COMPSYS 304 - Assignment 3

Dylan Fu (ID: 165746216), University of Auckland 13/08/2018

1 CPU Details

Intel Core i7 7700HQ	
L1	4 x 32 KB 8-way set associative instruction cache 4 x 32 KB 8-way set associative data cache
L2	4 x 256 KB 4-way set associative caches
L3	6 MB 12-way set associative shared cache

Table 1: Table of CPU cache details.

2 Cache Measurement

N	**Size of a** 32 bits (size of int) * N	**Time per Iteration** Case 1	**Time per Iteration** Case 2
4096	16KiB	393ps	390ps
8192	32KiB	395ps	404ps
16384	64KiB	392ps	566ps
32768	128KiB	390ps	638ps
65536	256KiB	393ps	756ps
131072	512KiB	390ps	915ps
262144	1MiB	392ps	1.16ns
524288	2MiB	409ps	1.56ns
786432	3MiB	410ps	2.21ns
1048576	4MiB	433ps	2.92ns
1572864	6MiB	439ps	4.32ns
2097152	8MiB	444ps	4.76ns
4194304	16MiB	454ps	5.27ns
6291456	24MiB	461ps	5.93ns
8388608	32MiB	462ps	6.56ns
16777216	64MiB	459ps	7.23ns

Table 2: Table of measured time per iteration for each case.

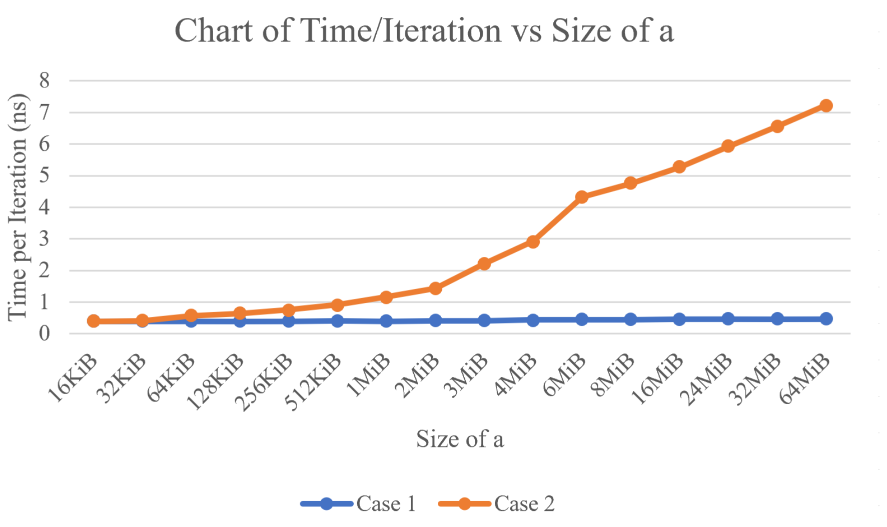

Figure 1: Chart of the time-per-iteration over size of a

The results in Table 2 indicate four plateaus as size of a increases. Due to the different speeds in L1, L2, L3 cache and main memory. We can deduce that when the size of a exceeds the levels cache capacity, it will have to load from from the next level down, thus increasing the access time due to the memory speed difference. The increase in times is seen in case 1 from 390ps -> 410ps -> 430ps -> 460ps. Also the results show a major difference between case 1 and case 2 especially as the size of a increases. In case 1, the array values are sequentially accessed, the hardware prefetcher will then automatically prefetch the next array values. However, in case 2 the array values are randomly accessed, therefore the system won't automatically prefetch further values leading to a higher miss rate compared to case 1. Consequently the performance implications of misses dramatically increases each cache/memory level down as the memory speed decreases which is shown in Figure 1.

3 Matrix Product

Straight forward implementation - Time: 0.97s

Listing 1: Code for straight forward implementation in C

```c
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        for (int k = 0; k < N; ++k) {
            c[i][j] += a[i][k] * b[k][j];
        }
    }
}
```

In the simple implementation, matrix **a** is accessed sequentially (row-major order). However, **b** is accessed in column-major order. This is not good, as it is essentially performing random accesses on matrix **b**. Thus, impacting performance negatively as accessing memory in contiguous locations is faster than jumping around among locations.

Temporary transpose matrix implementation - Time: 0.41s

Listing 2: Code for temporary transpose matrix implementation in C

```c
int tmp[N][N];
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        tmp[i][j] = b[j][i];
    }
}

for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        for (int k = 0; k < N; ++k) {
            c[i][j] += a[i][k] * tmp[j][k];
        }
    }
}
```

In the temporary matrix implementation, we solve the issue of the first implementation, by creating a temporary matrix for the transpose of matrix **b**. So that both matrices are accessed in sequential order.

Blocking implementation - Time: 0.67s

Listing 3: Code for blocking implementation in C

```c
for (int k = 0; k < N; k+=blocksize) {
    for (int j = 0; j < N; j+=blocksize) {
        for (int i = 0; i < N; i+=blocksize) {
            for (int i2 = i; i2 < min(i+blocksize,N); ++i2) {
                for (int j2 = j; j2 < min(j+blocksize,N); ++j2) {
                    for (int k2 = k; k2 < min(k+blocksize,N); ++k2) {
                        c[i2][j2] += a[i2][k2] * b[k2][j2];
                    }
                }
            }
        }
    }
}
```

In the blocking implementation, we improve on the straight forward implementation by performing matrix multiplication on smaller sub-matrices of size **kxk**. To fully optimise the use of the cache line we can select **k** as the cache line size divided by the size of int. The implementation has three outer loops iterating with intervals of **k**. This divides the matrix into several smaller matrices which can be handled with more cache locality. The inner loops iterate over the missing indexes of the outer loops and then perform matrix multiplication.