Evan Lang

Yang Lu

EC527 Lab 0

1a.

· Intel(R) Xeon(R) CPU E5-2640 v4

· At the time of the query the cpu was operating at 1256.982 MHz, its main operating frequency is 2.4Ghz

· The cpu has 20 “cores”, with 10 being real cores

1b.

· The CPU has 3 layers of cache. There is 64KB of L1 cache per core, with 32KB dedicated to instructions and 32 KB dedicated to data. There is also 256 KB of L2 Cache per core. Finally there is 25MB of L3 cache shared between the cores. All of the above are set-associative cache.

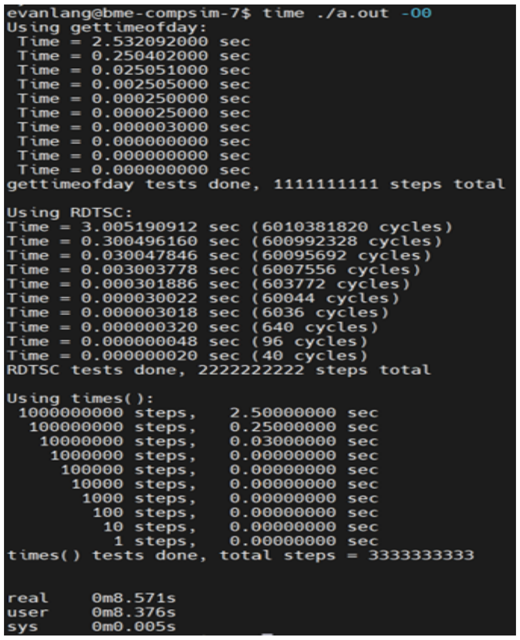
· The cores are Broadwell architecture

· While eng-grid reports 20 cores, the true number of real cores is 10. This cpu contains 2 threads per core. That means it can hypothetically act as a 20 core processer, with 10 real cores and 10 virtual cores.

· The max memory bandwidth (direct from Intel’s specs online) is 68.3 GB/s. This is the max read/write speed of the cpu.

· The base frequency of the processor is 2.4Ghz. It can run as low as 1.2Ghz and it can boost to 3.4Ghz.

2a. test_timers.c output:



```
evanlang@bme-compsim-7$ time ./a.out -O0
Using gettimeofday:
 Time = 2.532092000 sec
 Time = 0.250402000 sec
 Time = 0.025051000 sec
 Time = 0.002505000 sec
 Time = 0.000250000 sec
 Time = 0.000025000 sec
 Time = 0.000003000 sec
 Time = 0.000000000 sec
 Time = 0.000000000 sec
 Time = 0.000000000 sec
gettimeofday tests done, 1111111111 steps total

Using RDTSC:
Time = 3.005190912 sec (6010381820 cycles)
Time = 0.300496160 sec (600992328 cycles)
Time = 0.030047846 sec (60095692 cycles)
Time = 0.003003778 sec (6007556 cycles)
Time = 0.000301886 sec (603772 cycles)
Time = 0.000030022 sec (60044 cycles)
Time = 0.000003018 sec (6036 cycles)
Time = 0.000000320 sec (640 cycles)
Time = 0.000000048 sec (96 cycles)
Time = 0.000000020 sec (40 cycles)
RDTSC tests done, 2222222222 steps total

Using times():
 1000000000 steps,   2.50000000 sec
  100000000 steps,   0.25000000 sec
   10000000 steps,   0.03000000 sec
    1000000 steps,   0.00000000 sec
     100000 steps,   0.00000000 sec
      10000 steps,   0.00000000 sec
       1000 steps,   0.00000000 sec
        100 steps,   0.00000000 sec
         10 steps,   0.00000000 sec
          1 steps,   0.00000000 sec
times() tests done, total steps = 3333333333


real    0m8.571s
user    0m8.376s
sys     0m0.005s
```

· Gettimeofday had a resolution of 7 data points, and a medium-high accuracy. It should be noted that the precision decreased as the number of steps increased.

