

2.2 Temporary matrix

Took $1.54s$.

2.3 Blocking and temporary matrix

Took $1.07s$.
Note $k = 16$ blocks were found to be the fastest for me.
Instead of element-wise multiplication of a matrix's row and another's column for every element on a resulting matrix; taking a $k \times k$ block of one matrix, summing the element-wise multiplication of it with a $k * N$ subsection of the other matrix then combing the sums in the resulting matrix there is a performance boost.
This boost arises because the $k \times k \times size = 16 \times 16 \times 8 = 2048$bytes block, the value of each element 8bytes can both fit inside L1 cache and stay in there until they are never needed again. The stripe of the other matrix being multiplied $k \times n \times size = 16 \times 1000 \times 8 = 128000$ bytes can fit inside the L2 cache until it's never needed again thus all of the matrices' elements never have to be loaded from memory more than once.