Darcy Cox | dcox740 | 973248899

COMPSYS 304 – Assignment 3

Task 1:

Processor: **Intel Core i7-4558U**

Cache Sizes: **L1:** 128KB, **L2:** 512KB, **L3:** 4MB

The recorded times per iteration for each implementation across varying array sizes is summarized in table 1 and figure 1.

Table 1 - Measured Time Per Iteration for Each Program

N	**Size of array**	**Time per iteration (ns) Case 1 (sequential)**	**Time per iteration (ns) Case 2 (random)**
2048	8 KB	2.44	2.44
4096	16 KB	2.39	2.39
8192	32 KB	2.39	2.44
16384	64 KB	2.39	2.49
32768	128 KB	2.39	2.50
65536	256 KB	2.41	2.73
131072	512 KB	2.38	2.96
262144	1 MB	2.39	3.06
524288	2 MB	2.40	7.31
1048576	4 MB	2.48	12.55
2097152	8 MB	2.54	14.11
4194304	16 MB	2.50	15.23

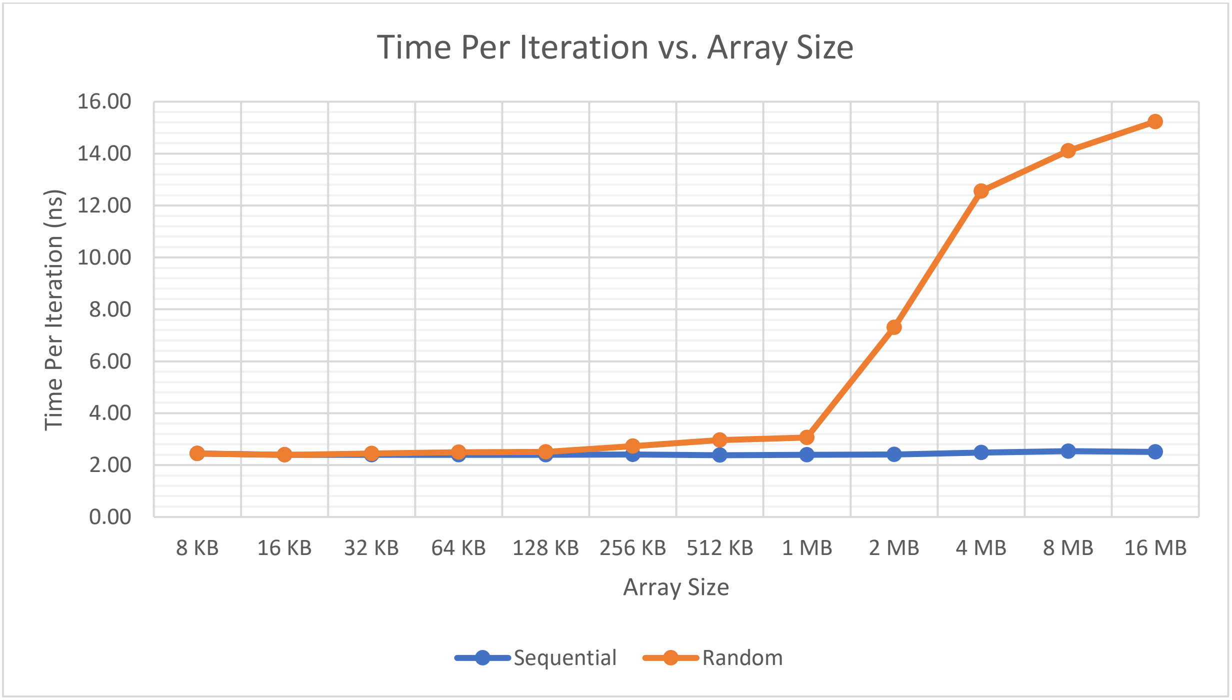

Figure 1 - Visual Representation of Recorded Times

For the sequential implementation, the time per iteration is relatively constant because it accesses contiguous values in memory (i.e. it has good spatial locality). Whenever there is a cache miss, the block of memory containing the requested value is loaded into a cache line and the subsequent memory accesses will use this data straight from the cache. Because we have this uniform access pattern, as the array size grows the cache miss rate remains relatively constant, hence the uniform access time per iteration.

For the random implementation, the time per iteration remains relatively constant until the array size exceeds the L1 cache size, this is because until this point the entire array can fit into the L1 cache, so even though we may have a high miss rate at the start due to the random accesses, the miss rate will be on average the same as the sequential implementation because accesses later on in the running time of the program will read values that were loaded into the cache on a previous cache miss. However, once the array size exceeds the L1 cache size, there will be a higher miss rate because data will be evicted from the cache that is probably needed later, forcing the program to access the L2 cache, which increases the average time per iteration. The same reasoning explains the increase after 512KB (L2 cache size), and 4MB (L3 cache size).

Task Two

1.

   Time taken for straight-forward matrix multiplication: 11.9 seconds.
   The problematic access pattern is the way we access the values of matrix B, where we traverse its columns “vertically”. This is problematic because the spatial locality of this access pattern is poor (B[0][0] is not close to B[1][0] in memory), so we are not using the memory that is loaded into the cache effectively.

2.

   Time taken to create a temporary transposed matrix and use it for multiplication: 3.17 seconds.

3.

   Time taken using blocking: 3.71 seconds
   My approach is to partition the B matrix into k x k blocks and partition the A matrix into 1 x k row slivers. The block size k was chosen to be 64 because since doubles are 8 bytes, we are using 64 x 64 x 8 bytes in the level 1 cache for each block of B, which is less than 128K (size of the L1 cache). Iterating through the matrices, we access B block by block, and perform all the calculations that need to use the entries in the current block. This way, each block of B is only loaded into the cache once and we take better advantage of the cache access time by re-using these values in the cache. The values of the product matrix C are accumulated over several iterations because by partitioning the matrix into blocks we can’t fully calculate the product values in one go, this must be done over several blocks. Unfortunately, this implementation is still slower than matrix2 where the B matrix is transposed first, however this is not by much and there is a significant speed up over the straight-forward matrix1 implementation.