Varsha Singh
Lab 0, EC 527
U50072781
Due: 01.29.25

Lab 0: EC527

Part 1. Find machine characteristics

1a. Use the following commands to find the basic machine characteristics:
less /proc/cpuinfo
lscpu
• What CPU are you using?
Model name: Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz

• What is the operating frequency of the cores?
CPU MHz: 3000.000
CPU max MHz: 4700.0000
CPU min MHz: 800.0000

• How many cores are there?
8

1b. Use the information from 1a to search the web to find out more machine details:
• How many levels of cache are there and what are the characteristics of each one? (Self study: do you know what these characteristics all mean?)
In the above model, the cpuinfo states that there are the following:
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 12288K

Wikipedia states that there are 4 levels.
Redis:

- L1 cache is the fastest cache, and also the smallest. It is embedded directly into the CPU, which means that it operates at the same speed as the CPU, and is usually divided into two parts - L1i for storing instructions and L1d for storing data.
- It it misses, the level 2 cache is checked, which is the next fastest and smallest cache, and may be located inside the CPU chip or located outside of it, but it is always closer to the CPU than the main memory. It can be shared among multiple cores or exclusive to a single core
- L3 cache is known as ‘last level cache’, and larger than both L1 and L2, but also slower. Located outside of the CPU and shared by all cores, therefore plays an important role in data shared and inter-core communication

• What is the microarchitecture of the processor cores?
The Intel website says the i7-9700 is formerly Coffee Lake.

• Are all of the cores real? (what does this mean?)

- In this CPU, there are all real cores. There is 1 thread per core, 8 cores per socket, and 1 socket. It also says that there are a total of 8 cores, so both of these indicate that all 8 cores in the CPU are real. The`ht` flag is present in the flags for the 8 cores, but it does not seem that any of them are enabled.
- You can have 'logical' cores if hyperthreading for a core is enabled. This would mean that there is a 'virtual' core, and 1 core can act as more than one core by switching between different tasks concurrently.

• Can you find information online related to the machine's memory bandwidth? If so, what did you find?
(You will use this in part 6.)
On Intel's website, I found that the max memory bandwidth is 41.6 GB/s.

• What is this processor's "Base Frequency" and "Turbo Frequency" in GHz? (You will use this in part 2.)
Base frequency: 3.00 GHz
Turbo Frequency: 4.70 GHz

Part 2. The computer's self-measurement of time

2a. Read notes_timers.txt and test_timers.c. The source file test_timers.c contains three different timing mechanisms. Compile and execute. Observe each method's resolution, precision, and accuracy.
• How can you tell whether the timer is accurate? In its deepest form, this is a fundamental question in physics; no need to go that far!

- There are ways of external validation of a trusted source of a timer. One could try to synchronize their own stopwatch start and end time with the start and end time of the program, but this event will have delays as there is a reaction time to observing a program's start and end time (if done by a human, they will have to see through the script that the program finished running to stop the stopwatch). Scientists determine accuracy by running their results repeatedly against an identified known correct value, seeing how close it can be to being around the same correct value. Without the aid of the un-trusted lab computer, I could determine the accuracy of something that is printed out if I have an idea of what to expect. I can personally determine a time measurement probably in the 1s or 10s magnitude of seconds.

The methods in order of file are:

1. Gettimeofday
   The resolution is in seconds and microseconds since the Epoch. This is OS dependent, as it is a system call handled by the OS. Additionally, the usecs_of_timeval function

multiplies the seconds field by 1e6 and adds the microseconds field to this.
The precision is fairly high. Among 5 runs, the variance is 8.7595931E-5.
The accuracy is flawed, as the given code file is set to different parameters. The obvious one to stand out is the frequency, as it runs at 3 GHz.

2. RDTSC
   The resolution is very high, with the data provided in a number of seconds (with many decimal places) as well as clock cycles.
   The precision is quite low in comparison to get time of day in seconds, with the variance being 0.00026191779.
   The accuracy is flawed, as this computation is divided by the clock rate, which has not been tested to update yet.
3. Times()
   Gets time in clock ticks, and finds the difference in the CPU time charged for the execution of user instructions of the calling process. The reference point for how many clock ticks stays the same since the point in the path does not change from invocation.
   The resolution of the UNIX utility time returns the time in clock ticks, but the get_seconds routine to read time makes it more human-readable, and returns the time in seconds. This makes the resolution quite poor, as there only a difference up to two decimal places between the log-levels of steps.
   The precision is very high, the variance between the most number of steps is 2.0E-5
   The accuracy is possibly flawed, as the computation requires calibration for the number of ticks per second. I have not tested this yet, so there is a chance that it is the same, there is a chance that this variable needs to be different.

