Joshua Stern
U51857986
EC 527 Herbordt
Lab 0

I used the workstations in the Photonics 307 laboratory, and used the same compilation instructions that were provided at the top of all the .c files.

1.

- CPU name: Intel Core i5-4570S Haswell Quad-Core
- Operating frequency = 2.9 GHz
- Number of cores = 4
- All the cores are real because there is no hyper-threading.
- There are 3 levels of cache. L1 cache has 64 KB per core. L2 cache has 256 KB per core. L3 has between 2-40 MB.
- Architecture: x86_64

2.

- I can determine the accuracy by simulating the same code multiple times and using different functions for recording the time. Then I can compare the results, and this will allow me to determine the accuracy of the different time functions. The best way to determine accuracy would be with nanosecond resolution.

- The problem with RDTSC is that it is not reliable when it comes to providing accurate time stamps. This is because in multicore systems, the timing amongst the different cores is not always synchronized, thus preventing RDTSC from providing accurate results. Also, the CPU might throttle, so a CPU clock might mean more time in that case. RDTSC is still useful for bench marking if used on non-trivial code executions.

- In order to calibrate the timers, I would have to alter either the clock rate for the RDSTC function, the tics per second for the time function, or the cycle rate for the gettimeofday function. The modification I made was increasing the clock rate for the RDTSC function from 2.0 GHz to 2.9 GHz, matching the clock rate of the processor. By doing so, I was able to have the RDTSC provide much more accurate results that we extremely close to the results from the other two time functions.

- By creating a for loop that incremented an integer 500,000,000 times, I was able to get the execution time to 1.002218546 seconds.
    - The resolution is 10 digits, or nanosecond resolution.
    - The percent error = 0.22%

I ran several trials in order to calculate an overall error and a variance.

Trial Number	Error (%)
1	0.22
2	0.33

3	0.31
4	0.83
5	1.18
6	0.38
7	0.18
8	1.25
9	0.86
10	1.46

Standard Deviation = 0.0060
Variance = 3.6080 E-5
Overall Error = 0.70% ± 0.006

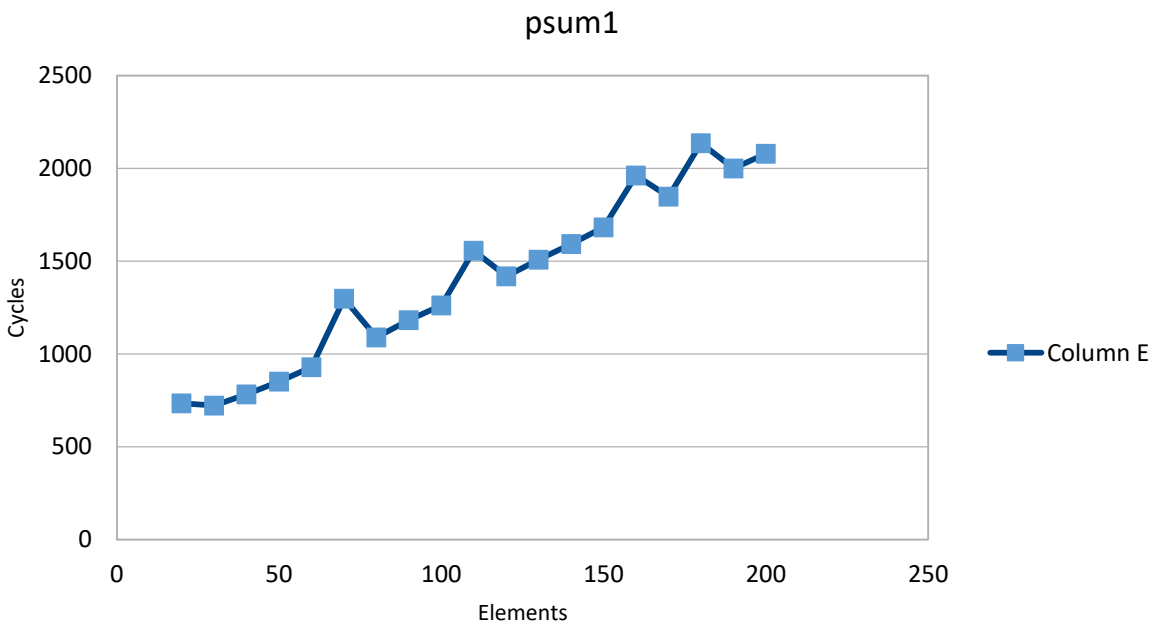
******My new test_clock_gettime.c was attached to my submission*******

4.
There is an anomaly in the first line of output. Sometimes when running a program, issues arise when we first start program. The cause of this could be that the program is running at different clock frequency or waiting on data. In regards to cache, when we first start a program, the cache is empty, so we need to access DRAM (this is also known as a cold miss). In every following iteration of the loop, the data is in the cache so the program runs faster. A good way to get rid of this is to run the program multiple times.

