CS3210 Assignment 1
Cache Profiling and Performance Optimizations

Chong Yun Long
A0072292H

1 Cache Hierarchy Profiling

1.1 Cache profile of computers used

There are 2 types of computers in the lab.

- Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz
  - Level 1 Cache: 32 kB
  - Level 2 Cache: 256 kB
  - Level 3 Cache: 8144 kB
  - Cache Line: 64 bytes
  - Write Policy: Write-back cache
- Intel(R) Core(TM) i5-2400S CPU @ 2.50GHz
  - Level 1 Cache: 32 kB
  - Level 2 Cache: 256 kB
  - Level 3 Cache: 6144 kB
  - Cache Line: 64 bytes
  - Write Policy: Write-back cache

1.2 Cache Line

1.2.1 Methodology

Every time a cache miss occurs, a block of data is transferred from the main memory into the cache. To find out the size of the block, we have to find out the boundary which separates the current block and the next block.

This is done by accessing a memory array with increasing strides. The goal is to find out what is the length of stride required to jump from the current cache block to the next cache block in exactly 1 stride. We should expect to see a huge increase in access times if a different cache line is accessed for each stride.

1.2.2 Results of Intel(R) Core(TM) i7-2600 CPU

Table 1 shows the results obtained from program runs on Intel(R) Core(TM) i5-2400S CPU:

	Stride Size in bytes/Runtime in milliseconds									
	2	4	8	16	32	64	128	256	512	1024
Run 1	133.37	154.41	171.09	123.04	407.59	1301.13	1536.68	2699.99	2727.01	2692.31
Run 2	129.93	156.36	168.25	120.94	398.35	1289.76	1533.44	2680.69	2713.9	2684.64
Run 3	129.03	158.18	166.17	121.7	390.15	1276.09	1530.17	2670.76	2700.27	2676.42
Average	130.78	156.32	168.50	121.89	398.70	1288.99	1533.43	2683.81	2713.73	2684.46

Table 1: Cache line measurements of Intel(R) Core(TM) i7-2600

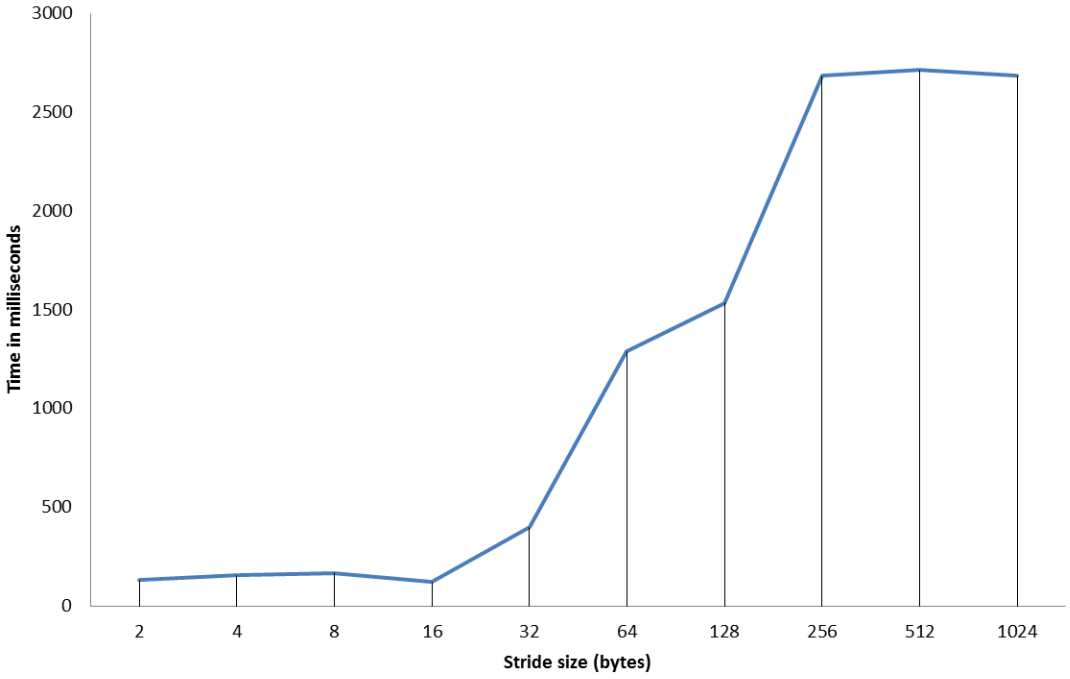

Figure 1: Cache line strides and access times on Intel(R) Core(TM) i7-2600

1.2.3 Results of Intel(R) Core(TM) i5-2400S CPU

Table 2 shows the results obtained from program runs on Intel(R) Core(TM) i5-2400S CPU:

	Stride Size in bytes/Runtime in milliseconds									
	2	4	8	16	32	64	128	256	512	1024
Run 1	151.51	185.65	208.13	224.44	828.57	2347.8	3408.93	4396.15	3760.46	3767.04
Run 2	150.84	180.86	204.18	190.65	697.03	1996.12	2697.9	3704.53	3458.16	3452.66
Run 3	149.06	183.12	208.01	234.3	767.95	2190.93	3055.12	4048.98	3658.54	3675.5
Average	150.47	183.21	206.78	216.46	764.52	2178.28	3053.98	4049.89	3625.72	3631.73

