COMPSYS304
Assignment 3
Oscar Sims - osim082 - 422812676

Cache Measurement

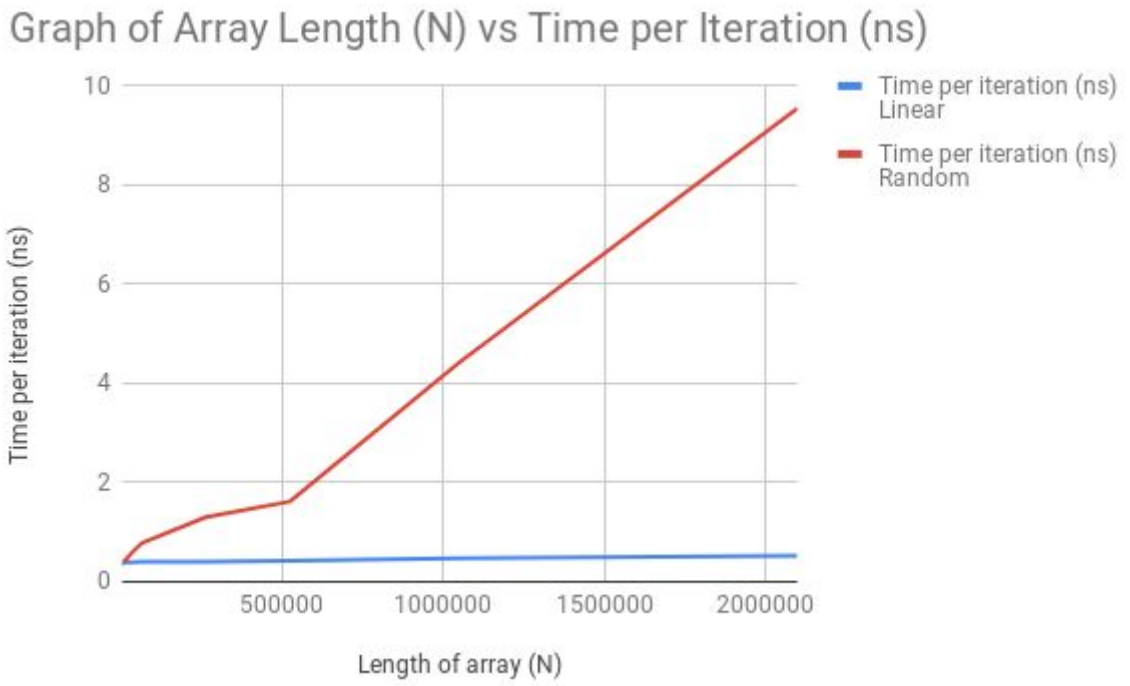
What is the name of the CPU?

- Intel Core i5-8250U @ 1.60GHz

What are the cache sizes?

- L1 instruction cache size: 32768 bytes (32 KiB)
- L1 data cache size: 32768 bytes (32 KiB)
- L2 cache size: 262144 bytes (256 KiB)
- L3 cache size: 6291456 bytes (6 MiB)

N	Size of A	Time per iteration (ns) ***Linear***	Time per iteration (ns) ***Random***
4096	16 KiB	0.4175	0.3983544922
5461	24 KiB	0.3794140267	0.3971964842
8192	32 KiB	0.3996374512	0.3927807617
32768	128 KiB	0.3951165771	0.5782531738
65536	256 KiB	0.4010864258	0.7819366455
262144	2 MiB	0.4038438416	1.297904243
524288	4 MiB	0.4253013611	1.619021263
1048576	8 MiB	0.4702310467	4.422646542
2097152	16 MiB	0.5187833738	9.550091634



The time per iteration remains essentially constant when using linear accesses. This is because the CPU is able to exploit the fact that the next item accessed from the array will always be adjacent to the current item. This means that it will be located next to the current item in the cache, which allows it to be retrieved very quickly.

In contrast, the random access pattern is different. It follows a roughly logarithmic trend of increasing time-per-iteration until the maximum size of the largest cache is exceeded. This logarithmic trend is to do with the logarithmically-increasing cache sizes. After that point, the time per iteration increases linearly.

We observe this trend because the random access pattern cannot exploit cache locality as aggressively as the linear access pattern. For array sizes less than the size of the largest 6MiB cache, the time to access will be dependent on which cache the next item is in; as the size of N increases, it becomes more and more probable that it will be located in a larger, slower cache. After the size exceeds 6MiB, the array will inevitably have to access main memory for some of its reads, which is slow (but constant-time, hence the linear trend).

Matrix Product

In the naive implementation 1, the problematic access pattern is that we are not accessing the elements of the array in a way that maintains the principle of locality, because we are not accessing elements that are adjacent in memory. Each iteration of the three loops accesses different parts of the three arrays A, B, and C. For example, at each iteration of the inner loop *k*, we access a different row of the matrix B. Given the way the memory for B is laid out, we don't take advantage of the cache, and there is a cache miss.

The blocking approach improves access patterns, by dealing with blocks of data (which will be located nearby in memory). The implementation consists of six nested loops. The outer three loops control the current blocks being accessed in the matrices. The inner three loops control the calculation of the matrix product itself (using the same implementation from 1 bounded by the outer loop variables); the implementation is order-independent, so we can control the particular memory locations we access without affecting the correctness of the algorithm.

(16 was the fastest measured block_size. 16 x sizeof(double) is 16 x 8, which is twice the size of the cache line.)

Each implementation is roughly twice as fast as the previous one:

Matrix 1 (Naive): 2.50s
Matrix 2 (Transposed): 1.2s
Matrix 3 (Blocked): 0.67s