Varsha Singh

Lab 1, EC527

Due: Wednesday, February 5th

# Lab 1: EC527

Varsha Singh

## Part 1: Memory bandwidth profile of your system

1a. When running just 1 task, the total bandwidth is the same as the "BW/task" (bandwidth per task). What memory size(s) give the highest bandwidth? (If there are several sizes that are close to giving the highest, indicate this.)

- The memory size that gives the highest bandwidth is a memory size of 262144 Bytes. This has a bandwidth of 3.671E+10 Bytes/s.
- There are another two sizes that are fairly close: 1048576 Bytes and 4194304 Bytes, with a bandwidth of 3.632E+10 Bytes/s and 3.562 Bytes/s respectively.

1b. When running just 1 task, what memory size(s) give the lowest bandwidth? What is the ratio between the highest and lowest bandwidths across all the 1-task tests?

- The memory size that gives the lowest bandwidth is a memory size of 16384 Bytes. This has a bandwidth of 2.297E+10 Bytes/s.
- The ratio between the highest and lowest bandwidth across all 1-task tests is 1.598.

1c. How many tasks and what memory size gives the highest total bandwidth? (If there are several combinations that are close to giving the highest, indicate this.)

- There are several combinations that are close to giving the highest total bandwidth, but they are all from 16 tasks. The highest is of memory size 262144 Bytes with 4.425E+11 Bytes/s. This is followed by 16384 Bytes with 4.375E+11 Bytes/s, and 1048576 Bytes with 4.324E+11.

1d. For the largest memory size, look at the bandwidth achieved when running just one task. When using that largest memory size, how many tasks can you run at the same time on this machine, and get close to the same bandwidth per task as what you get with just one task?

- The bandwidth for the largest memory size (67108864 Bytes) is 2.984E+10 Bytes/s. When using this memory size, the closest bandwidth per task is for 2 tasks, with a bandwidth per task of 1.847E+10 Bytes/s. The other number of tasks (4, 8, and 16) have a bandwidth with E+09 Bytes/s, so are not as close in bandwidth per task.

**Supplementary Information:**

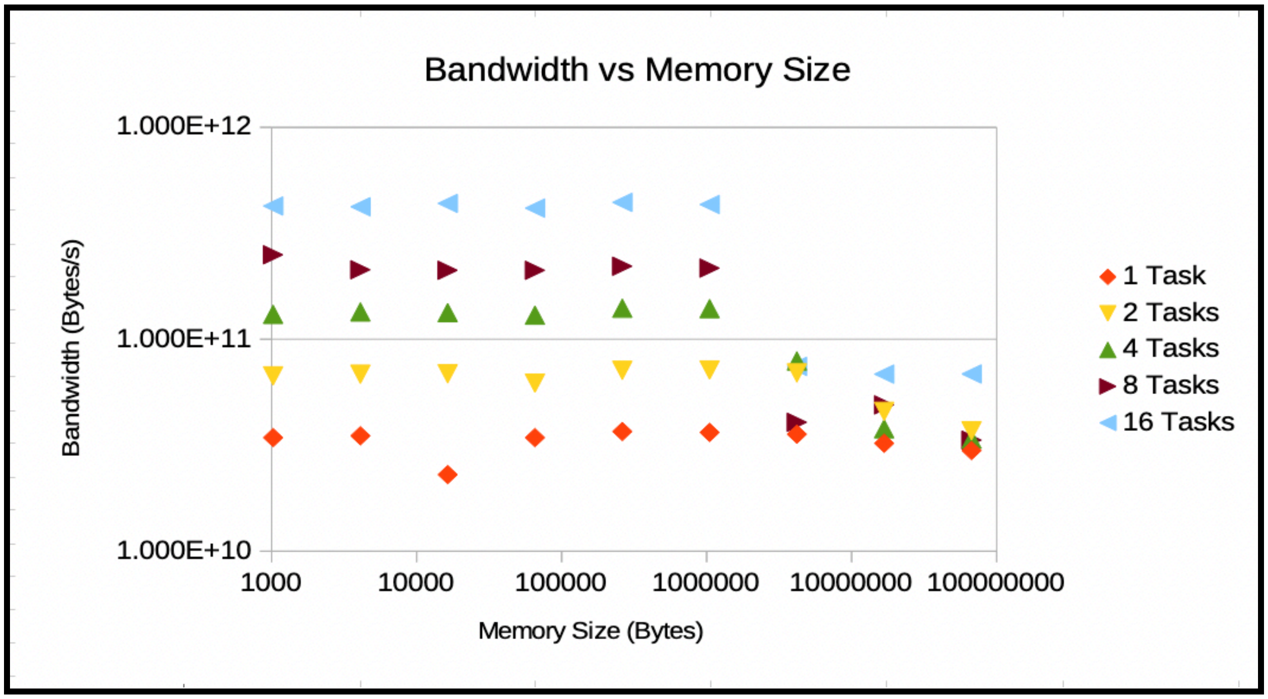


Chart presenting data for Part 1

## Part 2: Testing code transformations – Task: Optimize combining data from 2D arrays

2a. Set A, B, and C so that you test a wide range of array sizes. In particular:

- How big can you make the array before you run into execution problems? (Either it prints an error and won't run at all, or it is hopelessly slow. How do you find how big something is without knowing how big it is? Try successive doubling)
    - The largest I could go before running into an error for being unable to allocate enough storage was a NUM_TESTS value of 160. I was able to run with 80, but 160 gave the following output error: "COULDN'T ALLOCATE 8.720156e+10 BYTES STORAGE (row_len=104404)"
- Which function is faster? Roughly by how much?
    - The ratio of number of cycles per function varies as the number of elements increases. The last, and biggest number of elements I had was 675792016 elements (row_len = 25996) and had *combine2d* to be faster than *combine2d_rev* by a ratio of 1.98310037736737.
- At what array size(s) do the "interesting" things happen in the data? Why? (Something unexpected such as a sudden transition or a change of slope from positive to negative or a zigzag shape).

    - For an array size of 266256 elements (row_len = 516), the speedup of *combine2d* over *combine2d_rev* had a ratio that was less than one – this is the only array size in my experience where *combine2d_rev* had fewer cycles than *combine2d*. This could be happening because of a transition between using different cache sizes, it might be such that a transition to a different level of cache works better with the *combine2d_rev* loop setup, or there could have possibly been a user using the same lab computer that I was using (through SSH), and one of their processes took away computational time from the *combine2d* function.
    - Additionally, it seems that the number of cycles between the two functions is fairly comparable up until an array size of 26173456 elements (row_len = 5116). At this point, it *almost* monotonically increases in the ratio of cycles between *combine2d* and *combine2d_rev*.