· RDTSC had the highest resolution with 10 usable data points. However it has the lowest accuracy and precision

· times has the lowest resolution with 3 usable data points, however it has both the highest accuracy and precision

· Scientists can use an atomic clock, which uses the decay of particles to measure time and is the most precise way to gauge accuracy. I can manually time the code myself externally and compare it to the results. I can also use a kernel level timer around the code execution and compare the results. While it is useless for measuring the accuracy of each step, it will favorably measure the accuracy of the avg total result of the timer.

2b.

· RDSTC relies on increasing a time counter by 1 every cycle of the cpu. However modern cpu's have variable clock speeds for the cores leading to an inconsistent time per each cycle. It is still useful if you are able to lock the cpu frequency as you can use the locked frequency to find the time by multiplying it by the number of cycles. Generally, the base frequency is a good metric to

2c.

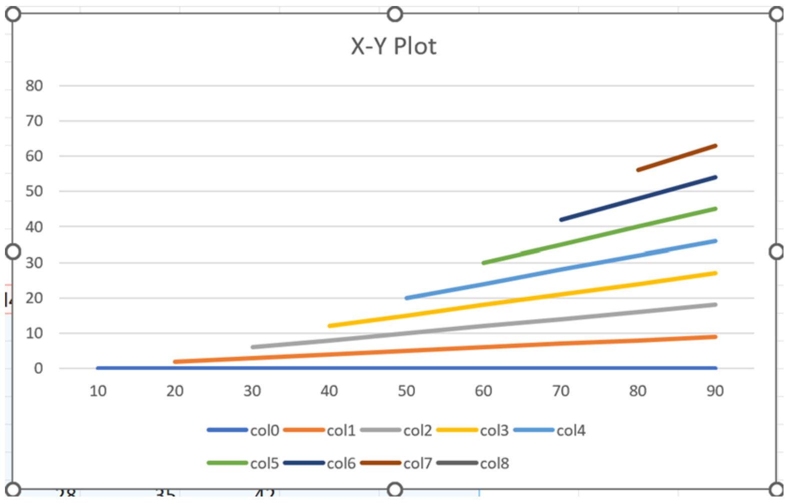
· I changed the value of the Clk rate from 2 Ghz to 2.4 Ghz to match the locked speed of the cpu I am using.

2d-f.

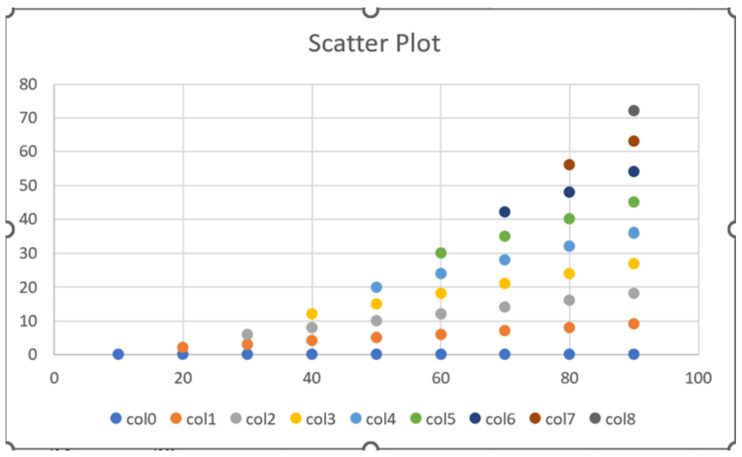
· Contained in test_clock_gettime.c

3.

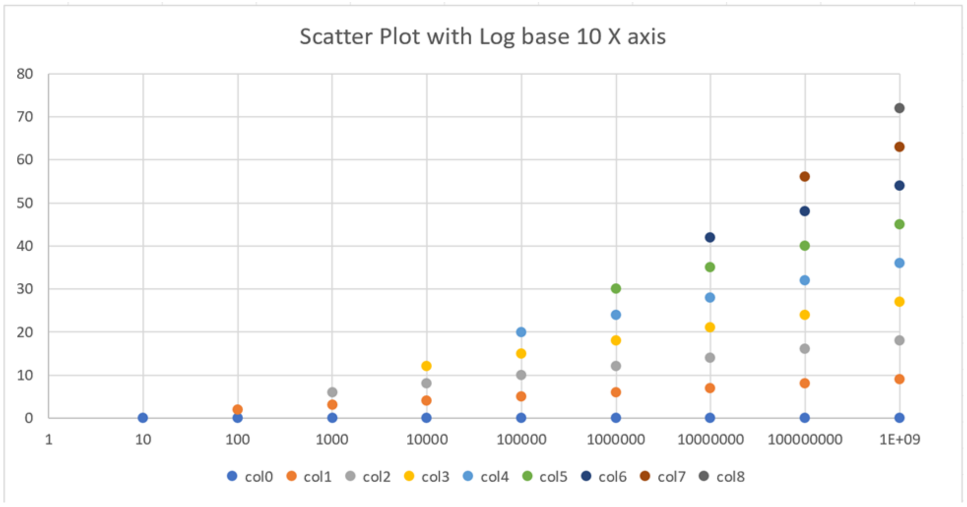
X-Y Plot:



Scatter Plot:

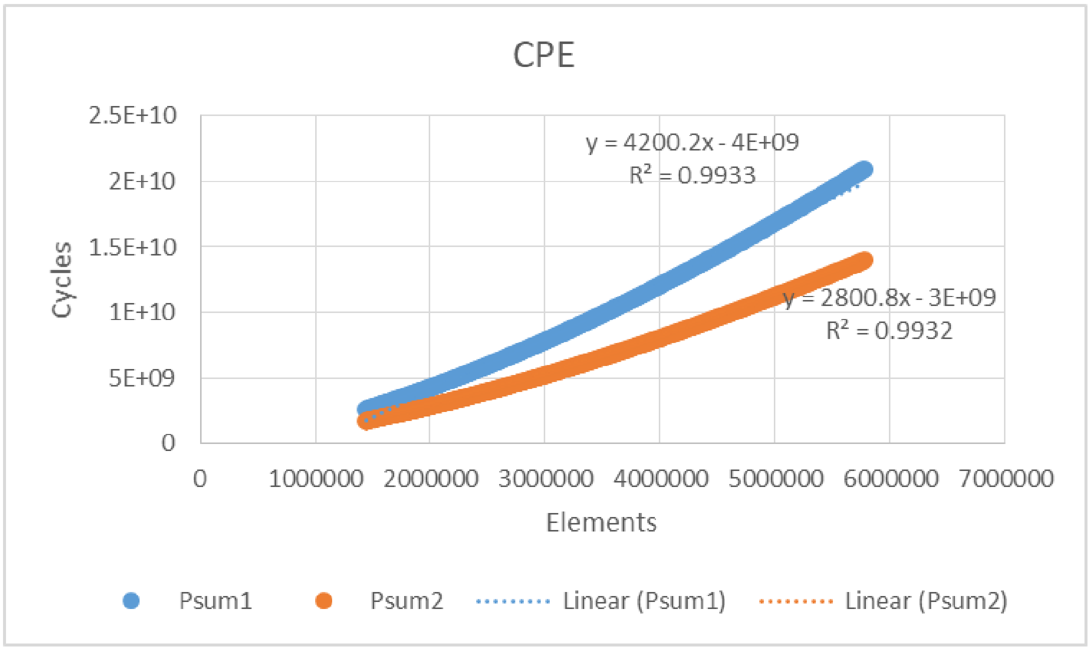


Scatter Plot with Logarithmic X axis(Using the multiples of base 10 version of the data for better looking results):