2b. The timer that uses RDTSC has some basic problems. Do an internet search to find out what they are.
• What problems do RDTSC-based methods have? Is it still useful? Note: you could solve some of these problems
yourself, but no need to do so for this lab!
• You may wonder what CPU clock frequency to use when converting an RDTSC measurement to seconds. Referring back to part 1b, an Intel processor runs at a really low speed (like 800 MHz) to save power until a program starts running; then if it is cool enough it runs at the "Turbo Frequency" — but only for about 1/10 to 1/4 of a second; and then after that it slows down to the "Base Frequency" or something close to that.

- One of the main issues with RDTSC-based methods is that the measurement reads the CPU time-stamp counter cycle register, and the actual timing is hard to properly measure as the CPU frequency can vary. The operating frequency of a CPU is sensitive, and will vary depending on the temperature, power-saving modes, and even intensity of running processes. The Turbo-mode for Intel kicks in if the temperature of the chip is low enough, and will increase the frequency of the chip to that greater than the base frequency, allowing for a better performance as measured by time. If there is too much difference in frequency than expected, the results will be inaccurate. However, as shown

later in 2C, it does appear to still be fairly accurate, as the frequency averages to about the base frequency for this benchmark to run.

2c. Each timer requires some calibration which may or may not have been done correctly in the source file.
• For each timer, what (potentially) needs to be done to calibrate the timer?
• As necessary, adjust the constants to give correct timing.
(Note that all are trying to print values in units of seconds.)
• Describe how you did this and the modification (if any) you made to test_timers.c

- gettimeofday does not need to be calibrated, GETTOD_TICS is used as a converting factor between usec (microseconds, 1E-6), and seconds. A value of 1000000 will convert usecs to seconds.
- rdtsc needs to be calibrated. CLK_RATE is used to convert cycles to seconds. As based on Part 1, the base frequency of the processor is 3.0 GHZ, and can possibly enter Turbo Mode if the temperature is cool enough. However, I can use RDTSC with a wrapped gettimeofday function to calculate the time elapsed of the program, finding the average frequency. I found the effective frequency to be 2999997823.93137455 HZ, which can be rounded up to 3.0 GHZ.
- times needs to be calibrated, possibly adjusting GET_SECONDS_TICS. Using a similar method as I did to find the effective frequency, I found the average number of ticks a second, which are 99.81249510, rounding to 100. This would be unchanged, then.
  ** The code left for calculating the number of ticks averaged in a second is left in the code, the times calls would be replaced by RDTSC if looking for effective frequency.

2d. Perhaps the best way to do timing nowadays (on our systems) is by using clock_gettime(). The source file test_clock_gettime.c has a partially written testbench for this function. Use the -lrt option when compiling this.

- Nothing to answer for this question

2e. Notice the function “diff()” which takes two timespec arguments and returns a timespec result. You need to add a call to this function, near the end of main(), get the information it returns and print it out, as indicated by the comments. This should be simple, but will also test your basic C knowledge