2b. Get data for two narrow ranges of array sizes, one small and one large (by adjusting A, B, and C).

- Use the methods you learned in Assignment 0 to find the number of cycles per element for those two ranges for the two different functions (four values in all). Remember that these are 2D arrays: the number of elements is the product of the height and width: compute the "number of elements" correctly.
- Applying your knowledge of the computer, interpret your observations here.
    - The combine2d for the small cycles per element varies with peaks at 3000, 3500, 4000, 4500, and 5000. It also has a very high cycle per element for 2500 – the start of my calculations. I think this might have to do with these values being related to the cache size. Additionally, I believe that there might be some conflict due to not being able to make the range narrow enough. Because I can only adjust the row_len size when adjusting parameters A, B, and C, having the squared value be the number of elements makes the granularity of the number of elements changing harder. There are times where the combine2d_rev has a worse performance, but it is not usual and tends to perform better.
    - For the large narrow range of elements, the combine2d_rev does consistently worse than the combine2d. Combine2d has a pretty stabilized graph, leveling out around 2.7 cycles/element, whereas there is much more fluctuation for combine2d_rev. I think that this has to do with cache size, as there will now be more misses in the combine2d_rev. I also believe that because these are FLOPs, the behavior is much more optimized for larger values. This has the same issue as with the smaller range, but in an increased size of elements.

2c. (If you don't see these effects don't spend too much time, just continue on to the next Part of this lab.) You should find that the graphs of quantities (d) and (e) are particularly interesting.

Zoom in on the region in (d) where it just becomes level and makes a transition. (again, by adjusting A, B, and C).

- I don't believe that this question is actually a question, I think I am supposed to answer 2d and 2e as well as I can.

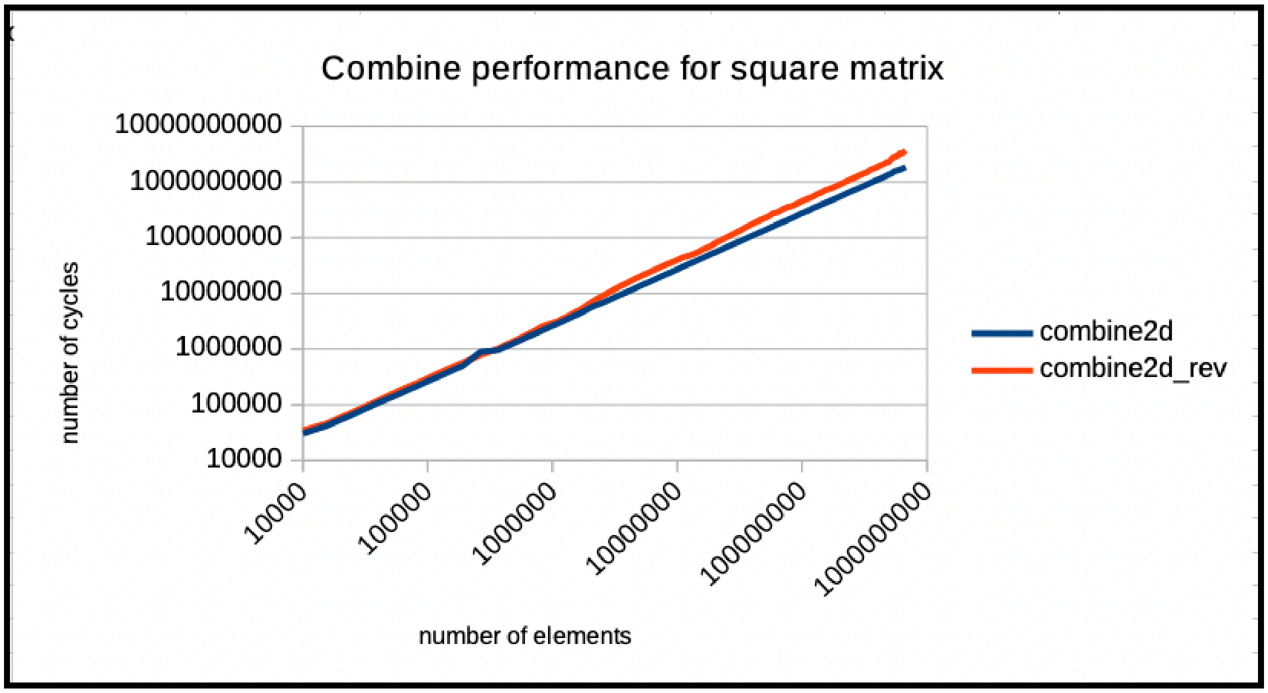
2d. (d) should have a "u" shaped behavior (or "v" or "bathtub" depending on your scales). If you do not see such a thing, make sure you are including data for very small sizes (10x10 array size, or even smaller) as well as large sizes (2000 x 2000 or larger). Explain why performance improves for a while (this is not obvious at all and may require that you run some more tests, including writing some new code). Explain why it gets worse as the size gets very large (this is more obvious).