Table 2: Cache line measurements of Intel(R) Core(TM) i5-2400S

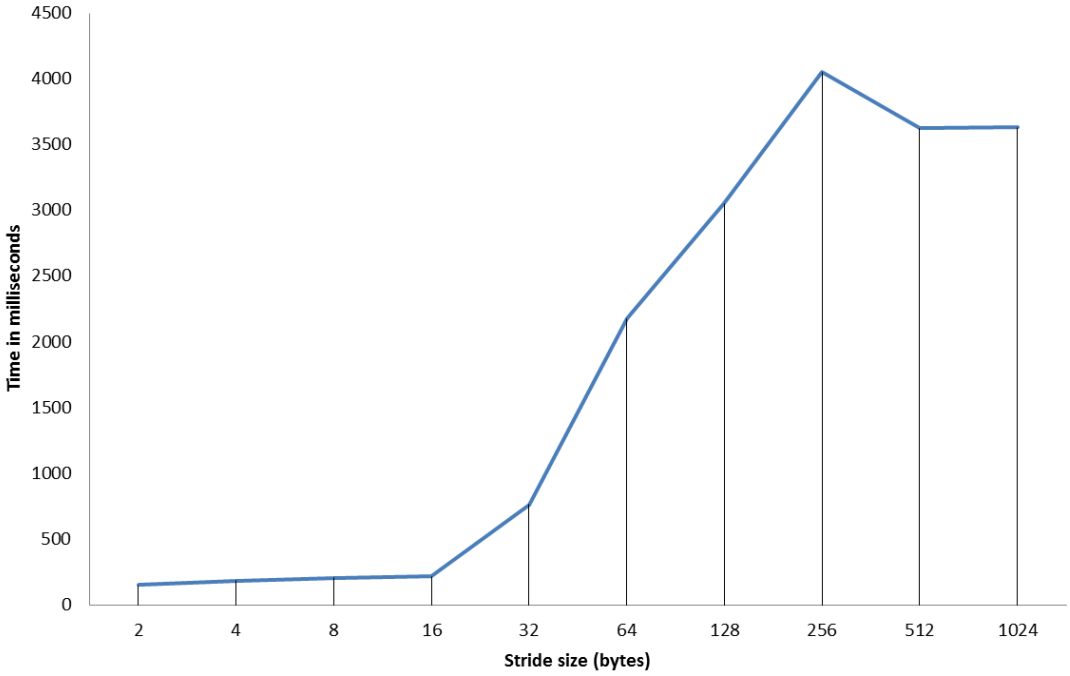

Figure 2: Cache line strides and access times on Intel(R) Core(TM) i5-2400S

1.2.4 Analysis of results

As shown in Figure 1 and Figure 2, the first steep increase is seen when stride size changed from 32 bytes to 64 bytes indicating cache misses for each access.

Conclusion: Cache line for both CPUs is 64 bytes.

We see further spikes after 64 bytes in both Figure 1 and Figure 2. This could by due to the associativity of the cache which may result in increased access times.

1.3 Cache Levels

1.3.1 Methodology

There are cache levels of various sizes and access times on modern CPUs. Multi-level caches operates by accessing caches of the smallest cache first since it is the fastest. The next larger cache is then check if there is a miss.

The strategy here is to access the array in increasing sizes. If there is a sudden increase in access time, it is likely that data is being read from the next level of cache which is slower.

For dedicated caches, the spike is likely to be obvious since the memory is not being shared with other processors which may be running background as well as OS applications. For shared caches, the memory may have already been partially occupied by other processors. Hence, the spike may be less obvious.

It is important to note that cache sizes need not grow in powers of 2 especially for cache sizes above 1 MB. Hence the code is designed such that the array size doubles until it reaches 1 MB. For sizes above 1 MB, the array size increase by 2 MB for each iteration.

Results will be presented separately for L3 cache since the access time is too large to be plotted together with L1 and L2 cache.

1.3.2 Results of Intel(R) Core(TM) i7-2600 CPU

Array Size in kilobytes/Runtime in milliseconds

	8	16	32	64	128	256	512	1024	2048
Run 1	78.18	78.33	75.27	88.51	85.65	97.22	138.36	139.49	138.69
Run 2	77.75	78.61	76.95	88.71	85.23	98.08	138.65	139.87	141.02
Run 3	78.14	78.21	79.22	88.84	85.7	96.84	140.93	139.74	139.49
Average	78.02	78.38	77.15	88.69	85.53	97.38	139.31	139.7	139.73