- Changes documented in the file

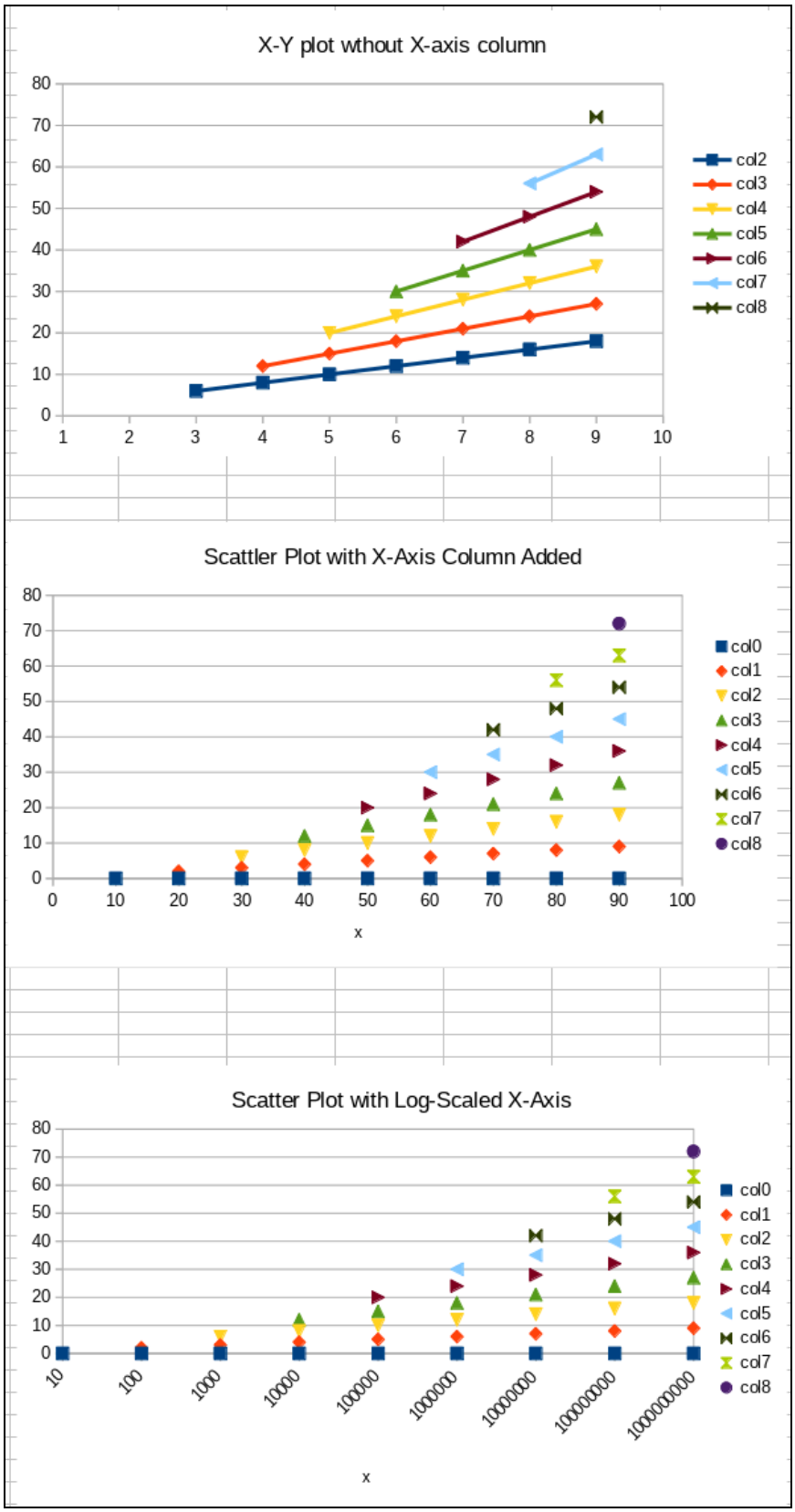
2f. Write dummy code that takes close to 1 second to execute. It won't always take exactly the same amount of time to run, but how close can you get it? Resolution, accuracy, and precision all play a role here: What is the resolution of the measurements given by clock_gettime, and (approximately) what is the standard deviation when you make several repeated attempts? If you do five runs in a row really quickly, remember (as explained in part 2b) that it will probably run a little faster the first time.

- Running the code 5 times gave me the following results (not including the print statement that was calculated)

1. that took 0 sec and 984749276 nsec (0.984749276 sec)
2. that took 0 sec and 981705717 nsec (0.981705717 sec)
3. that took 1 sec and 9813799 nsec (1.009813799 sec)
4. that took 0 sec and 991835912 nsec (0.991835912 sec)
5. that took 0 sec and 985922637 nsec (0.985922637 sec)

- The resolution given by clock_gettime is in nanoseconds.
- The standard deviation (assuming sample instead of population) is: 0.011243925.

Part 3. Plotting/Graphing Data



Part 4. Performance evaluation basics and using Cycles-per-Element (CPE)

4a. Read B&O Chapter 5 intro and Section 5.2. (in BO_chapter5_1.pdf and BO_chapter5_2.pdf)

- Nothing to answer for this question

4b. The example from Chapter 5.2, figure 5.1 (functions psum1and psum2) is provided in test_psum.c. Add timing using clock_gettime() and diff() as in test_clock_gettime.c; plot your results as discussed in the previous sections. The main loops select a variety of sizes to test. You should understand this thoroughly since we will be using similar mechanisms through the semester. Here it does so by evaluating the polynomial Ax2+Bx+C for values of x from 0 to NUM_TESTS-1. Adjust the coefficients A, B and C to make it cover a greater range of sizes. Also, you can increase NUM_TESTSto get more data points. Make sure to test large enough sizes so that your timing measurements are meaningful. Remember what you observed about accuracy and precision in part 2. The textbook only measures sizes up to 200, you will need to go a lot bigger.