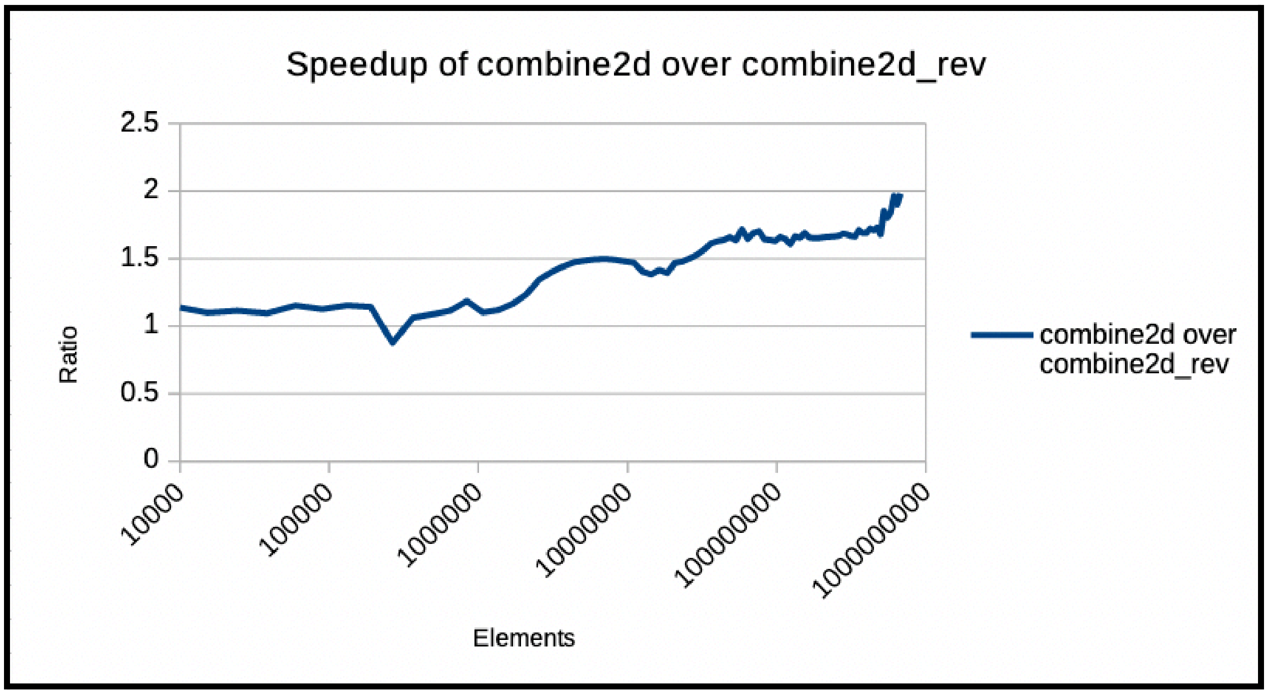
- Having the 'u'-shaped behavior indicates that for small sizes there is a poor performance because of reasons such as overhead dominating, branch predictions, and prefetching not being effective. For mid-sized matrices, there will be better utilization of cache sizes, and memory accesses will be more efficient – this is the bottom of the curve. For larger matrices, the cache will continue to miss and will increase, and the memory bandwidth becomes a bottleneck as there will be consistent waiting for data fetching from main memory. Performance gets worse as the cost for memory increases.

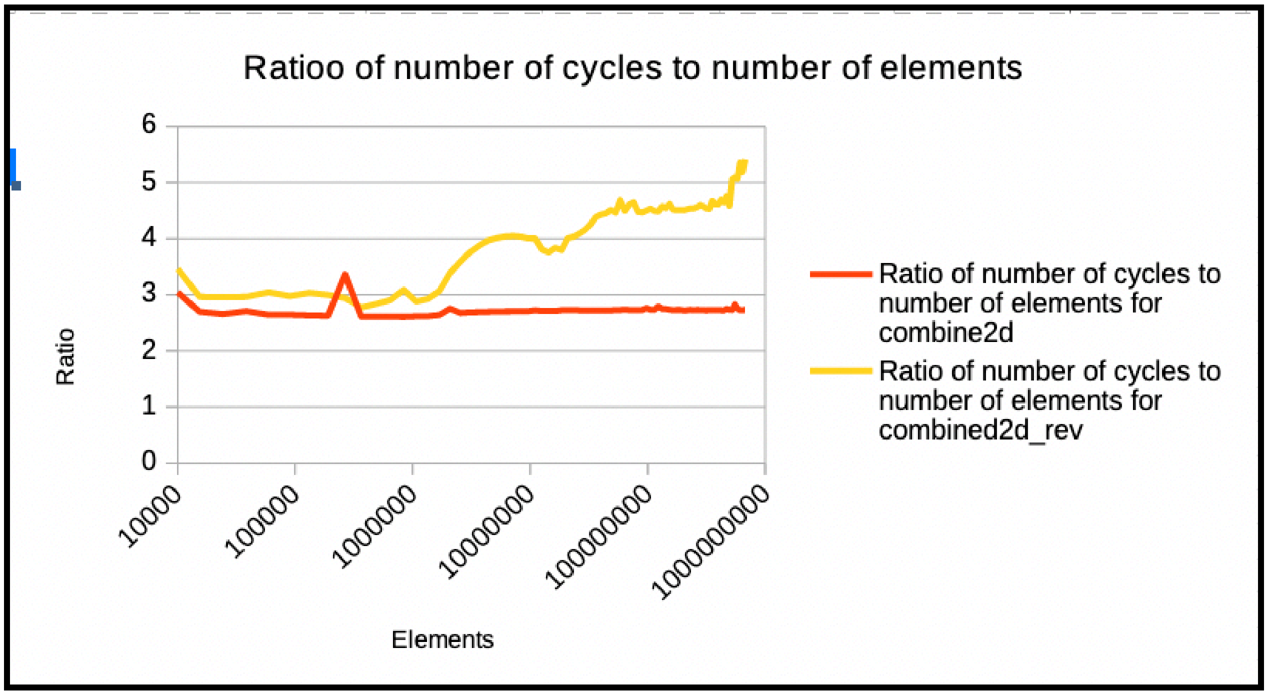
2e. (e) should have behavior that is more "jumpy". If you see a periodic pattern like this: /\/\/\/\/\ or like _/_/_/_/_ it is most probably caused by cache utilization of the program's access pattern, which dependents on (row_length*sizeof(data_t)) modulo (cache_size/associativity), using the stats for whichever level of cache (L1, L2, ...) is not big enough to handle the data access pattern. If the jumpiness annoys you, choose BASE and DELTA in such a way that BASE+i*DELTA is always an odd number. (This works because odd numbers are relatively prime to powers of 2).

- The jumpy behavior is shown in large elements combine2d_rev as well as small elements combine 2d.