For psum1, the cycles per element are:

Elements	Cycles
10	2853
20	735
30	723
40	783
50	852
60	930
70	1299
80	1089
90	1182
100	1263
110	1557
120	1419
130	1509

140	1593
150	1683
160	1962
170	1848
180	2136
190	2001
200	2079



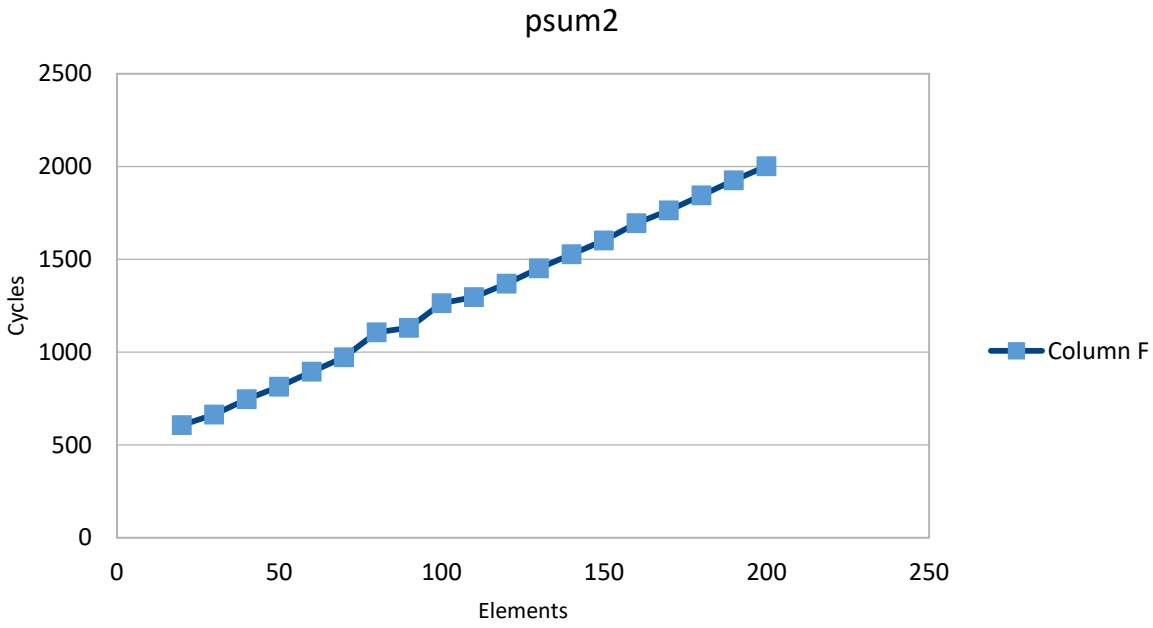
slope without outlier = 8.2 = CPE

The small jumps in the graph are a result of the data sizes stretching to higher levels of cache. So the first jump is when the data size uses L2 cache, and the second jump is most likely the jump to L3 cache.

For psum2, the cycles per element are :

Elements	Cycles
10	906
20	606
30	663
40	747
50	813
60	894

70	972
80	1107
90	1131
100	1263
110	1296
120	1368
130	1452
140	1527
150	1602
160	1695
170	1794
180	1845
190	1926
200	2001



slope without outlier= 7.8 = cycles per element

These CPE results are not exact matches to psum1 and psum2 from B&O, but they are quite similar. My results confirm that psum2 has a lower CPE than psum1, and both graphs having linear slopes with psum1 being steeper than psum2. This is the effect of the loop unrolling in psum2.

******** My new test_psum.c is attached to the submission **********

5.

1.
Starting a loop
done
real 0m0.211s
user 0m0.208s
sys 0m0.002s

2.
Starting a loop
done
real 0m0.006s (The execution time has noticeably dropped to almost 0s.)
user 0m0.000s
sys 0m0.001s

4.
The assembly code for test_O_level.c with optimization -O1 is has fewer lines than the code with no optimization. This is because the compiler was able to make the code more efficient. In the optimized assembly code, the compiler does not include the "steps" or "i" variable, which are both present in the non-optimized version. They are removed because the compiler realized that neither were being used after the loop, so it decided to remove them.

