Arthur Hua, Lukas Chin
EC413
Prof. Herbordt
Lab 8

Task 1: Find machine characteristics

Use the following command to find the basic machine characteristics:

$ less /proc/cpuinfo

What CPU are you using?

Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz

What is the operating frequency of the cores?

799.987 MHz

How many cores are there?

8

Search the web to find out more machine details:

Are all of the cores real?

Yes

How many levels of cache are there and what are the characteristics of each one?

L1 cache: 8 * 32 KB

L2 cache: 8 * 256 KB

L3 cache: 12 MB

What is the microarchitecture of the processor core?

Coffee Lake microarchitecture

What is the maximum memory bandwidth?

41.6 GB/s

Task 2: Practice using timers

Accurate timing is essential to performance evaluation. It is also a (perhaps) surprisingly subtle topic, and getting more so all the time. Use the -O0 optimization setting. Also note the -lrt switch needed to compile clock_gettime().

Perhaps the best way to do timing nowadays is by using clock_gettime(). The program test_clock_gettime.c is a testbench for this function.

Compile the program and execute.

Write dummy code that takes time close to 1 second to execute. How close can you get to it?

What is the resolution?

0.003773951

What is the precision?

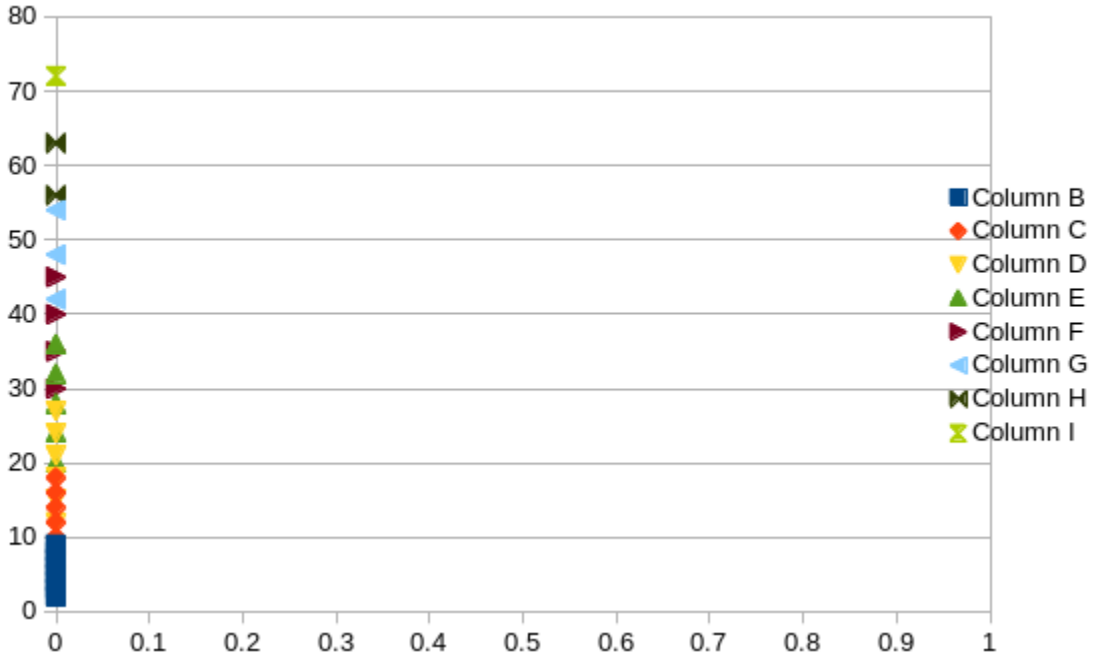
0.001680861

What is the error? (How can you determine the error? There are two answers to this and both are critical.)
You can determine the error by comparing the avg value and the single values and the difference between them.