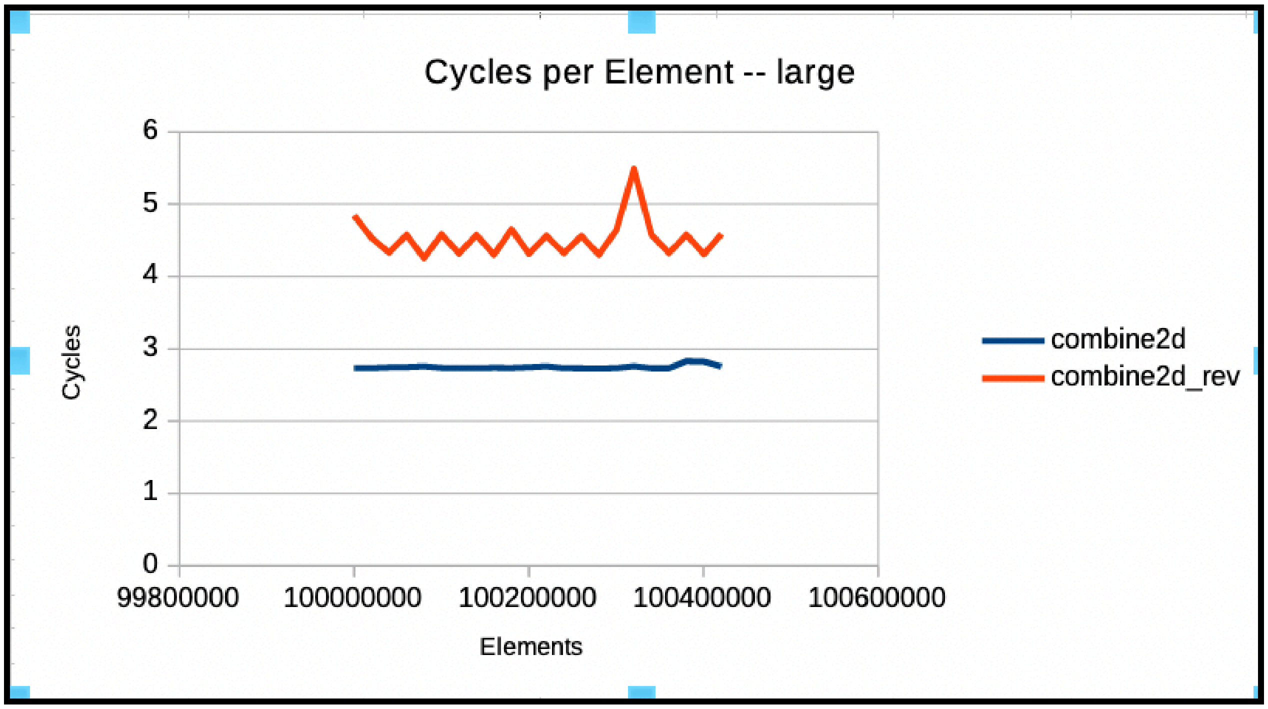
**Supplementary Information:**

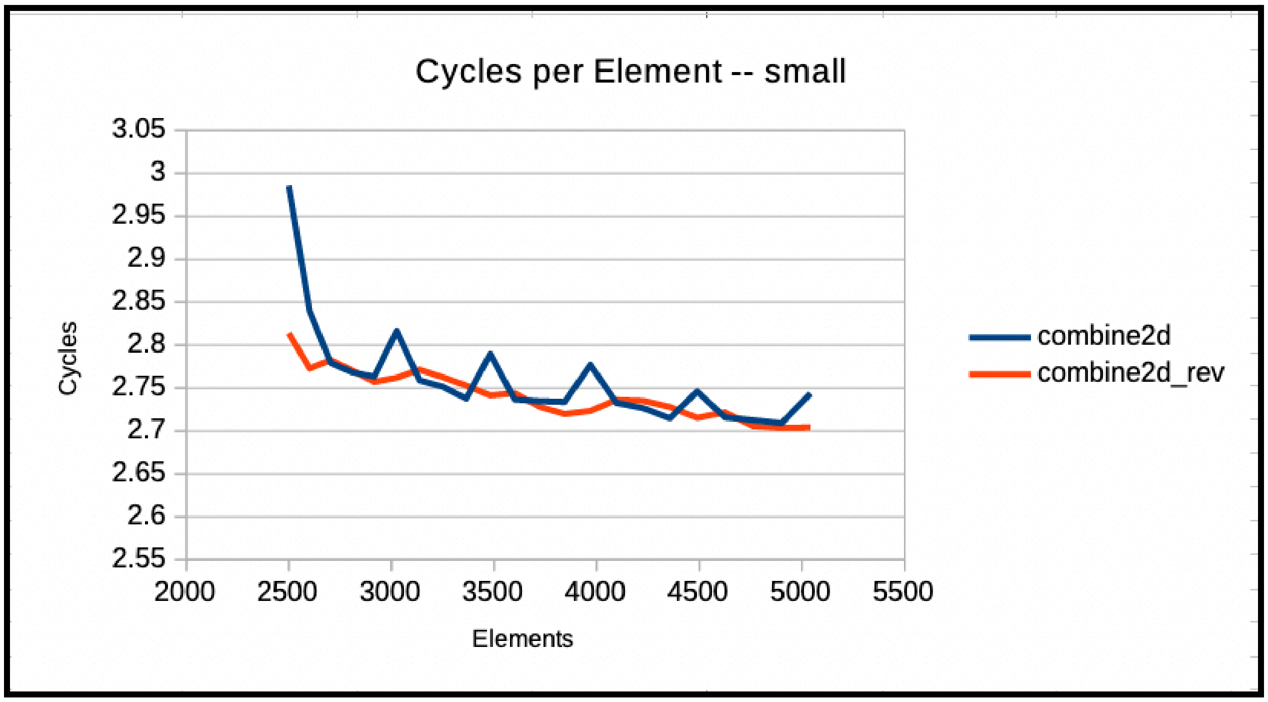

Combine performance for square matrix
number of cycles
10000000000
1000000000
100000000
10000000
1000000
100000
10000
10000
100000
1000000
10000000
100000000
1000000000
number of elements
combine2d
combine2d_rev


Speedup of combine2d over combine2d_rev
Ratio
2.5
2
1.5
1
0.5
0
10000
100000
1000000
10000000
100000000
1000000000
Elements
combine2d over combine2d_rev



2a. Graphs for performance of combine2d compared to combine2d_rev





2b. Performance of small and large matrices

## Part 3: Use these ideas on a more complex code – Task: Optimize MMM using loop interchange

I. How many plateaus are there? To find the answer to this, try a range of matrix sizes from 10x10 going up to at least to 2Kx2K. (If you are wondering "why 2K?", compute how many bytes it takes for three 2Kx2K matrices, and compare to the L3 cache size of your machine.) Combine your data together into a single graph. Set the axis types and scales so that you do not squeeze all the small sizes together into a corner.

II. For each plateau, what is the number of cycles per innermost loop iteration (as defined above)? Run new experiments to determine these, averaging results from several array sizes in the middle of the plateau (no new graph needed for this).

III. Between plateaus are the "transitions". Expressing your answer in terms of the matrix sizes (horizontal axis), where do the transitions occur? You may need to test more sizes to see where each transition starts and ends.

