Lab 0 — Getting Started

Part 1. Find machine characteristics

1a:

The CPU are using is Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz.
The operating frequency is 3GHz..
There are 8 cores.

1b:

There are 3 levels of cache.
The L1d and L1i cache has 32K, L2 cache has 256K, and L3 cache has 12288K.
The microarchitecture of the processor cores is x86_64.
The number of cores is 8 and the number of physical id of the cores is 8.
So, the cores are real.
According to the information online, the Max Memory Bandwidth is 41.6 GB/s.
Its base frequency is 3.0GHz and its Max Turbo Frequency is 4.70GHz.

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

2a:

The result shows that:
real 0m3.695s
user 0m3.661s
sys 0m0.011s
According to the figure1, the definition of the accuracy is the closeness of the measured value to a standard or true value. So, we can determine the accuracy of the timer by the closeness of its overall time recording result to the sys output.
Also, for the precision, if the range of the numbers are large, it has a low precision; on the other hand, if the range is small, it has high precision.
As the clock_gettime() has the resolution of nanosecond, and gettimeofday() has the resolution of microseconds, so the gettimimeofday() is more accurate.

2b:

The problem with RDTSC is that you have no guarantee that it starts at the same point in time on all cores of an elderly multicore CPU, and no guarantee that it starts at the same point in time on all CPUs on an elderly multi-CPU board.
Modern systems usually do not have such problems, but the problem can also be worked around on older systems by setting a thread's affinity so it only runs on one CPU.

2c:

The GET_TOD_TICS, CLK_RATE and GET_SECONDS_TICS are the parameters maybe needed to change for each timer.
The GET_TOD_TICS does not need to change since the function gettimeofday() only counts the difference between the seconds, and the function itself has calculated the transition of seconds and microseconds, so there is nothing to change.
The CLK_RATE is for the RDTSC, and it is based on the time-stamp. So, the clock rate should be changed according to different computers. As the CPU in the lab has the frequency of 3.0GHz, the CLK_RATE needs to be changed to 3.0e9.

The GET_SECONDS_TICS should be calculated in timers_1_scaling.c, and the _SC_CLK_TCK defines the number of ticks in the linux system. So, it should be 100.

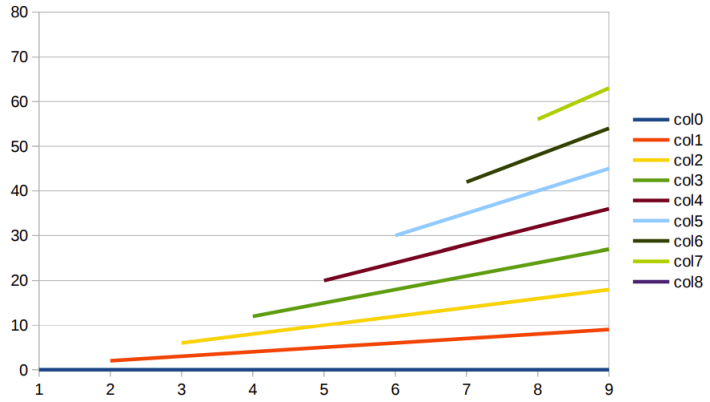
2d, 2e and 2f:

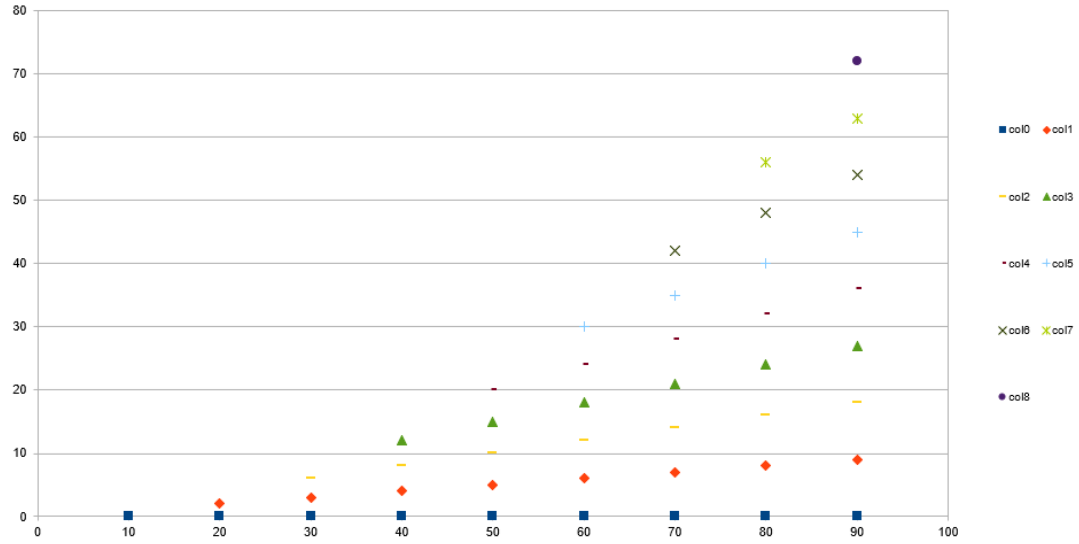
Shown in test_clock_gettime.c.