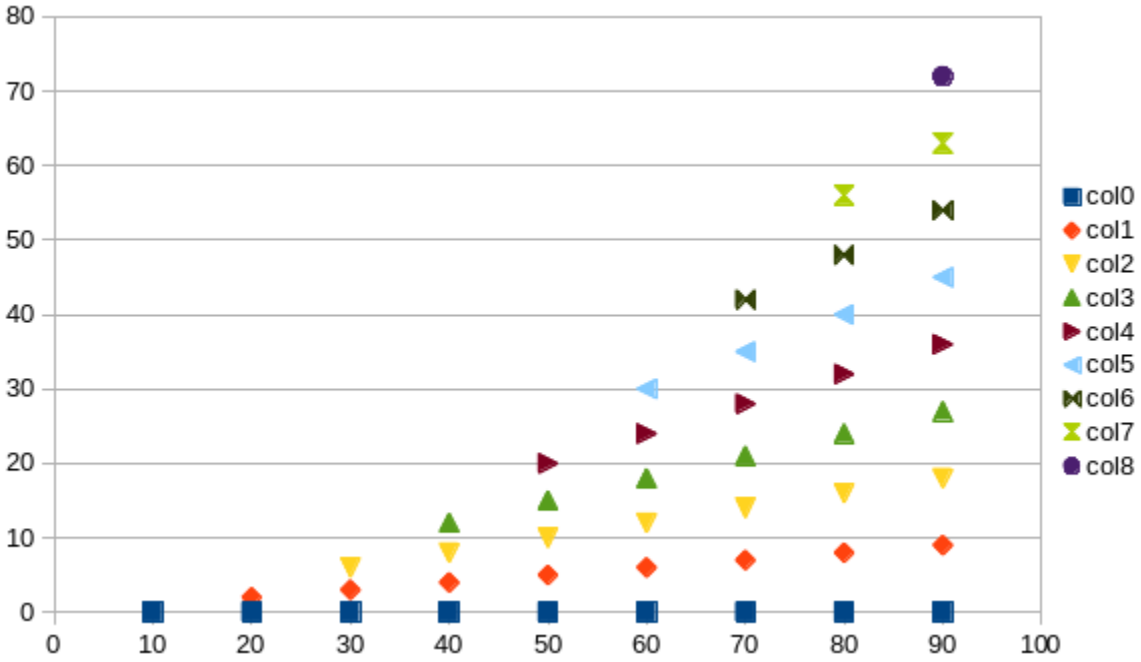
Task 3: Plotting/Graphing Data
Presenting data accurately and concisely is essential in performance evaluation. Most of the time simple
methods will do, once you figure out what you need to display.

• Compile plotting.c
• Run plotting and output data to plotting.csv (./plotting |tee plotting.csv)
• Start OpenOffice Calc (soffice plotting.csv &). Click on "OK" button.

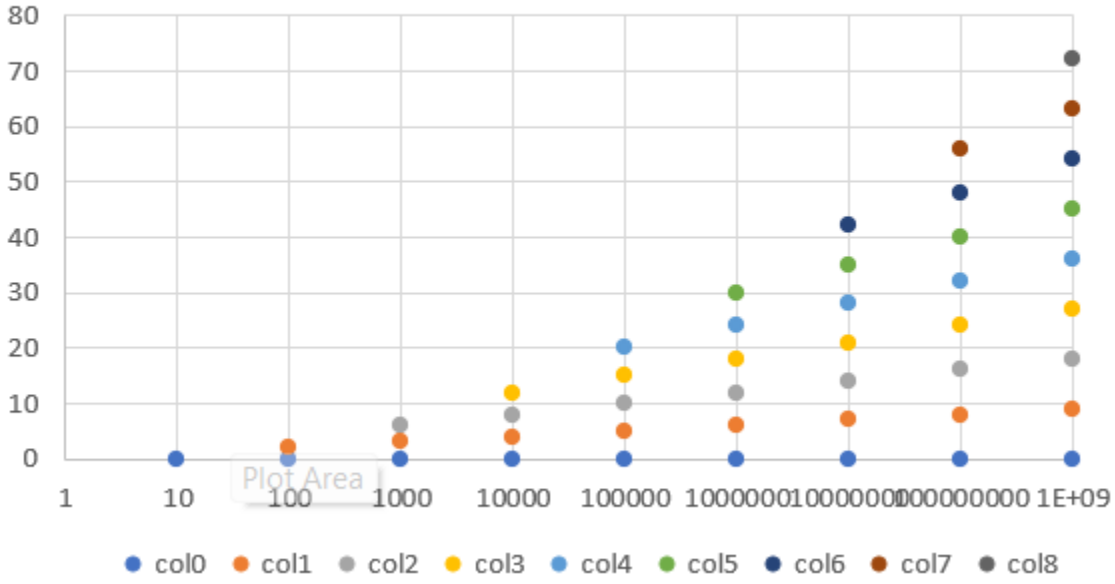
You will notice that there are three separate data collections. Graph them as follows:
1. X-Y plot



2. Scatter plot



3. Scatter plot with X-Axis as log axis



The legend should come up automatically.
Note: You can also copy-and-paste directly from your output device into the spreadsheet.