- Plot is under 4c

4b. You may notice some anomalies (making it impossible to draw a straight line through all the data points). What's a good way to get rid of them?

- A way that I could get rid of the anomalies would be to change my coefficients for deciding the number of elements to test. I do not think that dropping statistical outliers is the correct approach, as there might be certain piecewise segments that occur when reaching the border between L1 and L2 cache, for example.

4c. As in the textbook example, estimate the cycles per element (CPE) by finding the approximate slope of a line fitting the data points. How many cycles per element does it take on your computer for psum1and psum2? Is it the same as in B&O's Figure 5.2? If not, why might it be different? (A similar figure is in a slide titled "Cycles Per Element (CPE)" in the lecture notes)

- The slope of psum1 is 5.8, and the slope of psum2 is 4.3. These are different from the book, and I believe that this could be the case due to the different ISA used for the book's results and this processor's results. It is possible that the book's utilized processor takes 10 cycles to do an add for an element, whereas a newer processor could take 5.8 cycles to do an add for an element.

Part 5. Interacting with the Compiler

5a. Compile and run test_O_level.c as follows:
gcc -O0 test_O_level.c -o test_O_level time ./test_O_level
Here we are using the shell's time command to measure program run time, so the program doesn't need to measure itself. What gets printed? Pay most attention to the "user:" time.
Starting a loop

done

real 0m0.576s
user 0m0.574s
sys 0m0.001s

5b. Now compile and run test_O_level.c as follows:
gcc -O1 test_O_level.c -o test_O_level time ./test_O_level
What gets printed? You should see that the "user:" execution time has gone down nearly to 0 seconds.

Starting a loop

done

real 0m0.050s
user 0m0.047s
sys 0m0.000s

5c. This time compile test_O_level.c as follows to generate the assembly language code:
gcc -O0 -S test_O_level.c
Look at test_O_level.s and find the loop. Don't panic! Look for the two "call puts" or "callq _printf" lines which are theprintf() statements. Variables like "quasi_random" and "i" are at offsets from rbp, like -8(%rbp) or -24(%rbp). There should be some kind of "mul" and "sub" instruction inside the loop. Rename the file to save it.

- Nothing to answer for this question

5d. Now generate the assembly language code of test_O_level.c with basic optimization:
gcc -O1 -S test_O_level.c
Look at test_O_level.s again and find the loop. What do you see? How does it compare with the -O0 version?

- The O1 version is much shorter than the O0 version, and there are a few less mov instructions. I think that this has to do because the loop variable is not used later in the function whatsoever. There is a difference between the two, as one does a comparison for the loop boundary in O0 vs 01, which moves the loop variable into register eax without actually looping through it.

5e. You should have found something strange. Perhaps using the variable "quasi_random" will solve the problem (how?).
• Try this: modify test_O_level.c so that "quasi_random" gets printed out after the loop is done (it is a double, so use
the %f format specifier).
• Compile again with "-O1". How much time does the code take to execute now?
• Compile again with "-O1" and "-S" and look at test_O_level.s again. What is happening? That is, how has the code been optimized?

- The time it takes is now:
  real 0m0.365s
  user 0m0.363s
  sys 0m0.000s

It looks like the optimized version got rid of unnecessary stack frame operations, and might have done some work with the loop, as there is no longer a comparison cmpq instruction in L2 as there is in O0 optimization.

Part 6. Generating roofline plots

Recall that 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?

6a. Read the section in the textbook called "The Roofline Model" (Patterson & Hennesey section 6.10 in PH_6_p2.pdf). In it they talk about the stream benchmark (for example in the caption to figure 6.18). The stream benchmark is the subject of this part of the lab. Think about how such a benchmark might operate (what work does "Kernel 1" and "Kernel 2" try to do? How might they differ?).