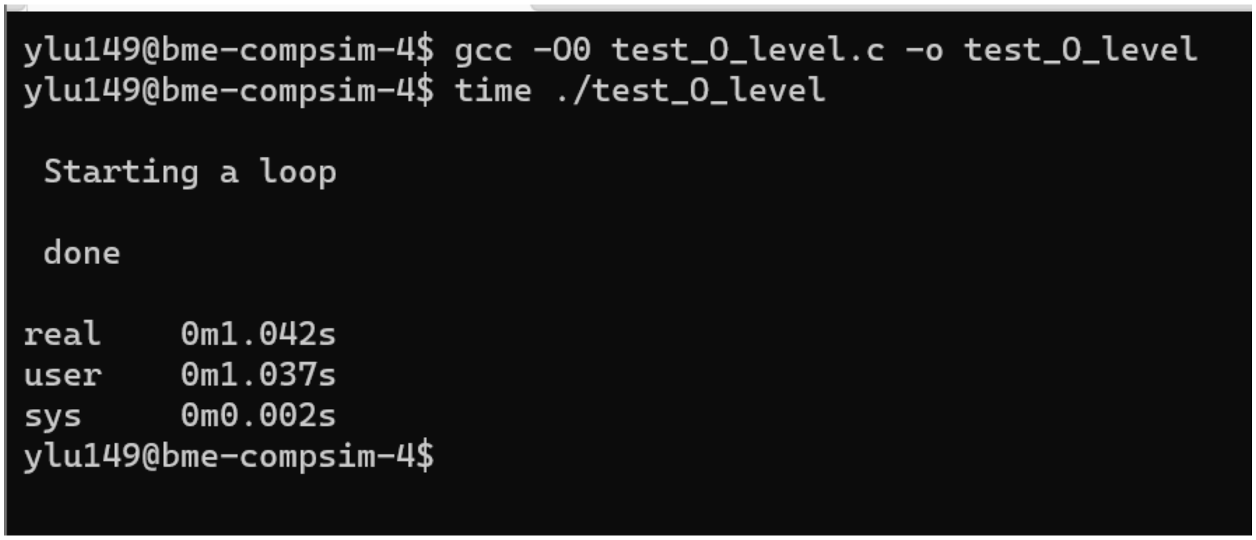
4b. It was observed that the slope of each run would increase as the runs went on. By changing the input variable C, one could vary the starting location of data collection and concatenate the results of smaller runs . This could prevent the slope creep we are currently observing in our data. Furthermore, ensuring the aforementioned inputs are valid and therefore the psum calculations stay within the MAX_INT size, prevent bits from overflowing and the sum resetting to 0.

4c.



It takes about 4200 CPE for psum1 and 2800 CPE for psum2. Although the ratio between psum1 to psum2 are similar, it is not at all close to the textbooks calculated slopes. I believe that this is because the graph provided by the textbook is still able to store all the values in the cache. While in our case, the array size has exceeded the bounds of the memory storage, hence it requires more time and therefore cycles to proceed.

5a.



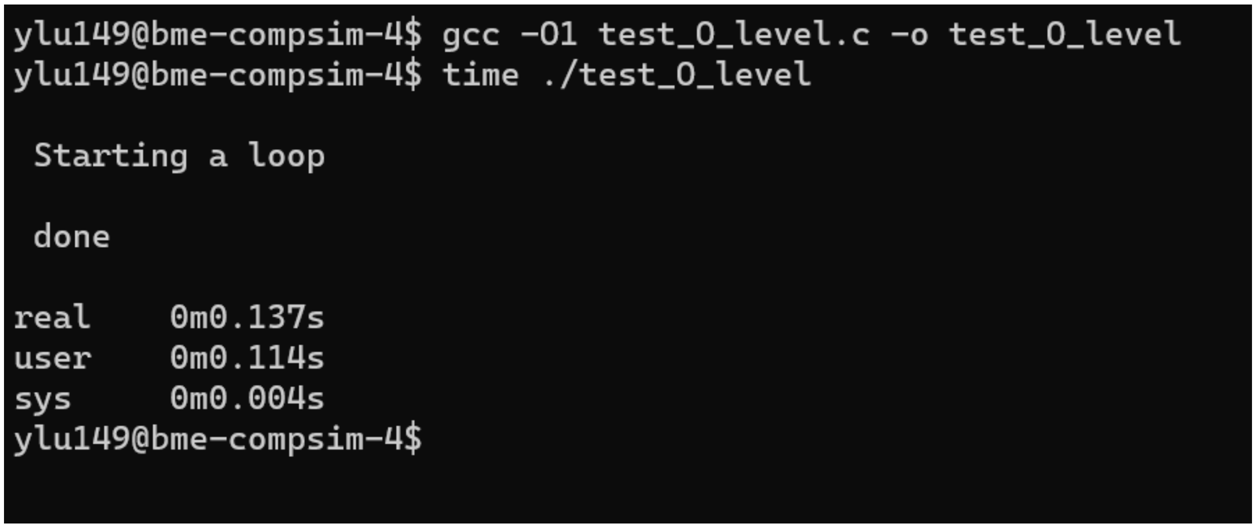
```
ylu149@bme-compsim-4$ gcc -O0 test_O_level.c -o test_O_level
ylu149@bme-compsim-4$ time ./test_O_level

 Starting a loop

 done

real    0m1.042s
user    0m1.037s
sys     0m0.002s
ylu149@bme-compsim-4$
```

5b.