Task 4: Generating roofline plots (multiparameter performance analysis)

Roofline plots are a nice way of displaying the bottleneck inhibiting performance of a particular

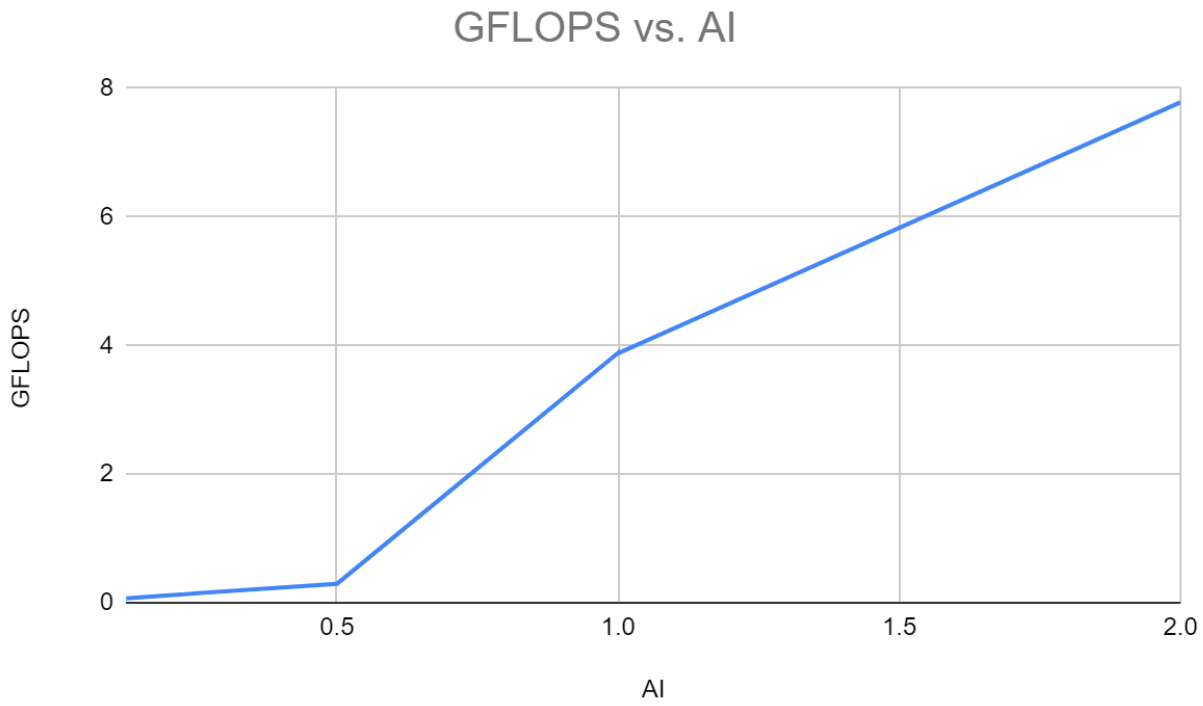
program/computer combination. Good roofline analysis tells the programmer where to put optimization
effort. For example, is the program compute bound or memory bound?

1. Read the section in the textbook on roofline graphs. In it they talk about the stream benchmark, the subject of this part of the lab.
2. Read the stream.c code. There is a lot of detail having to do with reproducibility which you can skip, but you should find the "payload" loops, i.e., the inner loops where the data transfers are being generated.
Are they what you expected? In particular, note that finding the performance of even something as simple as the memory bandwidth is quite complex. The authors of STREAM use four different methods and a variety of parameters. The standard way to interpret the results is to average of the highest values for the four methods.
3. Compile and run. What is the memory BW? How does it match the machine specification you found on-line (Task 1)?

It is 11458.8934 MB/s over the 4 tests, which matches the machines specifications found previously.

Use the stream_simple.c code for the following parts of this task. Start by reading and understanding the code.
4. Adjust the Arithmetic Intensity, recompile and rerun. Do this at least within the range AI = [1/8, 2].
5. Plot your results. You are likely to find regions with different slopes. Your scales and data points should enable you to determine them.

AI	GFLOPS
0.125	0.068841
0.25	0.145675
0.25	0.145798
0.5	0.291625
1	3.885756
1	3.879028
2	7.779959



6. What are your conclusions? What have you found out about Arithmetic Intensity and measured memory bandwidth? What does this say about achieved GFLOPS/s?

Arithmetic Intensity and the achieved GFLOPS/s are linearly correlated. Since there is sufficient BW throughout the tests, raising the AI will not bottleneck the GFLOPS, and that is why the slope remains nearly constant throughout.

Notes: it may be hard to get a good roofline plot. There are several causes for this, but the most likely is that your added complexity gets optimized away by the compiler (thinking it is doing you a favor!).

As an example, compile stream_simple.c with: gcc -O1 -o stream_simple stream_simple.c

Try running a loop that calculates a[j], and a loop that calculates a[j]*a[j]*a[j]*a[j]*a[j]*a[j]*a[j]*a[j]*a[j].

Even though the second loop has 8 times more FLOPs, you should get about the same performance. By compiling the code with the -S flag, you will get an assembly file stream_simple.s. Try searching for "mul"
inside the file. Notice that you will only find one or two multiplications in the whole assembly file! In order to actually run the code you write, disable optimizations by passing the -O0 flag (instead of -O1) to the compiler.

