CMPT 295 Assignment 9 (2%)

Submit your solutions by Friday, April 5, 2019 10am.
Remember, when appropriate, to justify your answers.

1. [6 marks] *Cache Particulars*

   (a) [3 marks] Complete the following table:

Cache #	m	C	S	E	B	s	b	t
1	48	64 KB	1		64			
2	40	64 KB			64			32
3		64 KB				12	4	18

   Reminder: 1 KB = 2^{10} bytes.

   (b) [3 marks] Suppose a cache has $(S, E, B, m) = (64, 6, 64, 32)$. Decompose each memory address into its corresponding cache set number, tag bits, and block offset. Express your answers using both binary and hex.

   - `0x6106c8`
   - `0x6216d4`
   - `0x6210c8`

2. [4 marks] *Miss Penalty Hierarchy*

   The Intel Core i7 has an access time of 4 cycles on L1, but 10% of the time it will incur a miss penalty of 10 cycles to interact with L2.

   Should a miss occur on L2 (5% of the time), the penalty will be 50 cycles to interact with L3.

   Misses can occur in L3 as well (1% of the time). The penalty here to access main memory is 200 cycles.

   Thus, a *cold cache* will miss three times, once on each level, and pay $4 + 10 + 50 + 200 = 264$ cycles for the first memory reference.

   (a) [2 marks] On an L3 cache hit, it would take 50 cycles to access a reference to L3, but that only happens 99% of the time. What's the average time to access a reference to L3? Express your answer in cycles.

   (b) [2 marks] What's the average time to access a reference from L1?

over . . .

3. [10 marks] *Matrix Mulitplication*

As a case study in optimizing cache behaviour, you will benchmark several versions of a matrix multiplication algorithm. Consider two matrices, A, B, each matrices of size $N \times N$. Their product, $C = A \cdot B$, is given by

$$c_{ij} = \sum_{k=1}^{N} a_{ik} \cdot b_{kj}$$

i.e., each entry of C is the dot product between a row of A and a column of B.

A direct translation of the equation into C code would give:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        total = 0;
        for (k = 0; k < N; k++) {
            total += A[i][k] * B[k][j];
        }
        C[i][j] = total;
    }
}
```

Observe the innermost loop: the pattern is stride-1 for A, but stride-N for B. The inefficient pattern for B suggests that the code could be optimized for a more cache-friendly pattern, by rearranging the ijk ordering of the loops.

Your textbook has proposed algorithms for all 6 permutations of these loops on p. 645. You are going to benchmark 3 of them, plus another experimental idea.

(a) [6 marks] *Hardcopy:* Within the file `mul.c` there are three different matrix multiplication routines. Using the `time` command, benchmark the three versions for $M = N = 512, 640, 768, 896, 1024$. Record the average user times in a table.

N	t_{alg1}	$\sqrt[3]{t_{alg1}}$	t_{alg2}	$\sqrt[3]{t_{alg2}}$	t_{alg3}	$\sqrt[3]{t_{alg3}}$
512						
640						
768						
896						
1024						

Take the cube root of each average time and plot N vs the cube root of time on a graph. Compute the slope of the best fit line.

When you cube each slope and divide by `NTESTS`, you have a value for the time taken per inner loop. To get the cycles per loop, assume the clock runs at 3 billion cycles per second. Report these values in your table.

(b) [2 marks] *C:* The first algorithm visits the elements of `B` in column-major order which is not cache-friendly. One alternative idea is to take the *transpose* of `B` before multiplying, i.e., to flip `B` along the diagonal. Let `D` be the transpose of `B`, then the innermost loop will be

```
total += A[i][k] * D[j][k];
```

which means that both patterns will be stride-1. Write the C code for this algorithm.

(c) [2 marks] *Hardcopy:* Benchmark your C code like you did for the three others. Add the data to your table, and plot it on your graph. Compute the time and cycles per inner loop.