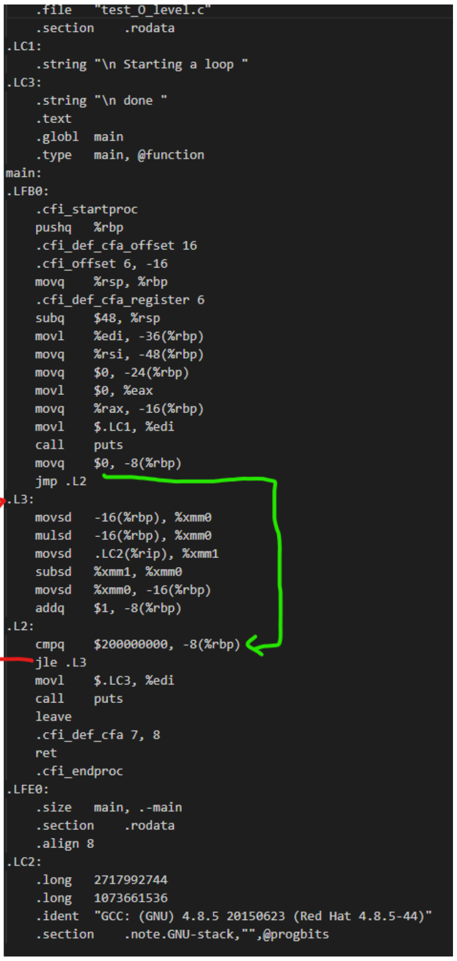
```
ylu149@bme-compsim-4$ gcc -O1 test_O_level.c -o test_O_level
ylu149@bme-compsim-4$ time ./test_O_level

 Starting a loop

 done

real    0m0.137s
user    0m0.114s
sys     0m0.004s
ylu149@bme-compsim-4$
```

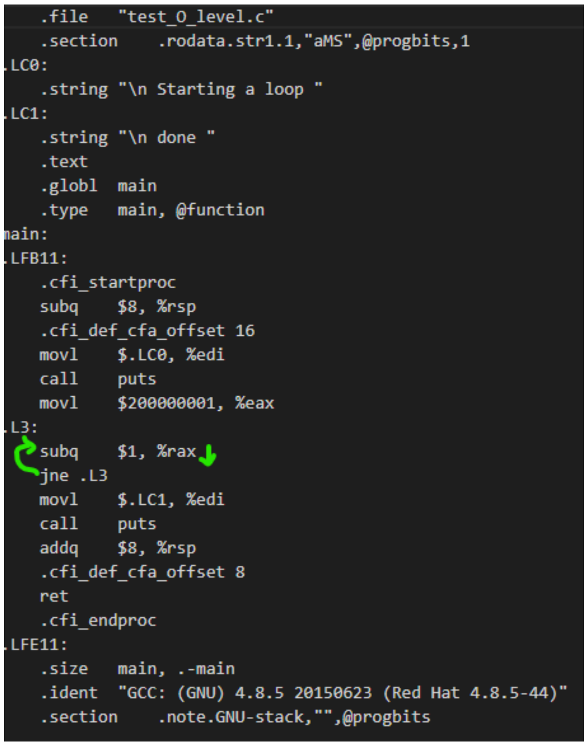
5c.



```
    .file   "test_O_level.c"
    .section    .rodata
.LC1:
    .string "\n Starting a loop "
.LC3:
    .string "\n done "
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $48, %rsp
    movl    %edi, -36(%rbp)
    movq    %rsi, -48(%rbp)
    movq    $0, -24(%rbp)
    movl    $0, %eax
    movq    %rax, -16(%rbp)
    movl    $.LC1, %edi
    call    puts
    movq    $0, -8(%rbp)
    jmp .L2
.L3:
    movsd   -16(%rbp), %xmm0
    mulsd   -16(%rbp), %xmm0
    movsd   .LC2(%rip), %xmm1
    subsd   %xmm1, %xmm0
    movsd   %xmm0, -16(%rbp)
    addq    $1, -8(%rbp)
.L2:
    cmpq    $200000000, -8(%rbp)
    jle .L3
    movl    $.LC3, %edi
    call    puts
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .section    .rodata
    .align 8
.LC2:
    .long   2717992744
    .long   1073661536
    .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
    .section    .note.GNU-stack,"",@progbits
```

The loop occurs in .L3, when it hits the jle call in .L2 it loops back to .L3 until the comparison is satisfied.