If you look at your new assembly, you should now see 8 multiplications in a row.

Task 5: Exploring the performance effects of temporal and spatial locality in data memory access

In this section, you will use the C program cache_temp.c to investigate the impact of temporal and spatial locality in memory accesses on program performance. The program features a function that loops through an array with references based on a separately stored set of indices. One set is ordered, the other not.

1. Investigate performance for a large dataset with respect to reference order:
- Run the program with a dataset size (N) significantly larger than the L3 cache for both ordered and shuffled indices.
- Record and compare the execution times.

A1: 0.030682031 sec

A2: 0.230897216 sec

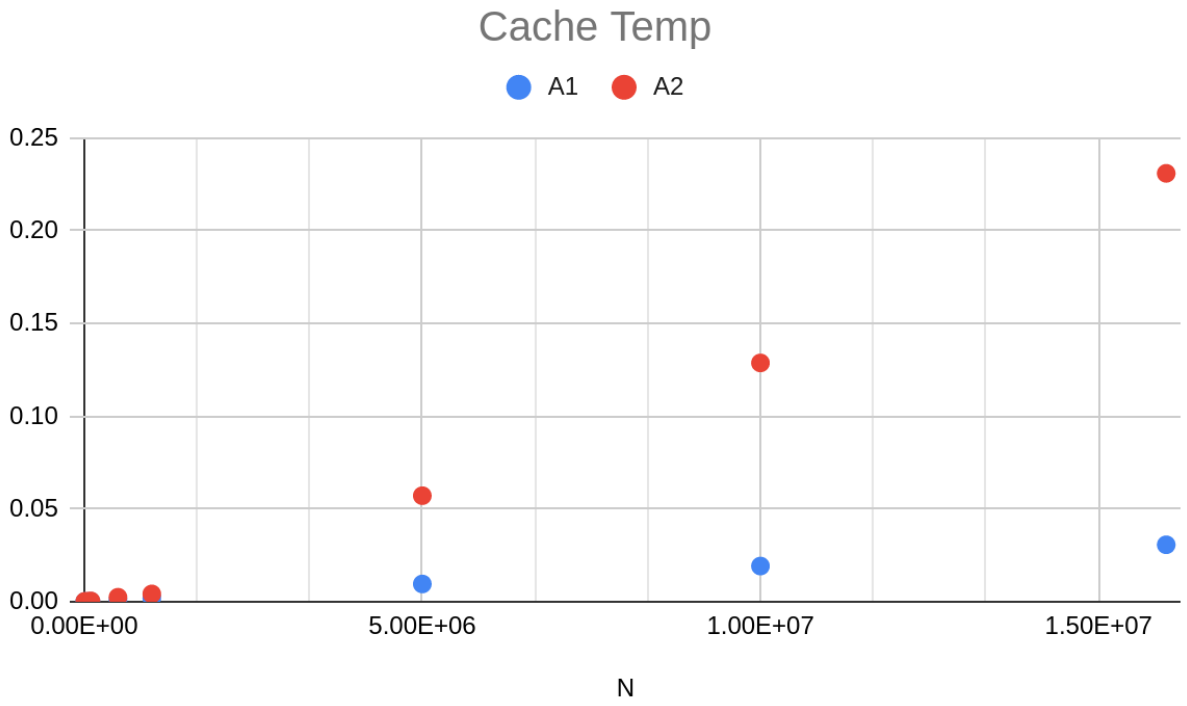
- What kind(s) of locality does this exercise address? What is your analysis? For example, is this what you expected? Why or why not?

Temporal locality because the execution time for both A1 and A2 are similar even though one set of indices is ordered and the other is unordered. This is what we expected, since it reflects the efficiency of the CPU, which is built to retrieve and update data as efficiently as possible.

2. Extend 1 by also varying dataset size:
- For various N values, ranging from smaller than the L1 cache to larger than the L3 cache, run the program with both ordered and shuffled indices.
- Record, plot, and compare the execution times for each N value.
- What kind(s) of locality does this exercise address? What is your analysis?

N	A1	A2
9.00E+03	0.000090074	0.000192289
5.00E+04	0.000090308	0.000183223
1.00E+05	0.000180053	0.00040975
5.00E+05	0.000983785	0.002406553
1.00E+06	0.002013391	0.00418152
5.00E+06	0.009552775	0.057137016
1.00E+07	0.019214312	0.128712566

1.60E+07	0.030682031	0.230897216



Both spatial and temporal locality because we are using different cache sizes, so the data is stored at different places and it changes the cost for accessing the data within the cache. This is why when the cache gets bigger, the time changes more.