Table 3: L1, L2 Cache size measurements of Intel(R) Core(TM) i7-2600

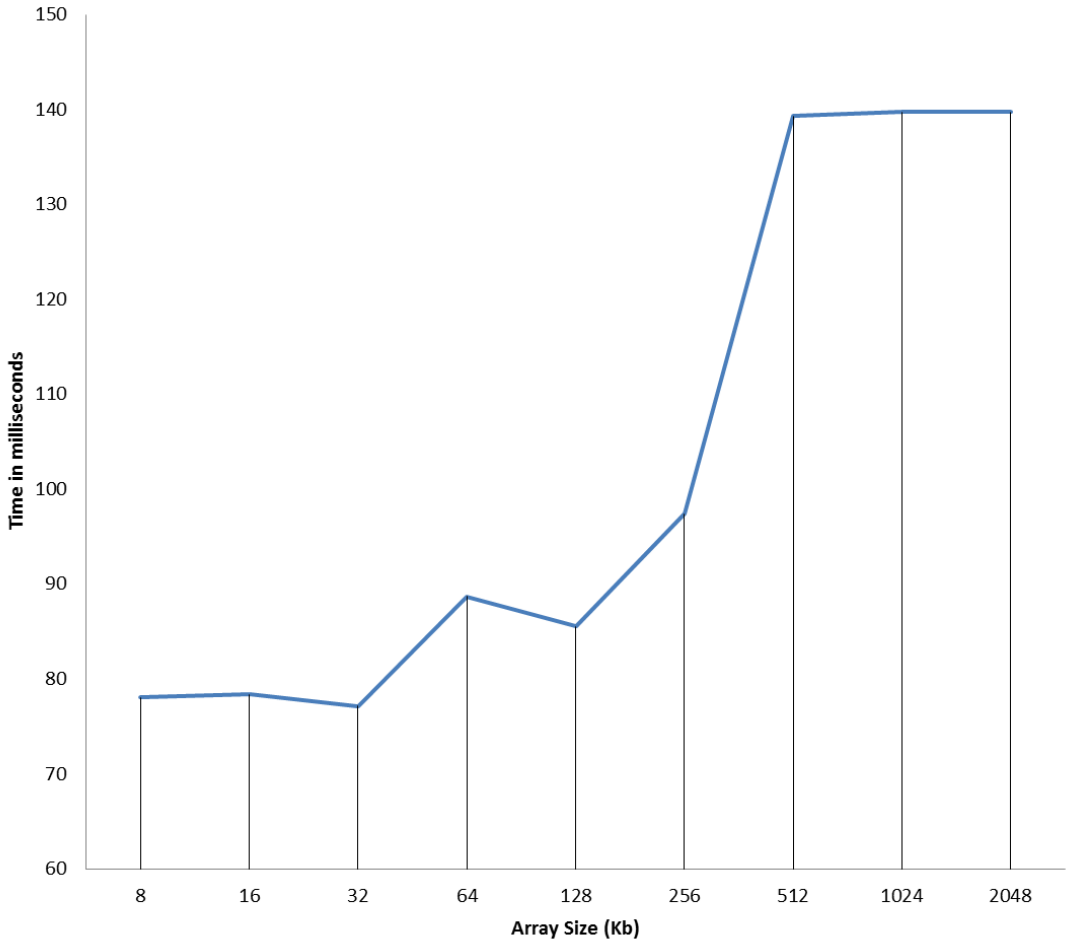

Figure 3: Array size and access times on Intel(R) Core(TM) i7-2600

Array Size in kilobytes/Runtime in milliseconds

	2	4	6	8	10	12	14	16
Run 1	138.69	150.78	225.26	449.71	705.31	765.02	816.33	838.41
Run 2	141.02	151.06	282.33	516.67	698.3	750.81	815.41	845.09
Run 3	139.49	150.85	219.1	505.63	667.74	753.23	809.21	832.11
Average	139.73	150.90	242.23	490.67	690.45	756.35	813.65	838.54