5d.

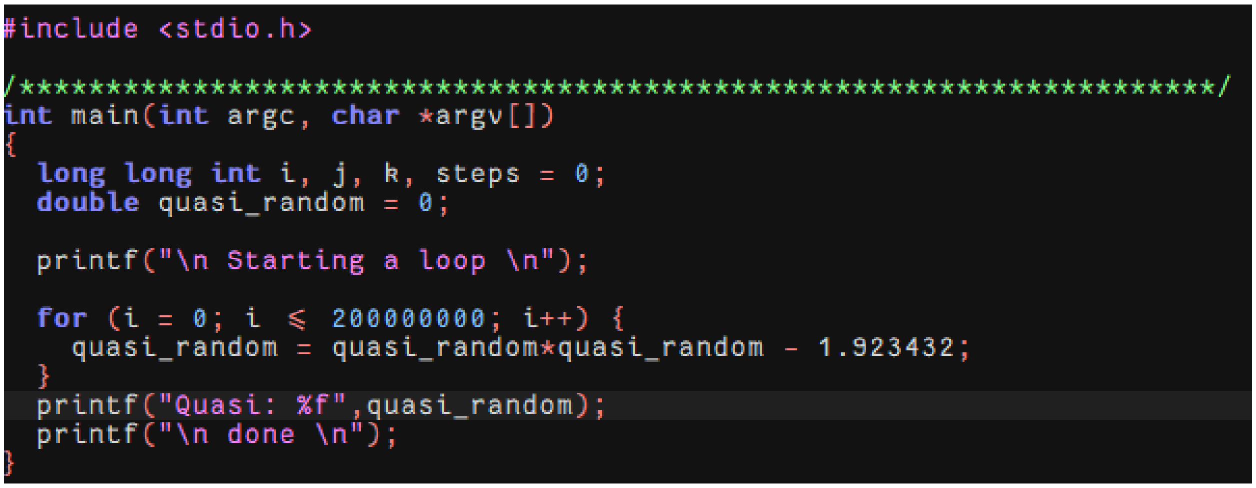


```
    .file   "test_O_level.c"
    .section    .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "\n Starting a loop "
.LC1:
    .string "\n done "
    .text
    .globl  main
    .type   main, @function
main:
.LFB11:
    .cfi_startproc
    subq    $8, %rsp
    .cfi_def_cfa_offset 16
    movl    $.LC0, %edi
    call    puts
    movl    $200000001, %eax
.L3:
    subq    $1, %rax
    jne .L3
    movl    $.LC1, %edi
    call    puts
    addq    $8, %rsp
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
.LFE11:
    .size   main, .-main
    .ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
    .section    .note.GNU-stack,"",@progbits
```

The loop now occurs entirely in .L3, with it looping back only to subq $1, %rax before once again using the jne call.

5e.

New code:



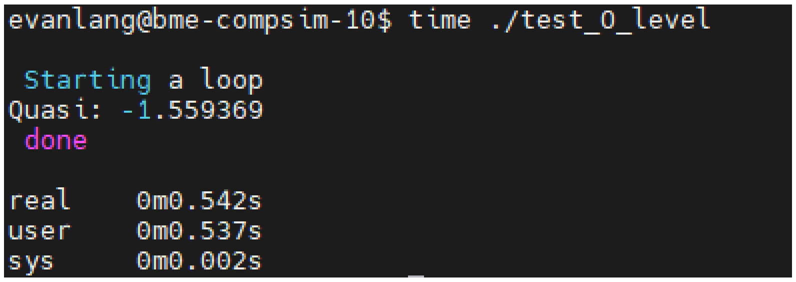
```c
#include <stdio.h>

/**********************************************************************/
int main(int argc, char *argv[])
{
  long long int i, j, k, steps = 0;
  double quasi_random = 0;

  printf("\n Starting a loop \n");

  for (i = 0; i <= 200000000; i++) {
    quasi_random = quasi_random*quasi_random - 1.923432;
  }
  printf("Quasi: %f",quasi_random);
  printf("\n done \n");
}
```