3a. For the first loop permutation (ijk)

1. It looks like there is one plateau. From the range of a 10x10 to a 2Kx2K, there is a region of relatively flat area from 10816 elements to 1340964 elements.
2. For this plateau, I collected data from 11449 elements to 1352569 elements. I found the average number of cycles per innermost loop iteration to be 1.68372593913594.
3. There seems to be a transition into another plateau greater than 2Kx2K, but I did not experiment with this region as I tried to follow the instructions for going up to at least 2Kx2K. There will likely be a transition that plateaus larger than a 2Kx2K, given its relation to the L3 cache of the lab computers.

3b. For the second loop permutation (jki)

1. It looks like there are two plateaus. From the range of 3136 elements to 17956 elements, and another region from 118336 elements to 1132096 elements.
2. For the first plateau, I collected data from 3136 elements to 1759. I found the average number of cycles per innermost loop to be 1.06919758458247. For the second plateau, I collected data from 118336 elements to 1132096 elements. I found the average number of cycles per innermost loop to be 1.84074169260328.
3. There is a transition between these two plateaus, with a sharp increase in cycles per innermost loop iterations between matrix sizes 248x248 and 344x344. I hypothesize that there will be an additional transition into another plateau greater than 2Kx2K, but I did not experiment with this region as I tried to follow the instructions for going up to at least 2Kx2K. There will likely be a transition that plateaus larger than a 2Kx2K, given its relation to the L3 cache of the lab computers.

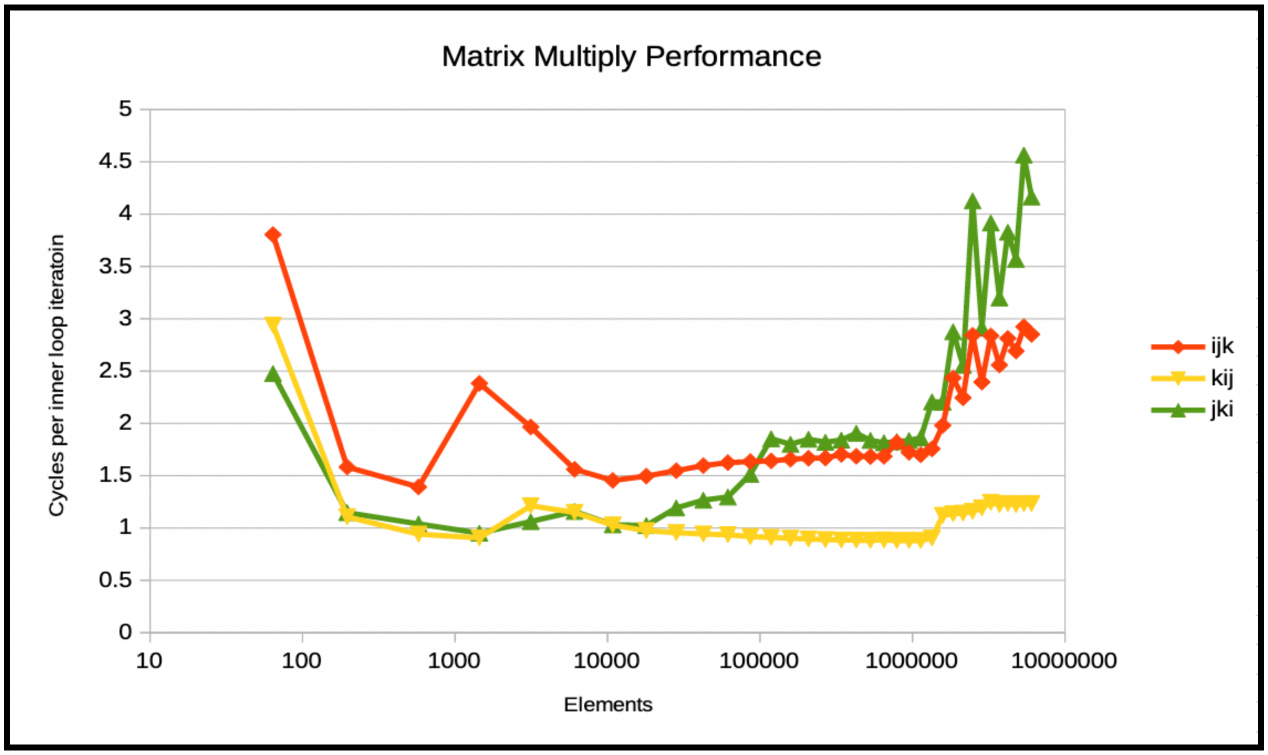
3c. For the third loop permutation (kij)

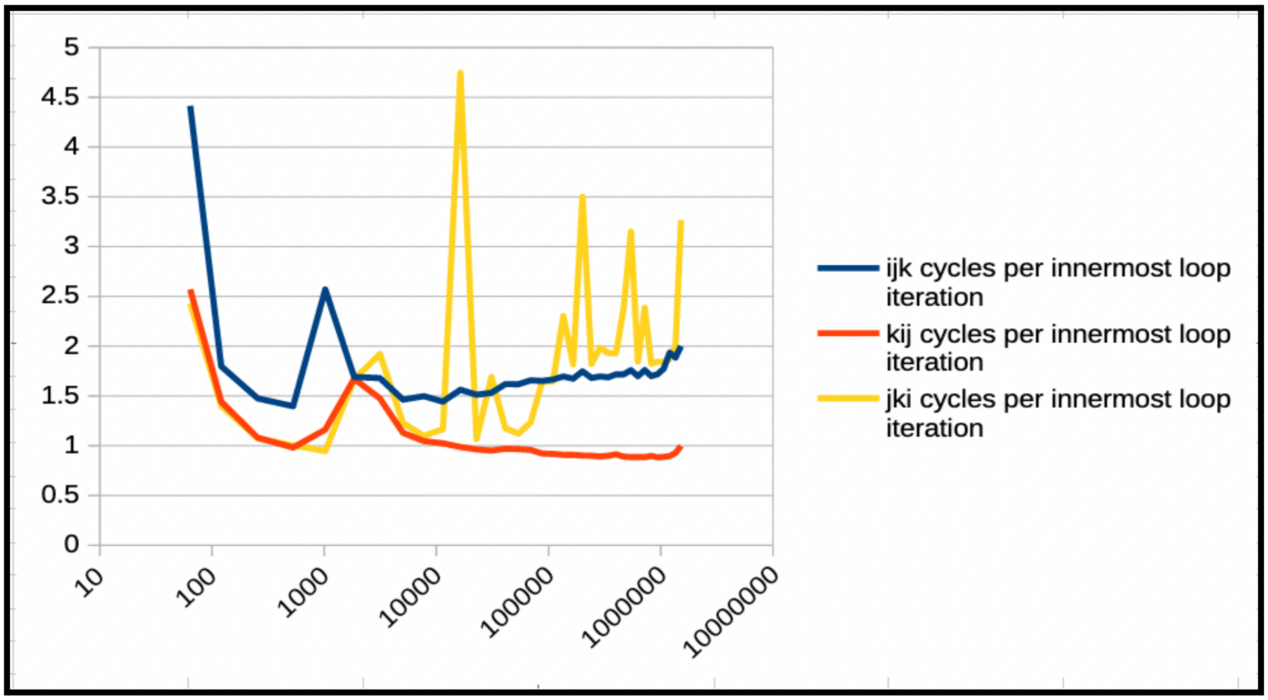
1. It looks like there are two plateaus. From the range of 17956 elements to 1340964 elements, and another region from 1844164 elements to 6031936 elements.
2. For the first plateau, I collected data from 17956 elements to 1340964. I found the average number of cycles per innermost loop to be 0.90610943115514. For the second