Table 4: L3 Cache size measurements of Intel(R) Core(TM) i7-2600

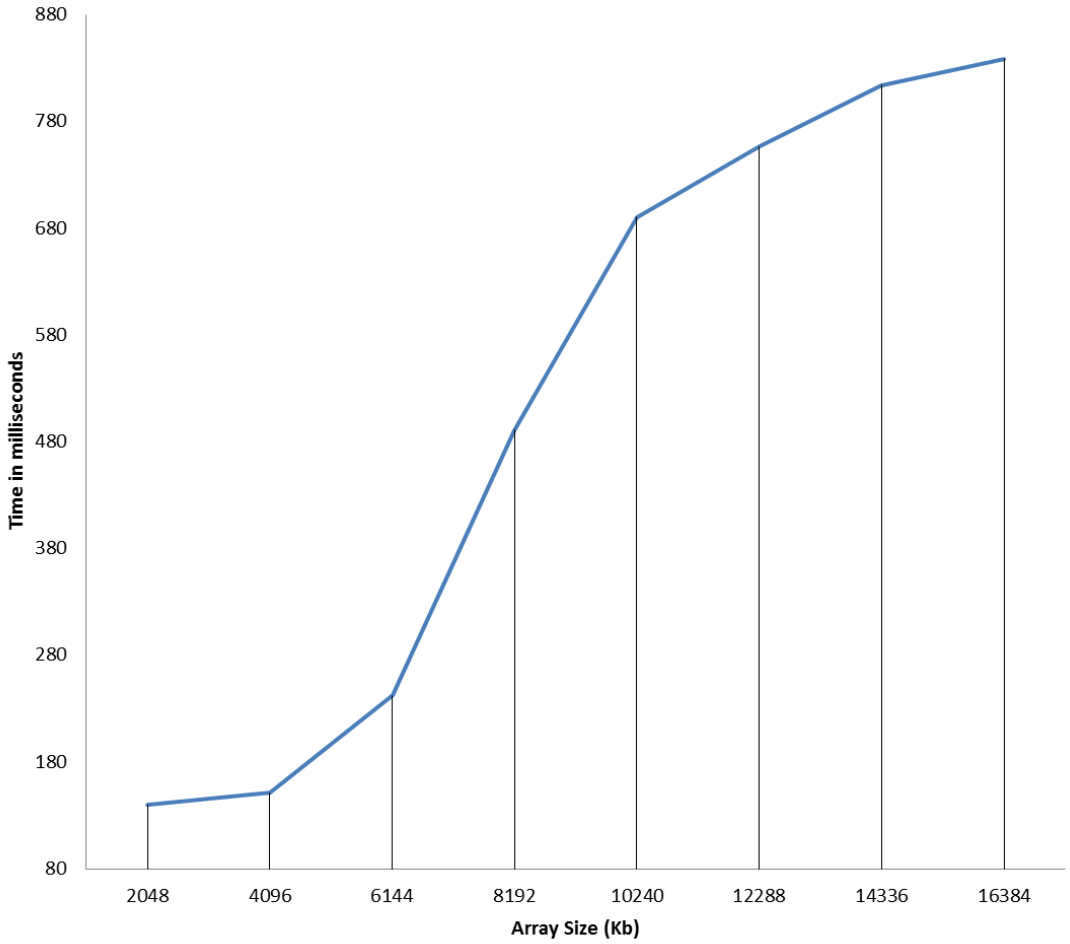

Figure 4: Array size and access times on Intel(R) Core(TM) i7-2600

1.3.3 Results of Intel(R) Core(TM) i5-2400S CPU

Array Size in kilobytes/Runtime in milliseconds

	8	16	32	64	128	256	512	1024	2048
Run 1	89.25	89.8	90.98	101.93	110.47	131.06	164.37	164.54	164.11
Run 2	91.54	90.16	87.58	101.95	97.87	108.31	161.94	161.35	160.52
Run 3	91.61	90.67	90.09	100.94	98.1	127.95	163.19	164.52	191.8
Average	90.8	90.21	89.55	101.61	102.15	122.44	163.17	163.47	172.14

