COMPSYS 304 Assignment 3

840454023, elee353

Part 1 Cache Measurement

Name of processor

i5-4210M

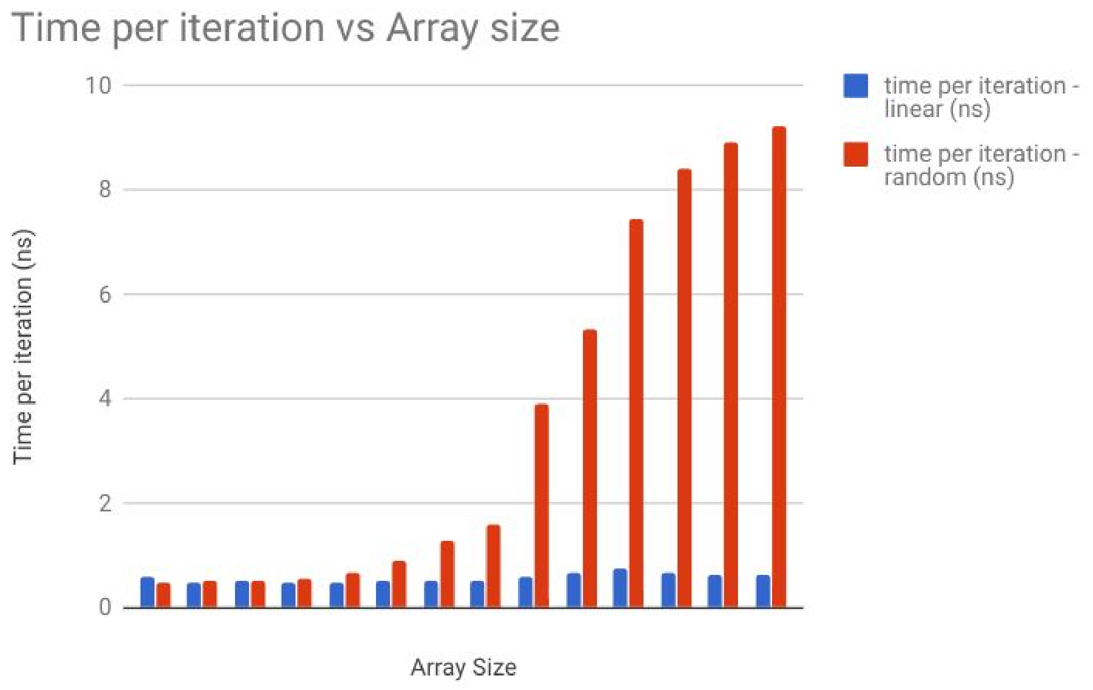
Cache sizes

L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	3072K

Table of measured time per iteration for each case

N	size of a	time per iteration - linear (ns)	time per iteration - random (ns)
2048	8 KB	0.581678	0.491641
4096	16 KB	0.481858	0.524513
8192	32 KB	0.50769	0.519984
16384	64 KB	0.489905	0.569707
32768	128 KB	0.492905	0.656628
65536	256 KB	0.499588	0.912292
131072	512 KB	0.502688	1.277182
262144	1 MB	0.511942	1.608544
524288	2 MB	0.584423	3.896412
1048576	4 MB	0.676256	5.330321
2097152	8 MB	0.738857	7.454531
4194304	16 MB	0.684187	8.410771
8388608	32 MB	0.646589	8.9241
16777216	64 MB	0.650203	9.228647

Chart (time-per-iteration over size of array)



Briefly explain the differences (or similarities) of the results. Explain changes due to the different sizes of a and differences between the two cases.

The time taken for the linear access case is relatively consistent across different array sizes.

On the other hand, the time taken for the random access case is fairly consistent when the size of the array is equal to or below 512 kb. The time taken then increases significantly when the array size grows above 512 kb.