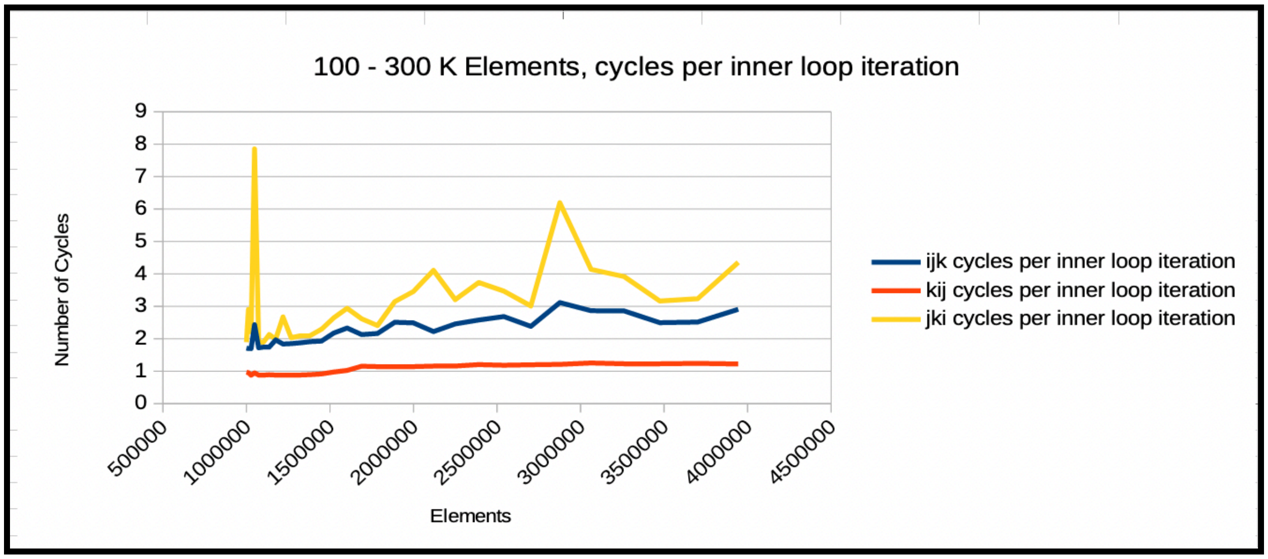
plateau, I collected data from 1844164 elements to 6031936 elements. I found the average number of cycles per innermost loop to be 1.20203670677183

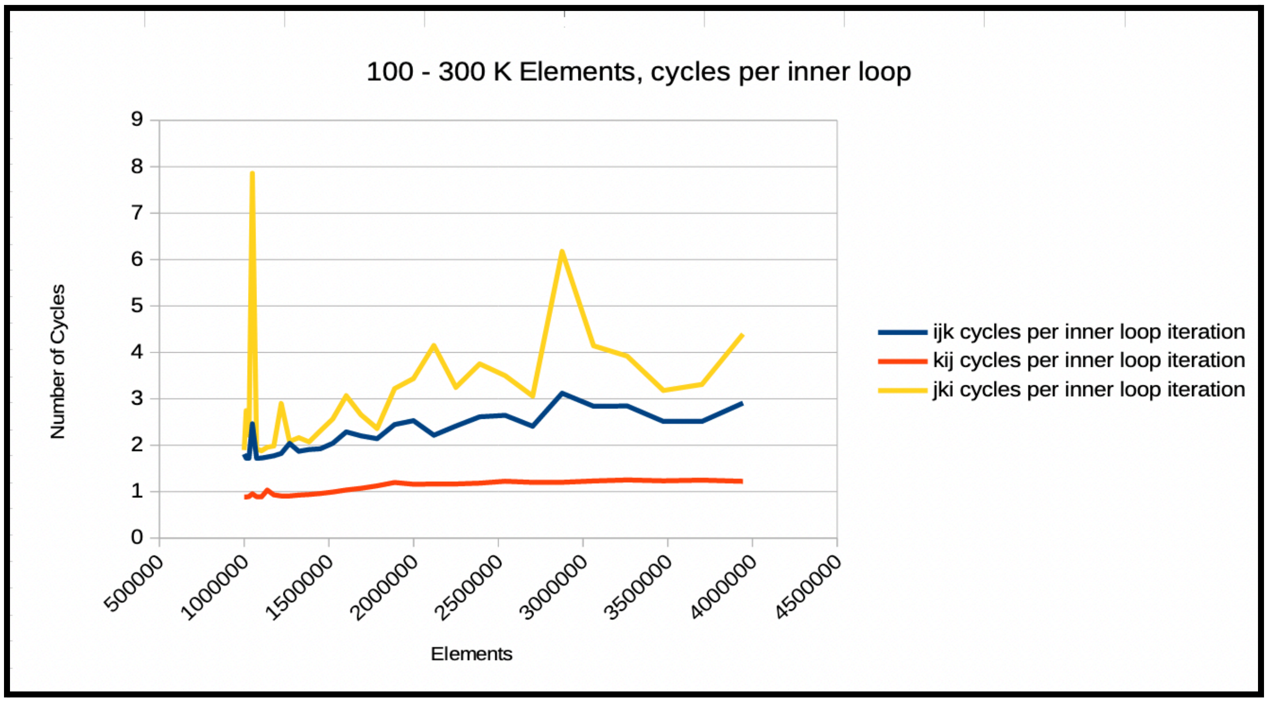
3. There is a transition between these two plateaus, with a sharp increase in cycles per innermost loop iterations between matrix sizes 1158x1158 and 1256x1256. I hypothesize that there will be an additional transition into another plateau greater than 2Kx2K, but I did not experiment with this region as I tried to follow the instructions for going up to at least 2Kx2K. There will likely be a transition that plateaus larger than a 2Kx2K, given its relation to the L3 cache of the lab computers.