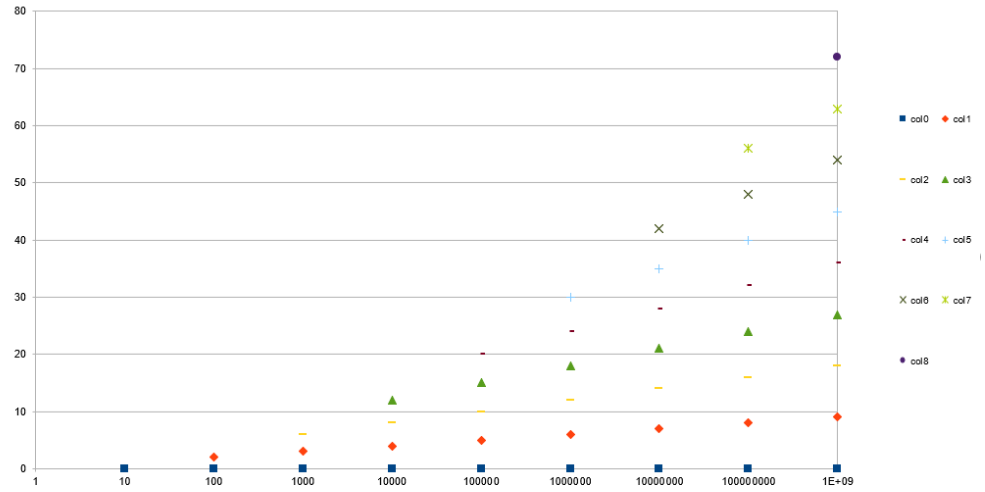
The resolution of the measurements given by clock_gettime is ns, that is to say, it hast the resolution of 10e-9s.

After trying 15 times, the differences calculated are 1.008247328, 1.005571848, 1.009572221, 1.008026257, 1.007845951, 1.008493109, 1.007978212, 1.008137999, 1.008206736, 1.007709298, 1.008493949, 1.007785199, 1.008130522, 1.008472485, 1.007902463. And their standard deviation is 0.00079s.

Part 3. Plotting/Graphing Data.



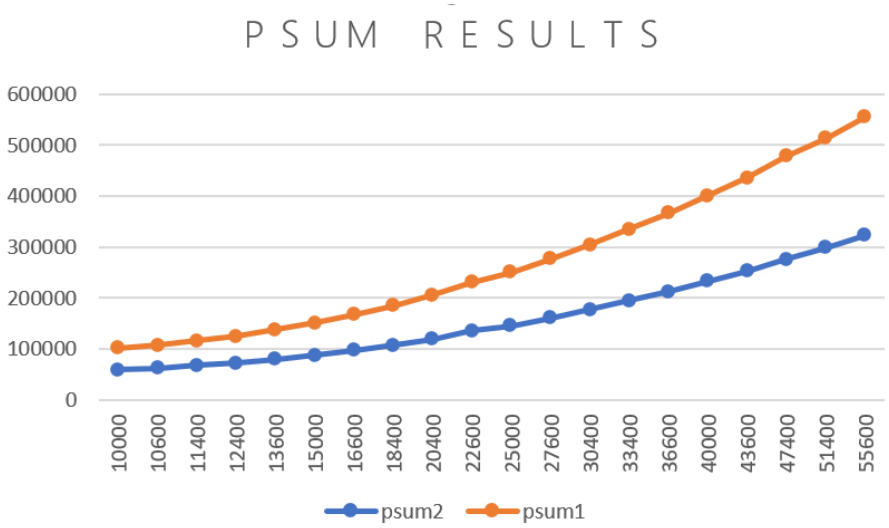




Part 4. Performance evaluation using Cycles-per-Element (CPE).