5.
For -O0 optimization:

Starting a loop
300000003
done
real 0m0.203s
user 0m0.200s
sys 0m0.001s

For -O1 optimization

Starting a loop
300000003
done
real 0m0.003s
user 0m0.001s
sys 0m0.000s

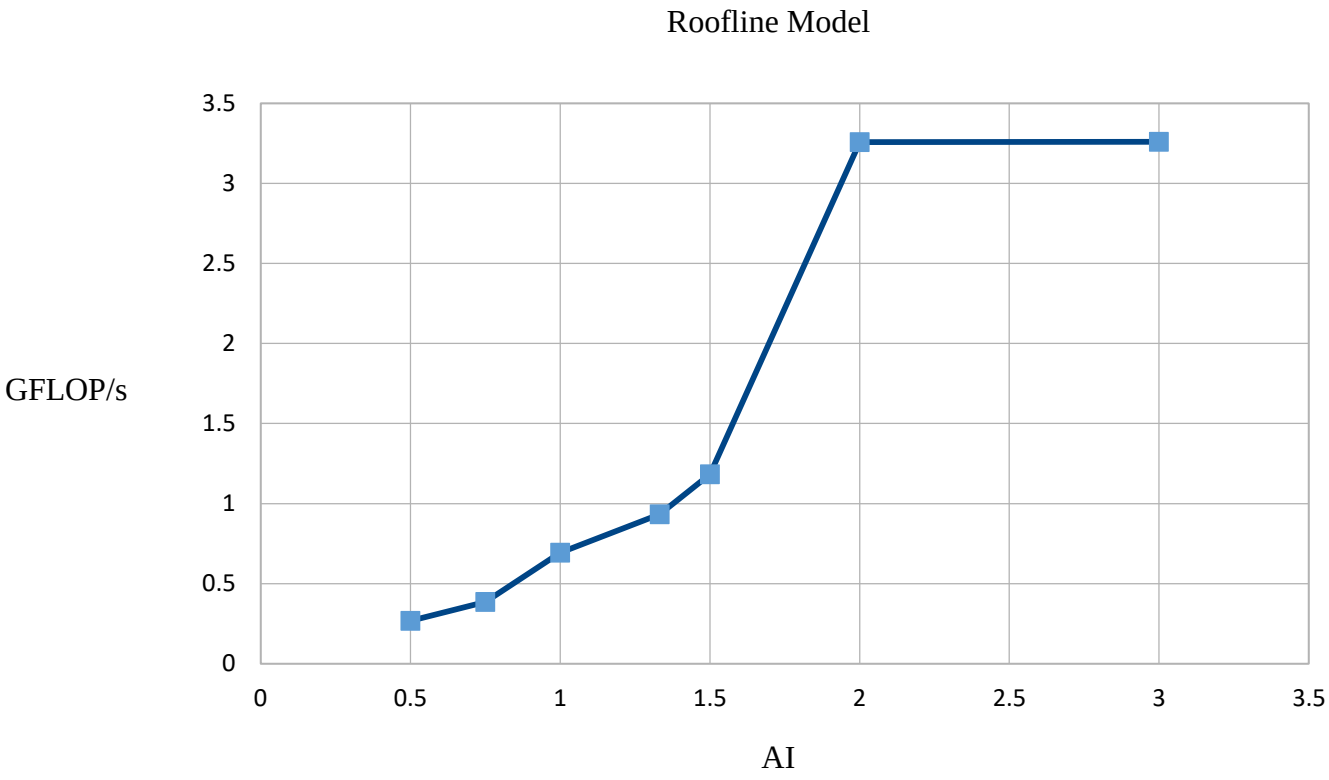
The code has now been optimized because in the assembly language with the -O1 optimization, the compiler knew to just print 300000003 instead of progressing through the loop, because the compiler knew before runtime what the final value of steps would be.

6.

1. The payload loop is the loop within the void twice_memory_Copy(int n) function.

2. The memory bandwidth is close to 8,000 MB which is equal to 8 GB. The memory bandwidth of the CPU is 32 GB, so the program is using about ¼ of the total bandwidth.

4.



5. I can conclude that the GFLOP/s peaks at 3.25 attainable GFLOP/s, which is first reached when the Arithmetic Intensity reaches 2. Prior to this point, the program is memory bound because it has a low AI. At every point with an AI greater than 2, the GLOP/s will remain at 3.25 because it is now CPU bound.

7.

1. I felt as if I possessed all the skills necessary to complete this lab assignment.
2. This lab took me between 5-6 hours to complete.
3. No part of the lab took an unreasonable amount of time to complete, but I felt as if some of the wording in part 2 was slightly unclear and vague. I felt as if the instructions for that part should have been more specific in regards to calibration.
4. Besides the aforementioned statement, I did not see any problems with this lab.