- The Stream benchmark is run to find the memory roofline. The memory ceiling requires running experiments on each computer to determine the gap between them. Because of this, I believe that it will require a good bit of memory to run in order to test memory bandwidth, but should not attempt to be computationally intensive. Therefore, the Kernels might have to do simple operations on large arrays, testing how quickly data can be transferred between the memory and the processor with different levels of arithmetic complexity. For the figure 6.18, Kernel 1 is bounded by memory bandwidth, while Kernel 2 is computation limited, so Kernel 1 would be doing a simple computation while Kernel 2 might be doing something more computationally complex, taking more time to perform the computation than it takes to transfer data.

6b. 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 imagined while you were reading the P&H section? 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 the highest values for the four methods.

- I do not believe I expected this to be the way to check for memory bandwidth, but while reading the chapter I did not expect the computations to be expensive and complex, as those will take longer to produce results and not necessarily a test of memory bandwidth. So, scalar multiplications and additions do not seem out of the ordinary for what I had expected.

6c. Compile and run. What is the memory bandwidth? How does it compare to any memory bandwidth information you found on-line (Part 1)? We will re-visit this question in the next assignment.

- I ran this code with no optimization directives, the memory bandwidth for the best run of the four functions are:

  Function Rate (MB/s) Avg time Min time Max time
  Copy: 37701.6090 0.0001 0.0001 0.0001
  Scale: 43862.0026 0.0001 0.0001 0.0001
  Add: 51098.1198 0.0001 0.0001 0.0001
  Triad: 46603.3778 0.0001 0.0001 0.0002

  Average MB/s over the four tests:
  44816.2773
  == 44.8 GB/s
  The website for this processor claims that the maximum max memory bandwidth is 41.6 GB/s. This is more than expected, and I believe that this is the case because I did not edit the file for calibration.

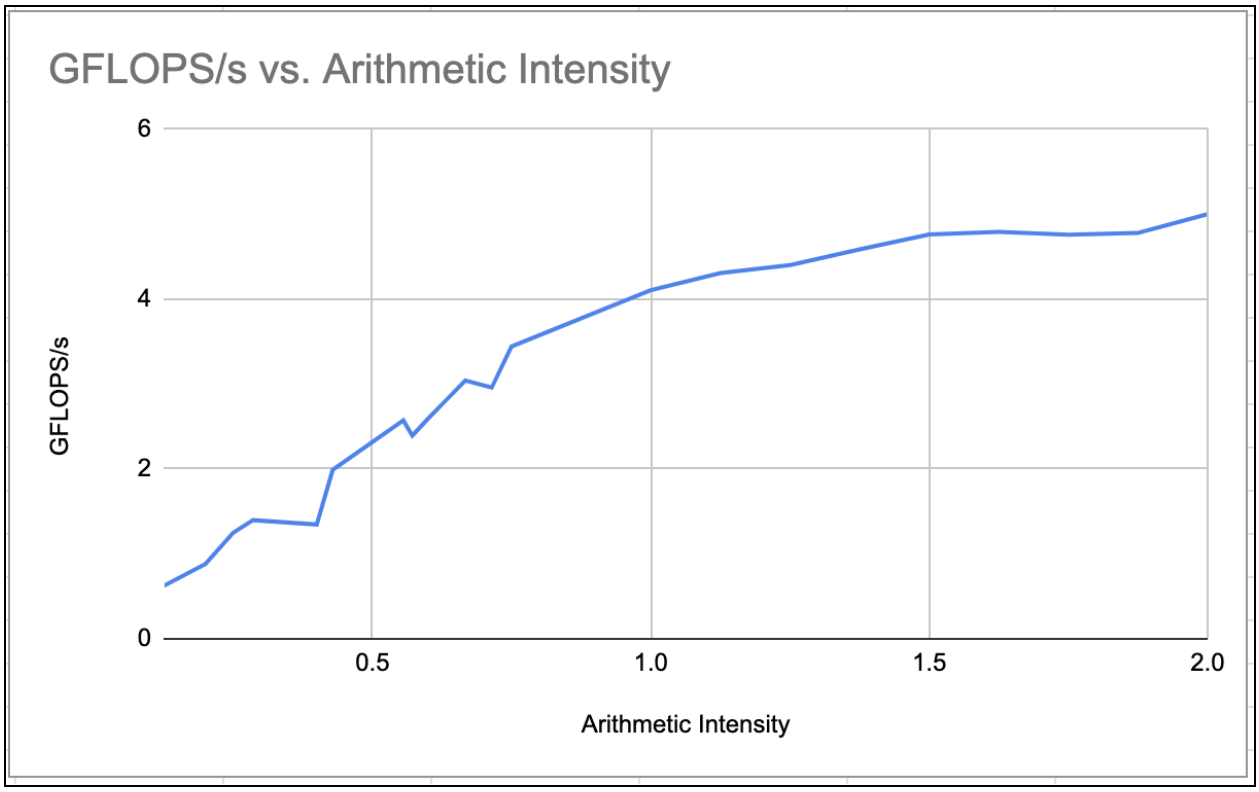
6d. Adjust the Arithmetic Intensity by modifying all of the lines marked "//Modify", then recompile and rerun. For example, the given code has 2.0 floating-point operations (FLOPs) per loop and 8.0 Unique Reads per loop. There are also some integer additions, and a type conversion from integer to float — but those are not FLOPs. The following corresponds to 4.0 floating-point operations (FLOPs) per loop and 2.0 Unique Reads per loop:

```
#define FLOPs_per_Loop 4.0
#define Unique_Reads_per_Loop 2.0
b[j] = quasi_random * in[j] + in[j] * in[j] + (float)(integer[0][j]);
```

