ECE 565
Fall 2019
Assignment #2

In this assignment, you will work through a few hand-written problems, and will create a few small, targeted test programs that have significant interactions with the cache and memory hierarchy.

For the coding portions, developing and experimenting with the targeted programs will provide an understanding of how the cache/memory interactions can impact performance, and how memory access patterns can be tuned to improve code performance. The should be written in C or C++. For experimentation, you must run on a Linux VM (Ubuntu) from the Duke VCM: https://vcm.duke.edu/. This is also how we will run your submitted code for grading so it must run as it is submitted with no modifications needed

Note: This is an individual assignment. You are free to reuse or adapt any concepts or pieces of the code are posted on our course web page from the in-class examples. But since this is an individual assignment, and so otherwise, all code should be developed on your own. You may reference the internet for help with general C/C++ questions, compile issues, etc. But you should not use or look at code or solutions that may be available online.

Submission instructions: You must follow these instructions and naming conventions, or points will be deducted. Submit one zip file **named hw2.zip** to your individual drop box on Sakai. When unzipped, a folder named **hw2/** should be created with 3 top-level items inside:

1. A PDF file named **hw2_writeup.pdf** that addresses the written portions of the homework (Problems 1-4).
2. A folder named **problem3** that contain all code-related files as described for problem 3.
3. A folder named **problem4** that contains all code-related files as described for problem 4.

Problem 1 (*15 points*)

It may help to refer to the cache background material to solve this problem. Assume that a processor has a single data cache with the following organization:

- Total cache size of 512 bytes (yes, it's very small for the purposes of this example)
- 64B cache blocks
- 2-way set associative
- LRU replacement policy

Also assume the processor references to the following sequence of addresses (shown below in hex):

```
ABCDE  14327  DF148  8F220  CDE4A  1432F  52C22  ABCF2  92DA3  F125C
```

For each reference, show whether the reference will result in a cache hit or miss. And show the final contents of the cache (you may assume the contents of each address match the address itself). You may draw a diagram to show these final cache contents.

Problem 2 (*15 points*)

Assume that a processor has a L1 and L2 data cache, with cache data from main memory. The L2 cache has a hit time of 20 CPU cycles and a miss penalty of 250 CPU cycles (i.e. the main memory access latency). The L1 cache has a hit time of 1 CPU cycle.

a) If the L1 miss rate is 2% and the L2 miss rate is 40% for a given program, what is the average memory access time (AAT) for each memory request from the program.

b) If the L1 miss rate is 10% and the L2 miss rate is 5% for a given program, what is the average memory access time (AAT) for each memory request from the program.

Problem 3 (*40 points*)

In class, we worked through the coding of a program that allowed us to measure the load-to-use latency of a processor's accesses to different levels of cache and memory. In this part of the assignment, your task is to construct a program to measure the data bandwidth that a processor can sustain to the L1 data cache. In other words, how many bytes can the processor read/write per unit of time, when the data is being supplied by L1 cache. You may refer to available information to find the size of the L1 cache of the processor on your VM (recall you can use the /proc/cpuinfo file to find the processor name and search for its specs on the internet, or you can consult the /sys/devices/system/cpu/cpu0/cache/index* files to find the info of each cache). Your program should measure achievable bandwidth to the processor for the following types of access patterns:

1. write traffic only
2. 1:1 read-to-write ratio
3. 2:1 read-to-write ratio

Hints: Like the in-class example, the program should calculate the memory bandwidth it achieves for some number of accesses to a data set of a particular size (e.g. by collecting the time required to complete the number of accesses). Think simply - the code to measure these bandwidths does not need to be complex. Also, to measure bandwidth do you want a dependent sequence of memory accesses, or independent accesses?

(a) Develop a program as outlined above. In the problem3/ submission folder, include 3 items: (1) your final program code, (2) a makefile or build script you used to compile the code, and (don't forget to compile with some optimization(s) like -O2 or -O3), (3) the compiled binary program that you used for testing.

(b) Experiment with the program to determine the bandwidth to the processor from L1 cache for each access pattern.

(c) In the writeup (hw2_writeup.pdf), clearly and concisely describe your program. This would include your methodology, i.e. how your program stresses bandwidth and how does it achieve the 3 access patterns listed above. Also, describe how to run the program, along with any important inputs the program takes. Show and discuss the results obtained from experimenting with your program from part (b). This can include figures and/or tables, as appropriate.

(d) Now change the size of the data structures used by your program such that they should be larger than even the largest cache in your machine. Re-run the code with each of the three access patterns. Summarize these results in the report, and compare them to the L1 bandwidth results. Describe whether they match your expectation.

Problem 4 (*30 points*).

In this part, you will implement and study the performance of matrix multiplication. This matrix multiplication will multiply two matrices A and B into a result matrix C. Each element of the A, B, and C matrices should be a C or C++ "double" type.

```
A[N][N] x B[N][N] = C[N][N]
```

In the program, use a large, fixed value of **N=1024**. Like the examples shown in class, the matrix multiplication should be performed with a triply-nested loop (i, j, k), where each iteration of the inner loop operates on one element of A, B, and C as follows:

```
C[i][j] = A[i][k] x B[k][j]
```

(a) Implement the matrix multiplication program for three orderings of the loop nests: I-J-K, J-K-I, and I-K-J, as covered in class. Don't forget to initialize the elements of matrices A and B in some manner of your choice (i.e. assign them initial values of your choosing)! **This should be done as a single program with a way to control which nest ordering is used.** Have the program record and print the time in seconds to perform the matrix multiplication. In the problem4/ submission folder, include 2 items: (1) your final program code and (2) a makefile or build script you used to compile the program.

(b) Run the program for each of the three loop nest orderings. In the writeup (hw2_report.pdf), record the results you observe. Do these results match what would be expected from the class notes that estimate the number of cache misses per inner loop iteration? Analyze and briefly discuss possible reasons for why they do or do not match.

(c) Add one more variant of matrix multiplication on the I-J-K loop nest ordering to your code. Integrate the Loop Tiling loop transformation approach into your matrix multiplication code, so that multiplication is performed one sub-block at a time across matrices A, B, and C. Choose a block size such that the sub-blocks of matrices A, B, and C will fit within the L2 cache size of your machine (again, you should be able to look this up based on your processor model or the Linux system files).

(d) For the writeup, measure the execution time of the tiled version of matrix multiplication, and compare it to the results from part 2(b). What block size did you choose? Is the tiled version faster than all three different loop nest orderings? Discuss these observations, along with any potential reasons for the measured performance patterns.