4a:

4b:



4c:

To deal with the anomalies, we may need to remove the redundant data from the database.

4d:

The CPE for psum1 is 5.779, and for psum2 is 4.19. The reason behind it maybe the gettime() function based on the changing frequencies, which making the results changed from time to time.

Part 5. Interacting with the compiler.

5a:

real 0m0.565s
user 0m0.559s
sys 0m0.004s

5b:

real 0m0.055s
user 0m0.048s
sys 0m0.004s

5c:

5d:

The whole process of the loop becomes shorter, and the steps in the loop get smaller. For as the optimized of the codes, the CPU does not calculated the quasi_random at all since it is not printed.

5e:

For the 00, the output is:

real 0m0.565s
user 0m0.562s
sys 0m0.001s

For the 01, the output is:

real 0m0.562s
user 0m0.559s
sys 0m0.003s

The time of the whole codes get slows down because of the output of the quasi_random requires the CPU to calculate it.

Part 6: Generating roofline plots

6b:

It does the same operation multiple times and takes the best value as the end result.

6c:

The output of the stream.c is:

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy:	24463.8775	0.0014	0.0014	0.0015
Scale:	23913.1035	0.0015	0.0015	0.0015
Add:	25226.0225	0.0021	0.0021	0.0021
Triad:	24432.8388	0.0022	0.0022	0.0022

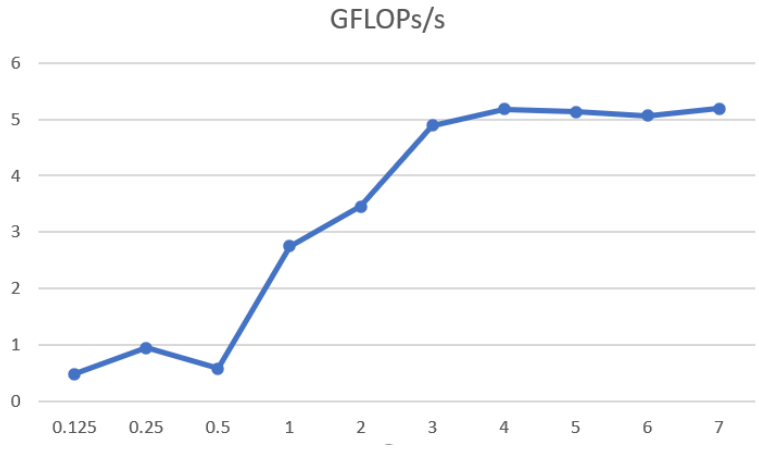
Average MB/s over the four tests:
24508.9606
Solution Validates

As we can see, the bandwidth is 24508MB/s, which is much smaller than the 41.6GB/s, as the documents shown online.

6d:

The GFLOPs/s for AI=1/8 is: 0.489957
The GFLOPs/s for AI=1/4 is: 0.951586
The GFLOPs/s for AI=1/2 is: 0.581604
The GFLOPs/s for AI=1.0 is: 2.722377
The GFLOPs/s for AI=2.0 is: 3.549086
The GFLOPs/s for AI=3.0 is: 4.983474

The GFLOPs/s for AI=4.0 is: 5.191539
The GFLOPs/s for AI=5.0 is: 5.122549
The GFLOPs/s for AI=6.0 is: 5.089410
The GFLOPs/s for AI=7.0 is: 5.091758
The figure shows below:



6f:
It can be seen from the plot above that the GFLOPS/s plateaus roughly after arithmetic intensity of 3. The GFLOPS/s plateaus roughly after this point because there is a limit to the performance of the CPU. The GFLOPS/s is bounded by the maximum memory bandwidth of the CPU, even if arithmetic intensity is increased past a value of 3.

Part7: Quality Control
7a: I haven't learnt about Assembly language.
7b: About 8 hours.
7c: Spent a lot of time debugging part4 for a simple mistake, and spend some time figuring out what part6 says.
7d: No.