CprE 381 Homework 11

[Note: This is your final homework of the semester. You are tasked with answering your curiosity regarding the real cache structures within the computers around you. While this homework will appear short on first glance, it will take some time, some effort, and certainly some thought. However, you will come away with some hard-won real-world understanding of computers. As always, enjoy.]

1. Programs and Caches
    a. Find at least one computer on which you can run the included C programs. While a lab computer or your laptop is fine, if possible, try testing on other platforms. *[In the worst case, you can complete this on the ECpE Linux lab computers.]* Compile the three test programs, run the programs for the following inputs and report their output. *[These inputs are just to help confirm that you can compile and run the examples; they are not the solution to the below problems.]*
    Answers vary by system. Mine are included below.

```
./test1 16 20 8
Total time (s) for j=1 is 0.626241 with 268435456
accesses
Total time (s) for j=2 is 0.611421 with 268435456
accesses
Total time (s) for j=4 is 0.612151 with 268435456
accesses
Total time (s) for j=8 is 0.615271 with 268435456
accesses
Total time (s) for j=16 is 0.610048 with 268435456
accesses
Total time (s) for j=32 is 0.616884 with 268435456
accesses
Total time (s) for j=64 is 0.613726 with 268435456
accesses
Total time (s) for j=128 is 0.619430 with 268435456
accesses
./test2 16 28 8
Total time (s) for j=1 is 1.458907 with 268435456
accesses
Total time (s) for j=2 is 0.934151 with 268435456
accesses
Total time (s) for j=4 is 0.683004 with 268435456
accesses
Total time (s) for j=8 is 0.654984 with 268435456
accesses
Total time (s) for j=16 is 0.638910 with 268435456
accesses
```

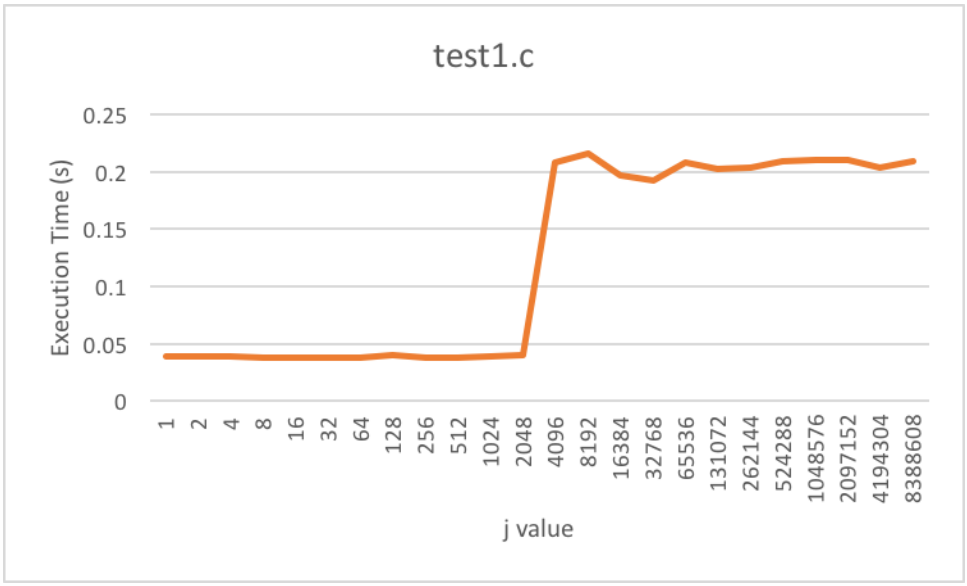
```
Total time (s) for j=32 is 0.676673 with 268435456
accesses
Total time (s) for j=64 is 0.650869 with 268435456
accesses
Total time (s) for j=128 is 0.631908 with 268435456
accesses
./test3 16 20 8 2
Total time (s) for j=1 is 1.986024 with 528482304
accesses
Total time (s) for j=2 is 1.981334 with 528482304
accesses
Total time (s) for j=4 is 1.974685 with 528482304
accesses
Total time (s) for j=8 is 2.048690 with 528482304
accesses
Total time (s) for j=16 is 2.077243 with 528482304
accesses
Total time (s) for j=32 is 2.321620 with 528482304
accesses
Total time (s) for j=64 is 2.020533 with 528482304
accesses
Total time (s) for j=128 is 2.028574 with 528482304
accesses
```

b. Open up the c code files for the tests and look at the meaning of the arguments (they are not the same throughout the tests). Play around with the arguments to get a feel for how they impact the runtime of the core loop in each test. Select a reasonably large array size (likely larger than the size used above) and iteration count that allows the execution time to be roughly 1s on your machine. For test1, run several different values and observe the differing behavior. Try to explain to yourself why this behavior is being observed. Once you can explain the behavior, move on to the following parts. *[There is nothing to turn in for this part, but it is probably the most important part of the HW.]*