Table 5: L1, L2 Cache size measurements of Intel(R) Core(TM) i5-2400S

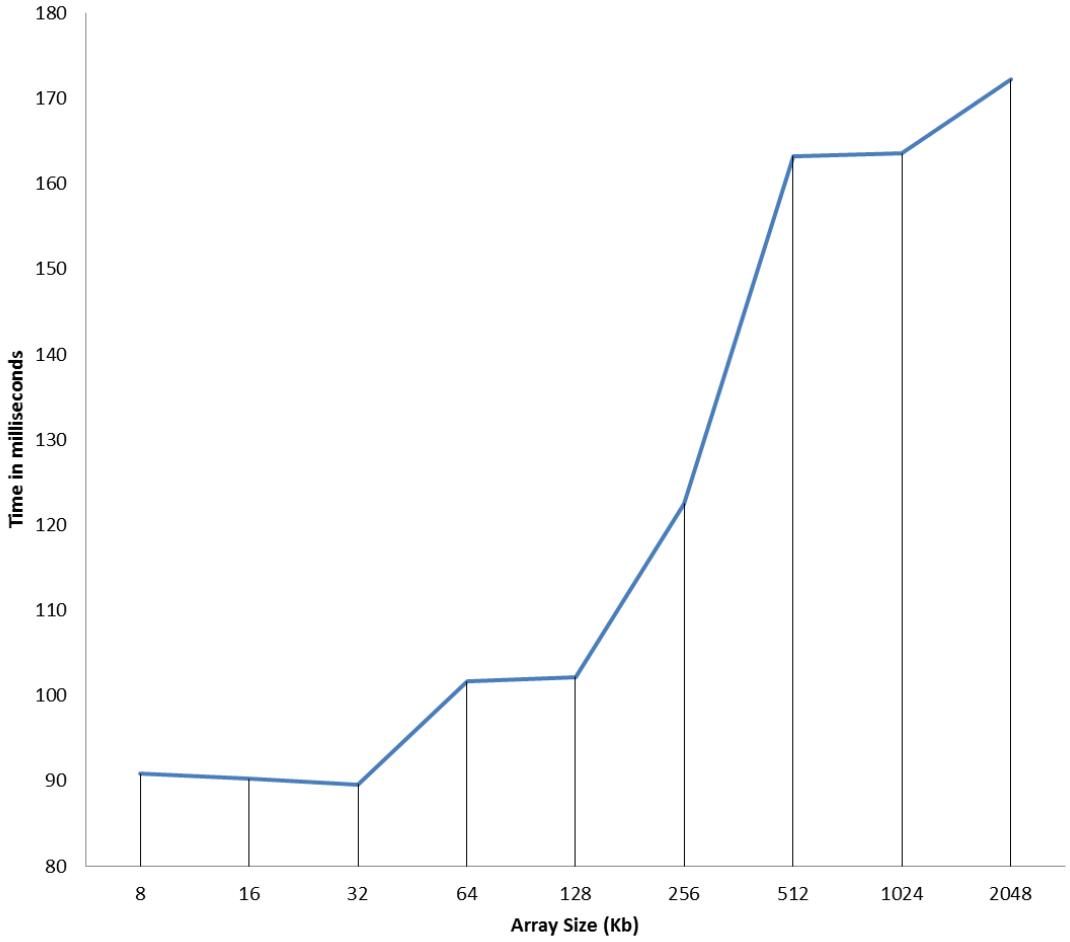

Figure 5: Array size and access times on Intel(R) Core(TM) i5-2400S

	Array Size in kilobytes/Runtime in milliseconds							
	2	4	6	8	10	12	14	16
Run 1	164.11	354.41	974.11	1536.3	1819.06	1929.75	1977.69	1999.83
Run 2	160.52	362.64	915.67	1516.49	1820.79	1947	1984.41	2004.4
Run 3	191.8	398.85	921.72	1503.61	1774.61	1905.61	1962.38	1981.98
Average	172.14	371.97	937.17	1518.8	1804.82	1927.45	1974.83	1995.40

Table 6: L3 Cache size measurements of Intel(R) Core(TM) i5-2400S

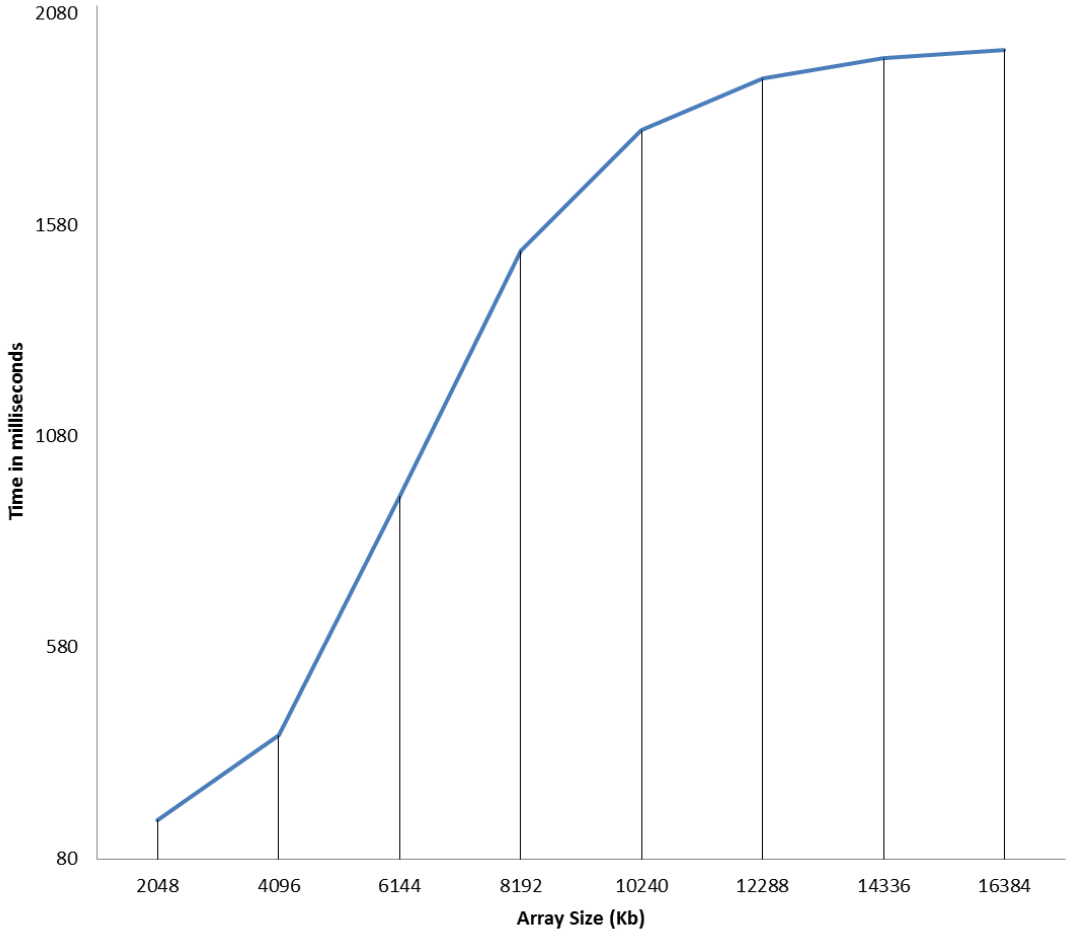

Figure 6: Array size and access times on Intel(R) Core(TM) i5-2400S

1.3.4 Analysis of results

As shown in Figure 3 and Figure 5, the first steep increase is seen when array size changed from 32 kB to 64 kB. It is likely that CPU have been reading from L1 cache up till 32 kB. When a read is then performed for an array of size 64 bytes, the CPU is reading data from both L1 and L2 cache. Since L2 cache is much slower than L1, the access time increases.

Conclusion: L1 cache for both CPUs is 32 kB.