New runtime:

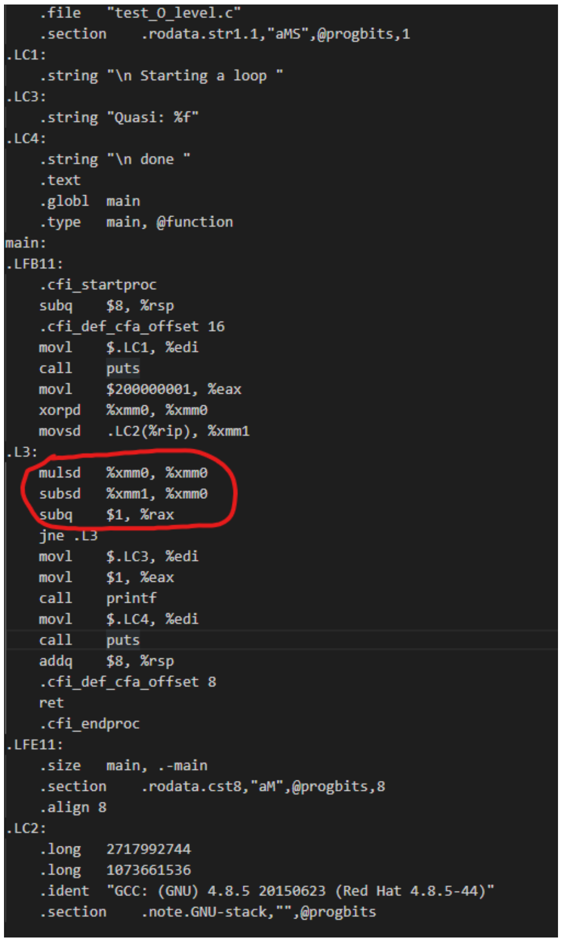


```
evanlang@bme-compsim-10$ time ./test_O_level

 Starting a loop
Quasi: -1.559369
 done

real	0m0.542s
user	0m0.537s
sys	0m0.002s
```

New Assembly:



```
	.file	"test_O_level.c"
	.section	.rodata.str1.1,"aMS",@progbits,1
.LC1:
	.string "\n Starting a loop "
.LC3:
	.string "Quasi: %f"
.LC4:
	.string "\n done "
	.text
	.globl	main
	.type	main, @function
main:
.LFB11:
	.cfi_startproc
	subq	$8, %rsp
	.cfi_def_cfa_offset 16
	movl	$.LC1, %edi
	call	puts
	movl	$200000001, %eax
	xorpd	%xmm0, %xmm0
	movsd	.LC2(%rip), %xmm1
.L3:
	mulsd	%xmm0, %xmm0
	subsd	%xmm1, %xmm0
	subq	$1, %rax
	jne .L3
	movl	$.LC3, %edi
	movl	$1, %eax
	call	printf
	movl	$.LC4, %edi
	call	puts
	addq	$8, %rsp
	.cfi_def_cfa_offset 8
	ret
	.cfi_endproc
.LFE11:
	.size	main, .-main
	.section	.rodata.cst8,"aM",@progbits,8
	.align 8
.LC2:
	.long	2717992744
	.long	1073661536
	.ident	"GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
	.section	.note.GNU-stack,"",@progbits
```

Unlike before the assembly is now actually executing the multiplication and subtraction within the loop. Before it was “optimized out” as the results of that mathematics were never used. Since we are now requiring the final result of math in the print statement the compiler is forced to include it in the assembly

6a.