   No answer necessary. But in playing around and reading the code I see that I probably want to have a large array of at least several MBs to make sure that no cache will hold the whole array, at least 16 accesses per iteration since that provides a sharp execution time increase (and from my part c reasoning that you need to have more accesses than ways), and iterations counts of 1M, 1M, and 128 for test1, test2, and test3, respectively, so that execution time is large enough to be reliability measured.

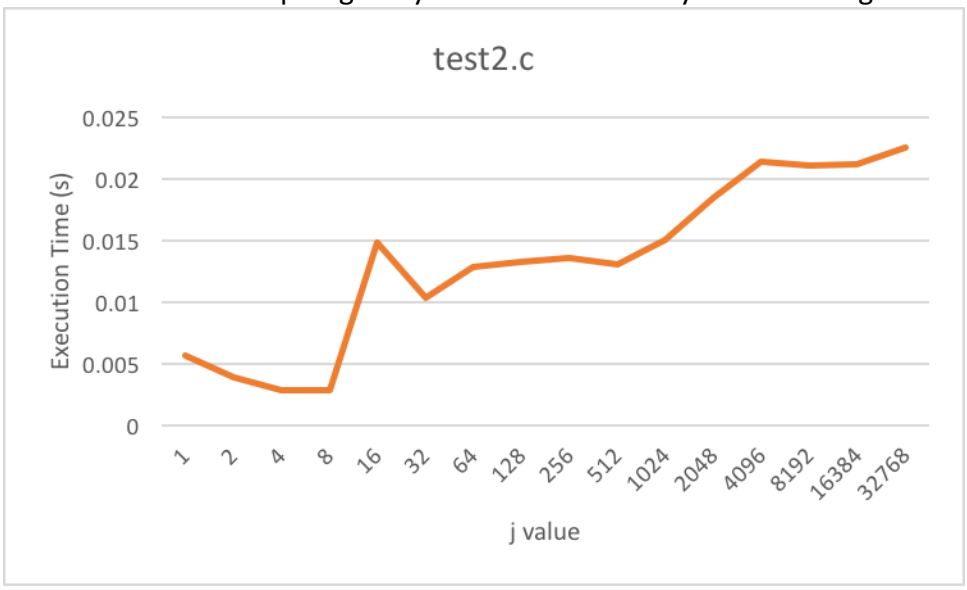
   [Note: there are graphical representations of the ideas below that were presented in lecture and you can see them there.]

c. Make a plot of j vs CPU time from the results of test1.c and report the input arguments you used. Does the processor in your computer/platform have a cache? Use the plot to justify your answer. Specifically point out one piece of sizing information that this plot gives you and write down your reasoning.