Since L2 cache is most likely to be share by a pair of cores, the cache have already been preoccupied with data from another core. Hence, small amounts of data may spill over to the L3 cache or the main memory even though the array that is being read is smaller or equal to that of L2 cache. Hence it is likely that we see a small spike followed by a larger spike in access times. The small spike occurs due to the small spillover of data. The larger one occurs when the array is larger than the cache size, a large portion of the data has to be read from the next level cache/memory. Access times should stabilise after the 2nd spike as most of the data will be stored on L3 cache.

From Figure 3, we see a small spike from 128 kB to 256 kB followed by a larger spike from 256 kB to 512 kB. The L2 cache is likely to be 256 kB. The increase from 128 kB to 256 kB is due to a small spillover. Increase from 256 kB to 512 kB due to a large portion of the data is being used in L3. From 512 kB, most of the data will be stored in the L3 cache. Hence, the access time stabilises.

From Figure 5, we see a similar trend. However the spike from 128 kB to 256 kB is much larger than that of Figure 3. The access time stabilises after 512 kB.

Conclusion: L2 cache for both CPUs is 256 kB.

Not all CPUs have L3 cache. However, if the CPU contains a L3 cache, we will see patterns similar to what we have saw in Figure 3 and Figure 5 for L2 cache. However, the first spike due to spillover of L3 cache will be larger because the difference in latency between L3 and main memory is much higher than between L2 and L3. L3 is also likely to be preoccupied with more data considering that it will be shared by all the cores rather than just 1 pair of cores.

From Figure 4, We see 2 spikes from 6144 kB to 8192 kB and from 8192 kB to 10240 kB. After 10240 kB the access time slowly plateaus.

Similarly in Figure 6, there is a spike from 4096 kB to 6144 kB and from 6144 kB to 8192 kB. After 8192 kB the access time slowly plateaus.

Conclusion:

L3 cache of Intel(R) Core(TM) i7-2600: 8192 kB

L3 cache of Intel(R) Core(TM) i5-2400S: 6144 kB

1.4 Write Policy

There are major differences between the various write policies employed.

For write-through, writes are performed synchronously to both the cache as well as the main memory. Normally a write buffer will be used to hold the data that will be written to the main memory so that there is no need for the CPU wait for a memory write. The memory latency is observed when the write buffer is fully filled and the CPU has to stall when there is a write to memory.

1.4.1 Methodology

We exploit the fact that write-through has a higher latency than write-back once the write buffer has been filled. Consider 2 arrays of different sizes. We must first fill the write buffer by performing a series of writes. We then measure the time it takes to write to 2 arrays of different sizes which occupies different levels of cache. The 2 arrays should be small enough to fit entirely into the cache so that there will not be any accesses to the main memory if write-back is used.

If write-through is used, the time taken would be approximately equal since the latency of memory writes will dominate the latency of cache writes. In this case, the size of the array does not matter.

If write-back is used, differences will be seen since writes are being performed to different cache of different speeds.

The access times for each write can also be calculated. The access time to the main memory should be easily more than 20 ns.

1.4.2 Results

	Runtime in milliseconds			
	i7-2600		i5-2400S	
	1 Mb	6 Mb	1 Mb	6 Mb
Run 1	314.38	467.81	375.34	2063.23
Run 2	313.93	579.49	375.25	2090.47
Run 3	314.2	513.61	375.7	2012.76
Average	314.17	520.30	375.43	2055.49

Table 7: Writing to small and big arrays

1.4.3 Analysis

We can see here that there is a significant difference in runtime when we write to arrays of different sizes.

We could also calculate the time it takes for each memory write:

$$\begin{aligned}
\text{Number of writes/iterations that is made in the program} &= 384 * 1048576 \\
&= 402653184 \\
\text{Time per write for i7-2600 for 6Mb array} &= \frac{520.3 \times 10^6}{402653184} \\
&= 1.292 \text{ ns} \\
\text{Time per write for i5-2400S for 6Mb array} &= \frac{2055.49 \times 10^6}{402653184} \\
&= 5.105 \text{ ns}
\end{aligned}$$

Since the access times are far below that of an access to the memory memory, it is like to be a cache write.

Conclusion: Both CPUs use a write-back policy

2 Optimization of Matrix Multiplication Performance

The default matrix multiplications makes use of 3 while loops accesses the memory without exploiting locality. A few methods have been employed to speed-up the matrix multiplication by exploiting locality.

2.1 Reordering

The original matrix multiplication accesses matrix A in row order and matrix B in column order. After reordering the iterations, both A and B are accessed in row order which is much more cache friendly.

Code Listing 1 Matrix multiplication reordering

```
for (k = 0; k < size; k++) {
    for (i = 0; i < size; i++) {
        temp = a.element[i][k] ;
        for(j = 0; j < size; j++)
            result.element[i][j] += temp * b.element[k][j];
    }
}
```

2.2 Blocking

Even after reordering, row 0 of A will multiply by all rows of B before incrementing row index of A. This means that the whole array of B will be cached before row of A is incremented. Since B is likely to be bigger than the cache, caches misses will happen. To reduce the cache misses, we ensure that A, B, and C are small enough such that all 3 matrices can fit into the cache. To do this we apply the Cannon's algorithm by dividing the matrices into blocks. We have 3 blocks in the cache each with a size of $\sqrt{n/3} * \sqrt{n/3}$ where n is the size of cache. In the program, the size of L1 cache is used (32kb).