**Supplementary Information:**


Matrix Multiply Performance
Cycles per inner loop iteratoin
Elements
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
10
100
1000
10000
100000
1000000
10000000
ijk
kij
jki


0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
10
100
1000
10000
100000
1000000
10000000
ijk cycles per innermost loop iteration
kij cycles per innermost loop iteration
jki cycles per innermost loop iteration





3. All Graphs

## Part 4. Use these ideas on more complex code – Task: Optimize MMM using blocking

4a. The benefit of blocking as a function of matrix size.

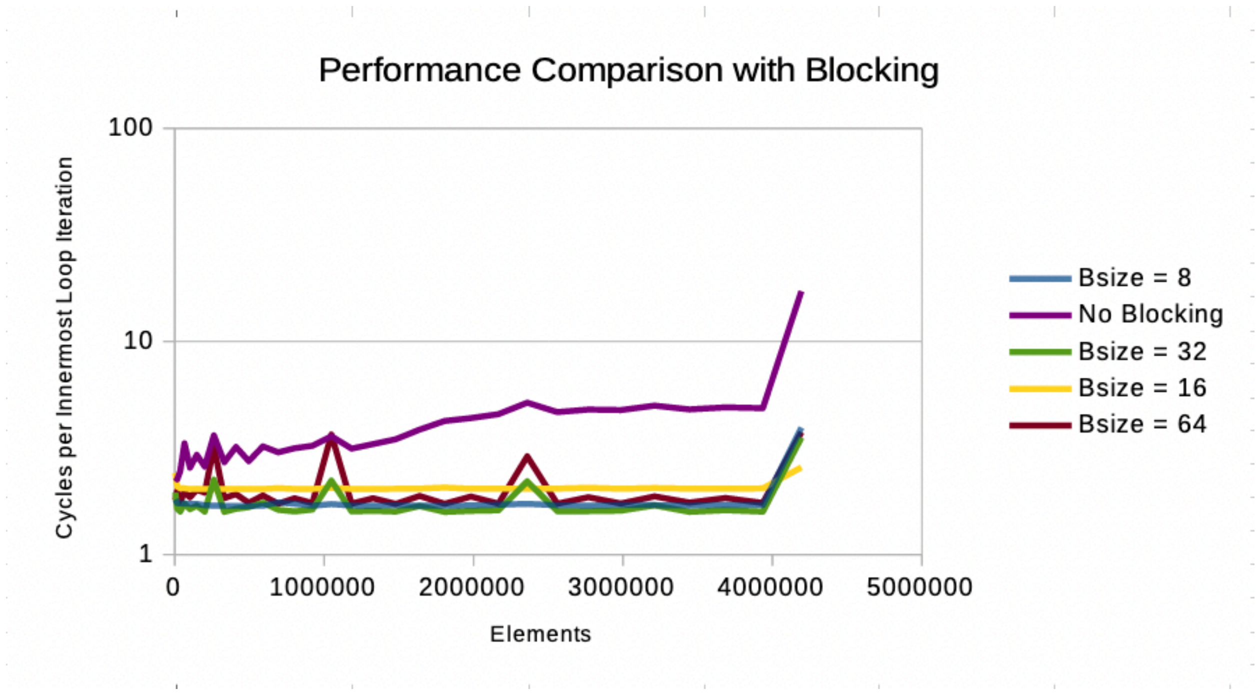
- It appears that the benefit of blocking as a function of matrix size depends on its size relative to cache sizes. While there is an observable performance enhancement for almost every cache size at every matrix size:

- For smaller matrices, the entire matrix might fit into the cache. This would make blocking an unnecessary optimization. It tended that the data relating to the starting of the program was higher in counts of cycles, this could have been related to the power-saving mode and that the wakeup delay was not long enough. Additionally, I tested the sizes uniformly for better comparisons. This meant that the smallest matrix size was 64x64, which does fit in cache. Smaller than this might be better with no blocks, as it might not be feasible for the other blocking methods that were tested out.
- For medium and large sized matrices, it seems that blocking helps optimize cache usage and will improve performance drastically. As the matrix size increases past 2Kx2K, it appears that all of the performances spike, which I believe to be related to the L3 cache utilization, as well as pulling from beyond L3 cache.

4b. The optimal block size as a function of matrix size. It should be sufficient to test exponential variations, e.g., 8, 16, 32, 64, etc. For each block size you try, remember to set your coefficients A, B, and C so that all tested sizes are a multiple of your block size.

- It is hard to get the region in which the unblocked data could be as efficient as Blocking with a size of 8, but it can be assumed that at the very least, matrices sizes 1x1 to 7x7 will better with no blocking (out of the given ranges), as it is assumed that the matrix sizes must be divisible by the block sizes. The chart appears to show that the Blocking method with a bsize of 8 does consistently well, and does not show strong peaks in cycle times as do the other optimization methods for bsize 32 and 64. However, it does look like a **bsize of 32 does on average, better than the other MMM approaches**, with periods of doing worse than bsize of 8 and 16 when cache sizes are filled. At the end of my calculations, further than a 2Kx2K, I find that all of the rates increase, but the bsize of 16 appears to be less impacted and might continue this behavior for larger matrices.

**Supplementary Information:**



4. Performance Comparison with Blocking

Description of experiments: I implemented the blocking code and chose to go from 64x64 to 2048x2048, in steps of 64. I did this to be able to have uniform data that could be compared properly for each block size. The data I recorded was row_len, inner loop iterations, cycle times, elements, and the cycles per inner loop iteration, which is shown on the graph.

**Extra Credit**

4c. For your best MMM, compute the FP utilization.