When changing the third line (the one with the code) you must also change the first two lines to agree with whatever change you made. Otherwise the program will just calculate incorrectly and print a useless result. This example is a ratio of 4.0/2.0 (FLOPs per Unique Read). Repeat this to get data points for ratios covering the range [1/8, 8].

- The AI is edited in the file

6e. Plot your results. You are likely to find regions with different slopes. Your scales and data points should enable you to determine them.



6f. What are your conclusions? What have you found out about Arithmetic Intensity and measured memory bandwidth? What does this say about achieved GFLOPS/s (billions of floating-point operations per second)?
Note: 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!).

Try making an example that does not use quasi_random or any integer[][] values, but is multiplying together nine copies of in[j](which takes eight multiplications, one unique read).
Compile stream_simple.c with:
gcc -O1 -S stream_simple.c
By compiling the code with the –S flag, you will get an assembly file stream_simple.s. Try searching for “mul” inside the
file. You should see 8 multiplications in a row somewhere.
Turn in your plots and answers to all questions.

- On the graph, the Memory Bandwidth limited region is from an arithmetic intensity of [0.125, 1.5], whereas the Computation limited region is from that point onwards. There is a slight increase in GFLOP/s from 1.875 to 2.0 AI, but it appears to level out from 1.5 onwards. My conclusion is that with a smaller AI, the performance appears to be Memory Bandwidth limited, but with an AI of 1.5 onwards, we reach peak floating-point performance and become limited by the computation. Going from that, the roofline model shows that if achieved GFLOPS/s is lower and scales with memory bandwidth, it is

memory bound, whereas if achieved GFLOPS/s is near the peak it will be compute-bound.

- For AI 8 and it is in[j] multiplied 8 times (8 multiplications and one unique read), the GFLOP/s is 5.000903.
- ** A discrepancy in plotting is explained in Part7, but I think that the reading the AI 8 with a larger GFLOP/s value than the leveled-off section will indicate a second memory-constrained area and another computation-constrained area – this could be due to a larger memory size, such as L1 and L2 cache.

Part 7. Quality Control

7a. Are you missing skills needed to carry out this assignment?

- I felt like I was not missing any skills needed to carry out this assignment. It took a very long time, but everything was able to be done. I had to reference the man pages as well as the IEEE site for the `times()` function, but it was not something I felt required more time in lecture to cover.

7b. How long did this take (hours)?

- I think this took 25+ hours, but this mostly came about because there were times where I was researching different subtopics (the assembly code instructions, roofline plots, man pages, figuring out how to determine effective frequency and ticks in a second, researching for part 1, graph plotting, etc). Additionally, there were times where I did not make substantial progress because I was distracted.

7c. Did any part take an “unreasonable” amount of time for what it is trying to accomplish?

- I think that it took the longest amount of time to complete Part 2. This was because I felt very unsure about proceeding without a clear understanding of finding the frequency to properly move forwards with the lab. I found a method that worked for me, and once I did this, I moved fairly quickly.

7d. Are there problems with the lab?

- There were discrepancies between the code instructions and the lab instructions. For example, in Part 6: the stream_simple.c file recommends a range of [⅛ to 2] for the arithmetic intensity, but the lab manual asks for a range from [⅛ to 8]. I talked to one of the TAs (Austin) about his implementation and he said that I would not have to go past an AI of 2 to find the roofline pattern, so I went ahead and limited my range from [⅛ to 2]. This was the only major one, though.