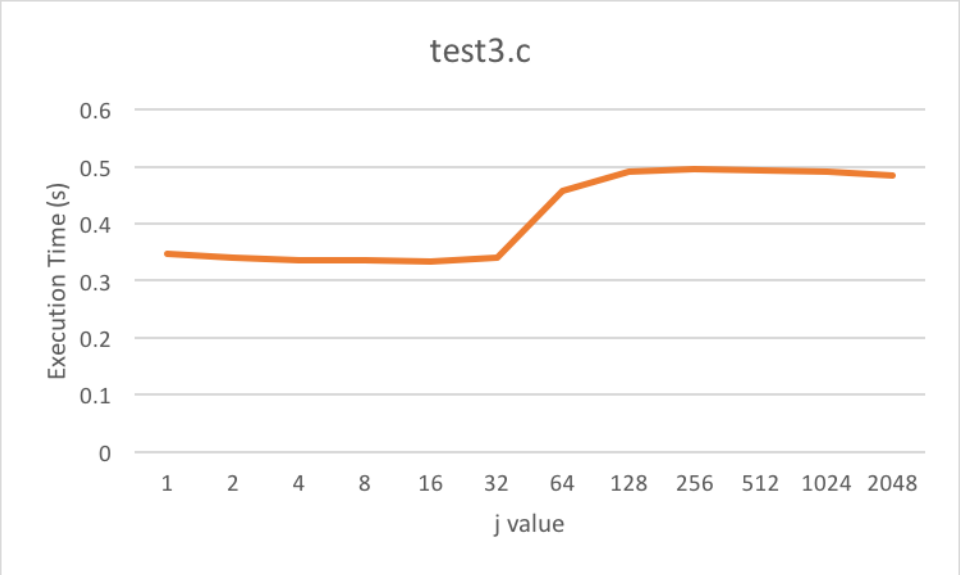
I ran the command "`./test1 28 20 4`" and plotted the results. What I see is that the program strides through my array for multiples of j's value and repeats the same set of accesses many times. The first iteration of accesses will be misses, but all the following iterations will hit in the cache so long as the number of accesses to a single set is less the set associativity (i.e., number of ways in a set). Therefore, as long as the number of accesses per stride size is larger than the number of ways, you start to see conflicts when the stride size is the number of sets times the size of blocks (i.e., the "way size"). In my plot you see the way size is 2^{12} or 4096 bytes or 4KB.

d. Make a plot of j vs CPU time from the results of test2.c where you use the information gained above to select the additional command-line inputs; report the input arguments you used. Specifically point out another piece of sizing information that this plot gives you and write down your reasoning.



Now that I know the "way size" is 4096 bytes, I use this as the last argument per the comments in the program (I ran the command: "`./test2 28 20 12`") and plotted the results. This application intentionally only accesses memory locations in the same set (assuming you correctly determined the way size). The program only makes j accesses per iteration. If j is less than or equal to the number of ways in the set only the accesses during the first of many iterations misses in the cache. However, as soon as j is larger than the number of ways per set, every access is missing in the cache (assuming the replacement policy is something similar to LRU). Therefore, a large increase in execution time indicates that the previous value of j is the number of ways in the cache. In my processor, the L1 data cache appears to be 8-way set associative.

e. Make a plot of j vs CPU time from the results of running test3.c where you use both pieces of information from above to set your additional inputs; report the input arguments you used. Specifically identify another piece of sizing information that this plot gives you and write down your reasoning.



Now that I know the "way size" is 4096 bytes and there are 8 ways, I use these as the last two arguments per the comments in the program and plotted the results (I ran the command: "`./test3 28 7 12 3`"). This program is the hardest to decipher. However, it steps through the array at memory locations that map to the same set (i.e., *ws which is the way size). It completely fills all ways be having i iterate up to number of ways. Then in makes adjacent accesses at stride j. If stride j is within the same block, then there will be no misses, but if it is equal to or larger than the block size, one of the ways will need to be evicted eventually causing a miss during the next iteration. A small nuance is that each during a single iteration (i.e., k loop), we must prepare the sets where stride j accesses may remap once it exceeds the block size. To do this we perform this sequence of writes for more sets than will exist in the cache. Therefore, when we see an increase in execution time, that indicates the block size. Note that the increase is not as dramatic as before since the first inner loop of test3.c is always

missing and the only hits will be in the second inner loop. From the plot we see that blocks are 64 bytes.

f. Finally, put it all together. What is the caching structure of the processor in your computer/platform? Specifically give its total data size, block size, and associativity (i.e., # of ways per set). Then report the specific processor on which you are running (e.g., run "cat /proc/cpuinfo" on a Linux system and report the "model name") and attempt to look up the cache sizes and configurations (specifically look for the L1 data cache). Does this match your experimental observations?

   In summary, I observed an 8-way set associative behavior with 4KB per way resulting in 32KB of total data space (8*4KB). The block size I observed was 64B. My processor was an Intel(R) Core(TM) i7-5650U CPU @ 2.20GHz which was gotten by running "sysctl -n machdep.cpu.brand_string" on the command line in Mac OS X. The L1 cache configuration is a 32KB and 8-way set associative according to http://www.cpu-world.com/CPUs/Core_i7/Intel-Core%20i7-5650U%20Mobile%20processor.html, which lines up with my results.