4d. While the best blocked MMM is probably better than the best non-blocked, the difference might be somewhat less than the analysis predicts. What's wrong with the analysis? How would the application have to change for blocked code to show more benefit?

4e. There is probably an easy way to enhance the distinction between blocked and non-blocked methods. Make the changes and try them out.

## Part 5. Use these ideas on an entirely new application. Task: Optimize Matrix Transpose.

5a. Use test_combine2d.c as a template, create test_transpose.c. Parameterize your code so that you can vary the block size as well as the matrix size

- This is done in the program.

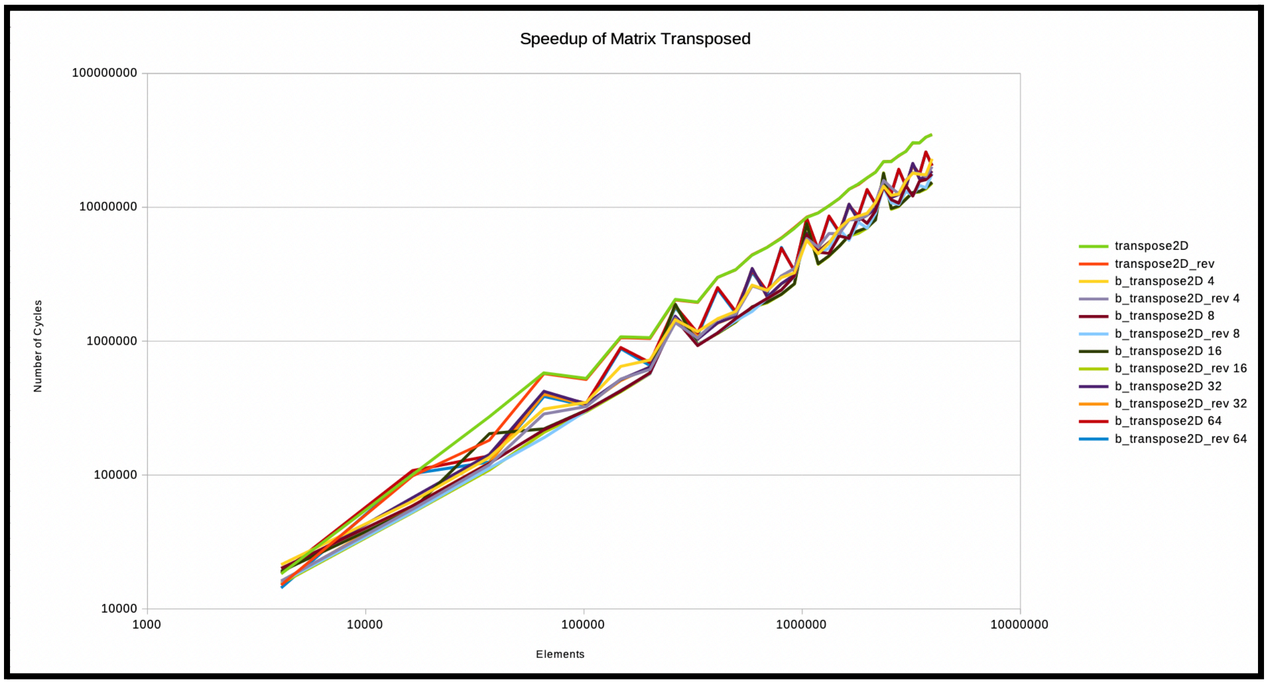
5b. Examine and ij and ji loop permutations over a wide range of matrix sizes as you did for MMM (including small enough to fit in L1 cache, and big enough to not fit in L3 cache).

- L1 cache is 64 KB, which can hold the biggest a 90x90 double matrix. I tested for 64x64 and incremented in steps of row length increasing by 64, to maintain the ability to test all blocking sizes from 4 - 64. L3 cache is 12 MB, which can hold up to a 1254x1254 double matrix. I tested from 64x64 to 1984x1984 sized matrices.
- I found it interesting that the original transpose and transpose_rev perform about the same. There is only slight enhancement in the reverse performance between elements 10000 and 100000. Overall, it seems that the reverse and normal for each blocking size performs about the same, but the spices in which there is a performance decrease happens more frequently with bsize 64 and 32. It seems that the best overall performance is bsize 8 or bsize 16 for the normal transposes.

5c. Block the code. How did you do this? Why should this help? Does this help? For what block sizes?

- I blocked the code by creating an input array and output array for the transposing, and ran the transpose example from B&O pg. 641, as well as the same logic from Part 4, in which I set an outer ii and jj loop that would go through the matrices in blocks instead of the entire row/column.
- This should help because the indices of an array for a transposing are accessed linearly, so working with blocks of it will help increase the rate of cache hits, reducing the access time.
- This does help for most points, there are certain points which I suspect are specific to cache sizes in which the performance of the transpose and its reverse are about the same for most blocking implementations.
- It helps out the most for a block size of 8, but helps out for all cache sizes from 4 to 64. The block size of 8 only appears to have one ‘bump’ in which the performance degrades along with the other sized blocks. At that moment, the best performing block size is a block size of 4. For larger matrices, it appears harder to distinguish between a block size of 8 or 16 performing better.

**Supplementary Information:**



5c. Speedup of matrix transpose

Description: I took the same approach as the previous question, setting my smallest row_len to be 64 and incrementing step size of 64 each time. I recorded data for row_len, inner loop iterations, number of cycles, elements, and the cycles per innerloop iteration.

## Part 6. Quality Control

6a. How long did this take?

- This took fairly long, I think I would estimate around 14 hours.

6b. Did any part take an "unreasonable" amount of time for what it is trying to accomplish?

- I had a pretty difficult time figuring out part 4, as my implementation had a bug. This was a time sink, but once I figured it out and figured a better way of charting my data, it was not necessarily unreasonable.

6c. Are you missing skills needed to carry out this assignment?

- I think I needed a better understanding of the different sizes of cache and that I should review calculating the size of matrices that can fit in the different levels of cache.

6d. Are there problems with the lab?

- I was a little confused about the section marked extra credit in part 4. I talked to Vance and he said that extra credit was from 4c to 4e, and due to time constraints I was not able to get to those sections. Additionally, I asked a clarification question for Part 3, when it asks for the data used to get our answers. I asked Vance if we should include our data in a CSV or if I can provide it in a spreadsheet, and he said a graph would be enough. I am still attaching my spreadsheet that holds all of my data just in case. Also, part 1 did not ask for any deliverables, but I answered the question and provided my data graph.