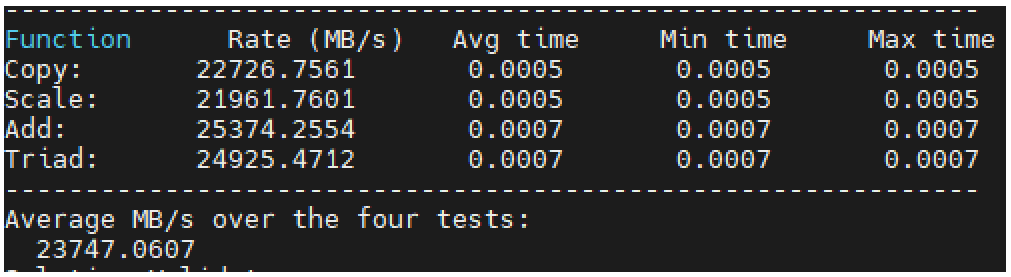
- Kernel 1 serves to test the memory bandwidth of the benchmarked processor by operating at a low arithmetic intensity. This guarantees that the memory bandwidth is not bound by the processor's compute capacity, allowing it to reach its peak bandwidth. This means the processor will not hit its peak computational performance during the run.
- Kernel 2 is the opposite, containing a high arithmetic intensity that forces the processor's compute performance to its limit. The resulting bottleneck will prevent the processor from pulling its max memory bandwidth, as it is already saturated with the bandwidth it needs.
  .

6b.

The loops mostly match my expectations, being split into memory streaming focused tests and arithmetic intensity based tests. (FLOP vs Stream). These should allow us to measure bandwidth in both high and low AI environments.

6c.

The peak memory bandwidth observed was 23747.0607 MB/s or 23.747GB/s. This is only ~1/3 of the stated bandwidth on Intel's specification page of 68.3 GB/s

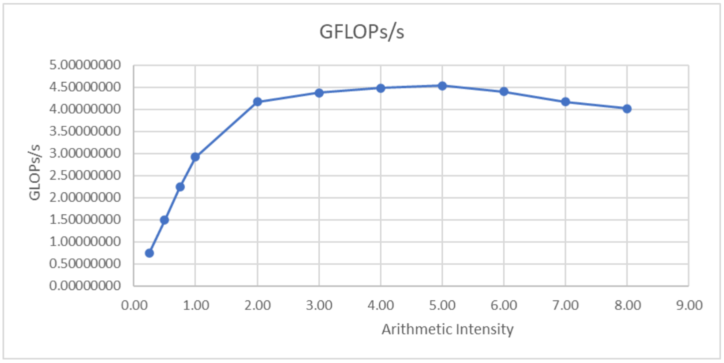


```
-------------------------------------------------------------
Function      Rate (MB/s)   Avg time     Min time     Max time
Copy:        22726.7561      0.0005       0.0005       0.0005
Scale:       21961.7601      0.0005       0.0005       0.0005
Add:         25374.2554      0.0007       0.0007       0.0007
Triad:       24925.4712      0.0007       0.0007       0.0007
-------------------------------------------------------------
Average MB/s over the four tests:
  23747.0607
```

6d.

AI	GFLOPs/s
0.25	0.73823400
0.50	1.48846800
0.75	2.24359000
1.00	2.92197400
2.00	4.16506100
3.00	4.36818300
4.00	4.47611600
5.00	4.53196300
6.00	4.39659400
7.00	4.17022800
8.00	4.01475

6e.



6f. According to our data, the max GFLOPS/s reached by our processor is 4.53196300 GFLOPs/s. This number was achieved at a AI of 4. Based on our results we can determine that arithmetic intensity and measured memory bandwidth initially have a proportional relationship. However, once the AI reaches a point (observed to be 2.0 in our data) the computational power of the processor is maxed out and the relationship becomes inverted. From this point on as AI increases, measured bandwidth decreases. The maximum GLOPs/s observed signals this point of inversion.

7a.

We were both missing experience using the C printf function. We were used to printing via cout and experienced multiple delays attempting to create the proper printf commands.

7b.

The lab took approximately 14 hours to complete.

7c.

Part 4 took a significant amount of time, specifically due to the aforementioned problems with printing.

7d.

We both did not know what exact metrics to use when measuring accuracy and precision in part 2. The instructions related to plotting assumed use of a lab computer. Since one of us was forced to only use eng-grid via mobaxterm these instructions were not relevant to them. The open-ended nature of many of the questions forced us to continuously rethink our answers. However, this is not necessarily a problem with the lab and should improve as we become more familiar with the expectations of the course.