The CPU has a total L1 cache size of 64 kb. When the array size is below 64 kb, the difference in time taken is fairly small. This is because the size of the array is not larger than the L1 cache size of 64 kb. The array elements can be accessed in the L1 cache and there should be no cache misses in this scenario. This means no cache line replacement would happen.

The CPU has a total L2 cache size of 512 kb. When the array size is below 576 kb, the time taken for the random access case takes slightly longer. The array size is larger than the total of L1 cache and the L2 cache of 256 kb is needed. Some array elements are loaded into the L2 cache. The CPU needs to access cache lines in lower level caches when there is a cache miss in higher level caches. The L2 cache is accessed when there is a cache miss in the L1 cache. Having to use the L2 cache increases latency. Furthermore, the bigger the array grows, the more array elements are stored in lower level caches.

Sequential access of array element promotes sequential locality. Sequential locality allows for fast sequential access of the array elements, because the array values are stored closer together. It is very likely that the next few array elements to be accessed will be in that cache line. For random access, the CPU has to spend more time searching for what it wants inside the cache.

The CPU has a total L3 cache size of 3 Mb. When the array size is below 3.576 Mb, the time taken for the random access case takes significantly longer. The array size is larger than the total of L1 and L2 caches and the L3 cache of 3 Mb is needed. Similarly, the CPU needs to access cache lines in lower level caches when there is a cache miss in higher level caches. Having to use the L3 cache increases latency. Cache line replacements take place in this condition. To fetch a new cache line after a cache miss, an existing line must be replaced. This operation increases latency.

When the array size is above 3.576 Mb, the time taken for the random access case takes significantly longer. There are cache misses in the L3 cache. This is because the caches can no longer store the entire array and memory access is required. Memory will be read in memory blocks. Accessing memory has a longer latency than accessing a lower level cache. Due to the random access nature, the random access case requires more frequent memory block reading. Unable to predict spatial locality makes memory management difficult and increases frequency and time taken of memory access.

Part 2 Matrix product

Implementation 1: product of two N × N matrices

time: 5.88 nanoseconds

For matrix multiplication, a row from the first matrix and a column from the other matrix need to be read.
The problematic access pattern is that the second matrix in multiplication is accessed in column major manner. This happens when a column of the second matrix is read. However, since the arrays are stored in row major manner, this makes the multiplication operation inefficient; we need to retrieve the entire row every time when we access a column value. This can be especially inefficient if the multiplied matrix cannot fit into the L3 cache, as we need to retrieve the rows from the memory.

Implementation 2: use of temporary matrix

time: 1.10 nanoseconds

Implementation 3: blocking algorithm

time: 1.16 nanoseconds

The blocking algorithm processes the organize the data structures in a program into large application-level chunks of data called blocks. This improves the temporal locality of inner loops.

The program is structured so that it loads a chunk into the L1 cache first. The program then does all the reads and writes that it needs to on that chunk. It then discards the chunk, loads in the next chunk, and repeats this process until all chunks are processed.

In this implementation, the matrices are partitioned into sub-matrices. These sub-matrices can be manipulated like scalars, multiplied by each other. This improves efficiency because the amount of data that needs to be retrieved at a time is small and cache misses are reduced. Cache misses are reduced as more elements/cache lines are brought into the L1 cache, thereby increasing the chance of the second matrix element existing in the cache.

The best performance takes place when the block size used is closer to the cache line size of the computer that the code is running on. The block size should be a factor of 1000. This enables us to divide the matrix into equal and complete blocks.

The cache line size of this computer is 64 bytes hence the code uses 8 as the block size. 8 double values could fit on each cache line. This makes the block matrix to be 8 * 8 bytes. This allows us to process 8 whole double values at a time for each cache line retrieval. Each cache line can hold 8 double values.

This is efficient because there is no excessive loading into the chunks. This reduces wasting resources to load unnecessary bits and hence reduces the number of cache retrievals. Moreover, it is worth noting that a larger block size would make use of the cache more efficiently. This is because a lower number of blocks in the L1 cache would mean fewer trips to access the lower level caches.