Code Listing 2 Cannon's algorithm with reordering

```
for (k_block = 0; k_block < size; k_block += step) {
    k_limit = min(k_block + step, size);
    for (i_block = 0; i_block < size; i_block += step) {
        i_limit = min(i_block + step, size);
        for(j_block = 0; j_block < size; j_block += step) {
            j_limit = min(j_block + step, size);
            for (k = k_block; k < k_limit; k++) {
                for (i = i_block; i < i_limit; i++) {
                    temp = a.element[i][k] ;
                    for(j = j_block; j < j_limit; j++)
                        result.element[i][j] += temp * b.element[k][j];
                }
            }
        }
    }
}
```

There are 3 blocks each time in the cache. The 3 outer loops does the job of dividing the array into smaller blocks.

2.3 SIMD

By making use of SIMD(SSE instructions), we are able to execute multiple operations in the same cycle. However, *SSE3* instructions operate on 128-bit of data in parallel which means that we can only multiply the matrices row wise. To do so we will need to transpose the matrix.

A normal matrix transposition is not cache friendly as it copies the rows from the source matrix into the columns of the target matrix. We could do the transposition in blocks to minimise cache misses.

Code Listing 3 Block Transposition

```
for (i_block = 0; i_block < size; i_block += step) {
    i_limit = min(i_block + step, size);
    for(j_block = 0; j_block < size; j_block += step) {
        j_limit = min(j_block + step, size);
        for (i = i_block; i < i_limit; i++)
            for(j = j_block; j < j_limit; j++)
                b.element[j][i] = a.element[i][j] ;
    }
}
```

Once the matrices are transposed matrix multiplication is done by multiplying row i of A by row j of B to obtain the answer for *C[i][j]*

2.4 Results

Below shows the results of runs on the 4 algorithms used: *Unoptimized*, *Reordered*, *Blocking* and *SIMD*. To reproduce the following results, simply run the program *mm* with the corresponding problem size desired. The time as well as the speedup obtained for all 4 algorithms will be displayed. The program will perform a check to see if the results are the same as the original *Unoptimized* matrix multiplication.

		Matrix Multiplication of of n * n			
		4096	3072	2048	1024
Unoptimized	Run 1	594.936	229.011	58.521	6.885
	Run 2	595.368	226.366	58.587	6.888
	Run 3	595.368	226.863	58.358	6.888
	Average	595.069	227.413	58.489	6.887
Reordered	Run 1	38.76	16.45	4.96	0.52
	Run 2	38.6	16.44	4.95	0.52
	Run 3	38.66	16.44	4.96	0.52
	Average	38.673	16.443	4.957	0.52
	Speedup	15.387	13.830	11.8	13.244
Blocking	Run 1	42.66	17.3	5.23	0.64
	Run 2	42.67	17.29	5.23	0.64
	Run 3	42.61	17.3	5.23	0.64
	Average	42.647	17.297	5.23	0.64
	Speedup	13.953	13.148	11.183	10.760
SIMD	Run 1	19.73	8.24	2.45	0.23
	Run 2	19.79	8.23	2.44	0.22
	Run 3	19.66	8.26	2.44	0.22
	Average	19.727	8.243	2.443	0.223
	Speedup	30.166	27.588	23.938	30.837

Table 8: Runtime(in seconds) for Intel(R) Core(TM) i7-2600

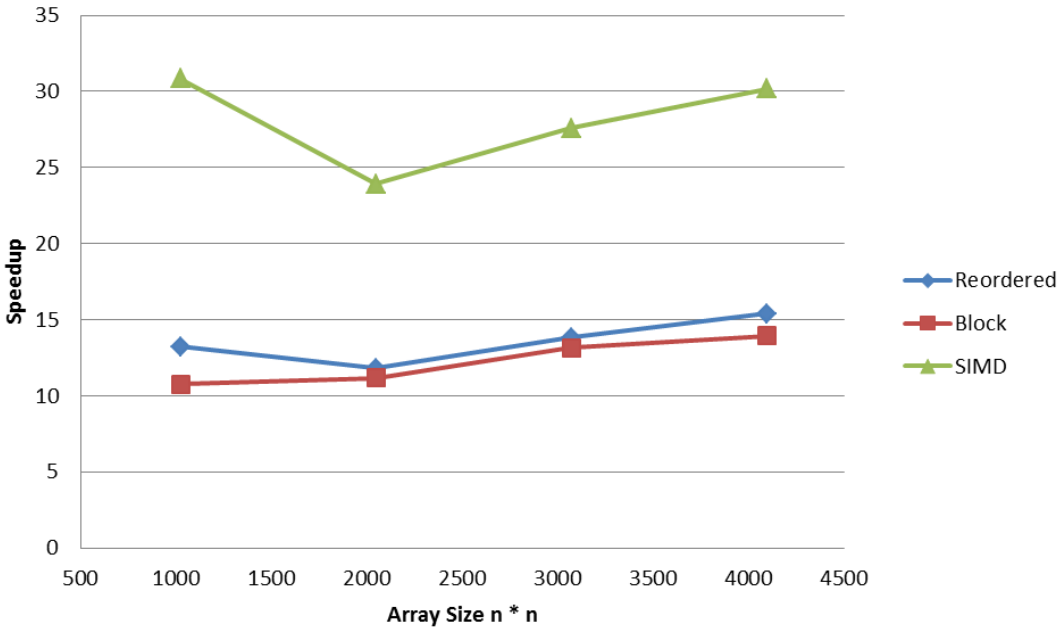

Figure 7: Speedup of Intel(R) Core(TM) i7-2600 for different problem size

		Matrix Multiplication of of n * n			
		4096	3072	2048	1024
Unoptimized	Run 1	698.211	265.255	68.4	7.962
	Run 2	693.43	268.445	68.181	7.982
	Run 3	692.705	268.95	69.854	7.988
	Average	694.782	267.55	68.812	7.977
Reordered	Run 1	62.58	26.99	8.03	0.6
	Run 2	60.02	27.01	8.04	0.6
	Run 3	59.98	27.06	8.00	0.6
	Average	60.86	27.02	8.02	0.6
	Speedup	11.416	9.902	8.576	13.296
Blocking	Run 1	49.33	20.01	6.06	0.74
	Run 2	49.35	20.02	6.05	0.74
	Run 3	49.32	20.03	6.06	0.74
	Average	49.33	20.02	6.057	0.74
	Speedup	14.083	13.364	11.361	10.780
SIMD	Run 1	31.03	13.81	4.1	0.26
	Run 2	31.04	13.82	4.1	0.26
	Run 3	31.06	13.81	4.1	0.26
	Average	31.043	13.813	4.1	0.26
	Speedup	22.381	19.369	16.784	30.682

Table 9: Runtime(in seconds) for Intel(R) Core(TM) i5-2400S

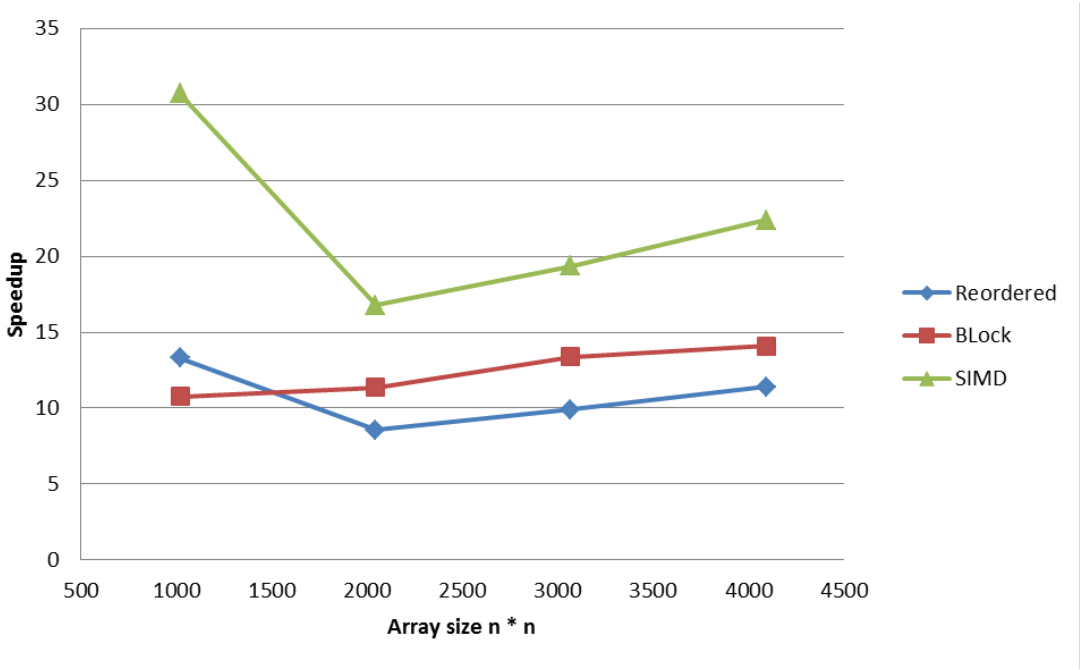

Figure 8: Speedup of Intel(R) Core(TM) i5-2400S for different problem size

2.5 Analysis of speedup

Both CPUs performed well for *SIMD* as well as *Reordered* matrix multiplication for a small array size of $1024 * 1024$. However the performance dips when the array size increases to 2048. Since a matrix of size $1024 * 1024$ is likely to be small enough to fit inside the cache, there is little cache miss and a good speedup can be obtained. A matrix of size $1024 * 1024$ is unable to fit inside the cache, result in a drop in performance.

Since the *Block* matrix multiplication divides the matrix into cache friendly sizes, such a dip is not observed.

The effect of Gustafson's Law clearly demonstrated here. All 3 algorithms show an increasing speedup as the problem size increases (other than the inital dip from 1024 to 2048 due to cache effects).

It is also interesting to note that *Block* algorithm performed better than *Reordered* for Intel(R) Core(TM) i5-2400S. The opposite is true for Intel(R) Core(TM) i7-2600 CPU. Perhaps this may be due to the difference in their cache sizes. Intel(R) Core(TM) i7-2600 CPU has a larger cache size hence there is little performance gains when we perform *Blocking*. Instead the overheads involved mean that *Blocking* is slower than *Reordered*.